HAL
open science

CEYLAN : Un canevas pour la création de gestionnaires autonomiques extensibles et dynamiques.

Yoann Maurel

To cite this version:

Yoann Maurel. CEYLAN : Un canevas pour la création de gestionnaires autonomiques extensibles et dynamiques.. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2010. Français. NNT : . tel-00545113v2

HAL Id: tel-00545113

https://theses.hal.science/tel-00545113v2

Submitted on 9 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE DE GRENOBLE

Spécialité INFORMATIQUE

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

Yoann MAUREL

LE 1 DECEMBRE 2010

CEYLAN : Un canevas pour la création de gestionnaires autonomiques extensibles et dynamiques.

Thèse dirigée par **Philippe LALANDA** et codirigée par **Ada DIACONESCU**

JURY :

PRESIDENTE	**GAËLLE CALVARY**, Professeur à l'Ensimag, Grenoble INP, Grenoble
RAPPORTEURS	**FRANÇOIS CHARPILLET**, Directeur de recherche à l'INRIA, Nancy
	ROY STERRITT, Lecturer in informatics, University of Ulster
EXAMINATEUR	**JULIE A.MCCANN**, Reader, Imperial College London
ENCADRANTS	**PHILIPPE LALANDA**, Professeur à l'Université Joseph Fourier, Grenoble
	ADA DIACONESCU, Maître de conférences, Telecom Paris-Tech, Paris

THESE PREPAREE AU SEIN DU **LIG** DANS **l'Ecole Doctorale MSTII**

(MATHEMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE)

RESUME

Les applications modernes sont de plus en plus dynamiques et hétérogènes. L'architecture des systèmes modernes n'est plus figée et prévisible. Il en va de même pour les besoins des utilisateurs, les capacités des ordinateurs et des réseaux, et les technologies utilisées. Il nous parait essentiel que les gestionnaires autonomiques soient dynamiquement adaptables et extensibles pour prendre en compte ces changements et faciliter la maintenance.

L'objectif de notre travail est de définir et d'implanter un cadriciel, ou framework, facilitant le développement de gestionnaires autonomiques. Dans cet effort, nous visons à définir un modèle architectural permettant le développement de gestionnaires autonomiques modulaires, homogènes, souples, évolutifs, dynamiques et administrables. Un objectif important de ce travail est de clairement définir et séparer les concepts appartenant de façon générique à un gestionnaire autonomique et les aspects métier, développés au cas par cas. Le but est de permettre au experts domaine de se concentrer sur l'écriture des fonctions autonomiques et de ne pas gérer entièrement l'enchaînement et le contrôle de ces fonctions.

Pour atteindre nos objectifs, nous avons tout d'abord défini la notion de tâche d'administration. Une tâche d'administration est une entité indépendante et spécialisée qui réalise une ou plusieurs fonctions d'administration. Le gestionnaire autonomique résulte de la combinaison opportuniste de ces tâches. A chaque instant, l'ensemble de tâches utilisées peut être modifié en fonction du contexte et des informations remontées par la plateforme. Des mécanismes de sélections permettent de gérer les conflits éventuels et permettent d'assurer une cohérence du comportement du gestionnaire.

Nous avons ensuite défini une architecture de gestion de ces tâches permettant la combinaison opportuniste de ces tâches en fonction du contexte. Cette architecture a été implantée sous la forme d'un framework fondé sur la technologie des composants orientés services. Le framework que nous avons développé fournit un cadre pour l'intégration dynamique de fonctions autonomiques et pour leur gestion en fonction du contexte et de politiques d'administration, elles mêmes évolutives. En conclusion de ce travail, nous présentons une implémentation particulière sous la forme d'un modèle à composants de sorte que le travail des développeurs est facilité et que la réutilisation est favorisée. Enfin nous donnons un exemple d'application développée au dessus de ce framework.

Mots-clefs : Informatique autonomique, Gestion autonomique, Composition dynamique, Cadriciel, Approche orientée services, Composant orienté services, OSGi, Cilia, iPOJO.

ABSTRACT

Modern applications are increasingly dynamic and heterogeneous. The architecture of modern systems is no longer rigid or predictable. This also applies to the user's needs, the capabilities of computers and networks, and the technologies we use. We consider essential that autonomic managers become dynamically extensible and adapt to these changes to facilitate the maintenance.

The objective of our work is to define and implement a framework that facilitates the development of autonomic managers. With that purpose, we define an architectural model for the development of modular, homogeneous, flexible, administrable and dynamic autonomic managers. An important objective of this work is to clearly define and separate the generic concepts belonging to autonomic managers from business aspects, developed on a case-by-case basis. The goal is to enable the experts to concentrate on writing autonomic functions and not to bother with management and control flow of these functions.

To achieve our objectives, we first define the concept of administration task. An administration task is an independent and specialized element that achieves one or more administrative functions. The autonomic manager behaviour results from the opportunistic combination of these tasks. At every moment, all tasks used can be changed depending on the context and the information given by the platform. Selection mechanisms help to manage potential conflicts and ensure consistency of manager behaviour.

We then define an architecture to manage tasks and their opportunistic cooperation depending on the context. The current framework prototype is based on a Service Oriented Component technology. The developed framework allows the dynamic integration of autonomic functions and their management depending on context and management politics, themselves evolving. In conclusion we present a particular implementation of the framework as a component model so that the work of developers is facilitated and that reuse is promoted. Finally, we give an example of an application developed based on this framework.

Keywords: Autonomic Computing, Autonomic Management, Dynamic Composition, Problem Solving Modules, Framework, Service-Oriented Computing (SOC), Service-Oriented Component, OSGi, Cilia, iPOJO.

REMERCIEMENTS

Je voudrais remercier les membres du jury qui m'ont fait l'honneur de participer à ma soutenance et qui ont su s'adapter remarquablement aux conditions météorologiques particulières. Je les remercie pour les remarques et commentaires qu'ils ont réalisés sur mes travaux.

Je tiens à remercier et témoigner ma reconnaissance à Philippe Lalanda et Ada Diaconescu pour toute l'aide qu'ils m'ont apportée durant la thèse, leur patience, leur écoute et les nombreuses discussions qui ont permis de développer et étoffer les travaux présentés dans ce manuscrit.

Je souhaite également remercier les membres de l'équipe ADELE pour leur bonne humeur et la bonne ambiance qu'ils font régner. Je remercie les responsables Jacky Estublier et Philippe Lalanda ; ceux avec qui j'ai travaillé : Clément et Johann qui m'ont encadré avant ma thèse et ceux que j'ai encadrés : Morgan et Bertrand. Ne pouvant être exhaustif, je tiens à remercier tout particulièrement Stéphanie et Vincent pour toutes nos discussions et leur aide et Eric pour sa bonne humeur communicative.

Enfin, je tiens à remercier ma famille et tout particulièrement mon père et Magali pour l'attention et le soutien qu'ils m'ont apportés pendant cette période.

Table des matières

Chapitre 1 - Introduction 8
1 Contexte : L'informatique autonomique 9
2 Problématique 14
3 Objectifs de cette thèse 15
4 Structure du document 16
Chapitre 2 - Informatique autonomique 18
1 Influences 19
2 Définitions 21
2.1 Informatique autonomique 21
2.2 Système autonomique 22
2.3 Contexte 23
3 Propriétés d'un système autonomique 25
3.1 Capacités autonomiques et qualités 25
3.2 Taxonomie 26
4 Architecture des systèmes autonomiques 29
4.1 Principaux éléments 29
4.2 Architecture logique d'un gestionnaire autonomique 33
4.3 Choix d'implantation 37
5 Exemples de systèmes existants 40
5.1 Rainbow 40
5.2 Jade 41
5.3 Unity 43
5.4 Autonomia 45
5.5 Automate 46
5.6 Auto-Home 48
6 Evaluation des systèmes autonomiques 51
7 Synthèse 54
Chapitre 3 - Approches orientées service 56
1 Une réponse à deux besoins industriels 57
2 Principes 58
2.1 La notion de service 58
2.2 L'approche à services, un style architectural 59
2.3 Une approche SOC, des architectures SOA 61
2.4 Notion d'application à services 63
2.5 Dynamisme dans l'approche à services 66
3 Exemples d'architectures à services 67
3.1 Les services web 67
3.2 OSGi 71
4 Les approches à composants orientés services 75
4.1 Composant et modèle à composants 75
4.2 Orientation service des modèles à composants 77
4.3 Service Component Architecture (SCA) 78
4.4 iPOJO 79
5 Synthèse 81
Chapitre 4 - Proposition 82
1 Problématique 83
2 Objectifs 85
3 Notre approche 86
3.1 Principes 86
3.2 Tâches d'administration 88
3.3 Architecture de gestion des tâches 90
4 Synthèse 92
Chapitre 5 - Framework proposé 94
1 Tâches d'administrations 95
1.1 Architecture interne globale 95
1.2 Les ports et les données manipulées par la tâche 98

1.3 Le scheduler : déclenchement de la tâche ... 102
1.4 Le coordinateur : Autorisation de traitement des donnees ... 106
1.5 Le processeur : fonctionnalité et rôle de la tâche ... 108
1.6 Interfaces d'administration, cycle de vie et configuration de la tâche ... 112
1.7 Notion de Type de tâches ... 114
2 Contrôle des tâches et du gestionnaire ... 117
2.1 Structure générale ... 117
2.2 Administration de l'ensemble de tâches : cycle de vie et configuration ... 118
2.3 Mécanisme de communication ... 121
2.4 Sélection et résolution de conflits ... 124
2.5 Base de données ... 133
3 Administration du gestionnaire : construction et adaptation ... 135
3.1 Construction ... 135
3.2 Adaptation ... 137
4 Tâches composites ... 139
5 Synthèse ... 142
Chapitre 6 - Réalisation ... 144
1 Modèle de développement ... 145
1.1 Approche à services et modèle à composants ... 145
1.2 Plateforme d'exécution utilisée : OSGi, iPOJO et Cilia ... 147
2 Implémentation d'une tâche ... 151
2.1 Type de données et de tâches ... 151
2.2 Nature des sous-modules d'une tâche ... 153
2.3 Déclaration d'une implémentation de tâche atomique ... 154
2.4 Implémentation de ports ... 155
2.5 Implémentation de scheduler ... 157
2.6 Implémentation du coordinateur ... 158
2.7 Implémentation du processeur et configuration de la tâche ... 158
3 Implémentation des composites ... 161
3.1 La Communication, l'administration et la base de données ... 161
3.2 Mécanismes de sélection ... 162
4 Construction et administration du gestionnaire ... 164
4.1 Construction du gestionnaire ... 164
4.2 Administration du gestionnaire ... 165
5 Synthèse ... 168
Chapitre 7 - Validation ... 170
1 Application de surveillance de domicile ... 171
2 Gestion de l'application ... 173
2.1 Une première gestion de la mémoire et du processeur ... 174
2.2 Gestion du dynamisme et adaptation ... 176
2.3 Gestion des caméras ... 178
2.4 Gestion des alarmes ... 179
2.5 Algorithme de compression ... 181
2.6 Vers une meilleure gestion de l'espace disque ... 182
2.7 Utilisation conjointe des tâches développées. ... 184
2.8 Performances ... 185
3 Synthèse ... 186
Chapitre 8 - Conclusion ... 188
1 Synthèse ... 189
1.1 Contexte ... 189
1.2 Notre contribution ... 190
2 Perspectives ... 192
2.1 Auto-adaptation du gestionnaire ... 192
2.2 Distribution ... 193
2.3 Atelier et outils de développement ... 193
Chapitre 9 - Bibliographie ... 194

Chapitre 1 - **Introduction**

Dans ce premier chapitre, nous présentons le contexte de notre travail, ainsi que les objectifs précis que nous avons poursuivis. Nous détaillons également la structure de ce document de thèse.

1 CONTEXTE : L'INFORMATIQUE AUTONOMIQUE

L'informatique autonomique est un domaine récent, apparu pour faire face à la forte augmentation de la complexité des logiciels ces dernières années. Les applications logicielles d'aujourd'hui se caractérisent en effet par des tailles de plus en plus importantes, des besoins en dynamisme accrus, l'utilisation généralisée de composants tiers hétérogènes, une forte distribution de ces mêmes composants au travers de réseaux privés ou publics, et, parfois, par une diffusion massive au sein du grand public.

Ces évolutions, détaillées ci-après, posent de sérieux problèmes, non seulement pour le développement des logiciels, mais aussi pour leur maintenance. Les opérations de mise à jour sont ainsi devenues difficiles, couteuses et, de plus en plus, sources d'erreurs. La situation est telle que la maintenance constitue, dans de nombreux cas, un péril pour la qualité des logiciels. Il est souvent préférable de se passer d'une évolution fonctionnelle plutôt que de risquer des effets de bord non maîtrisés lors de la modification d'une application.

Réduire la complexité et les coûts de maintenance des logiciels n'a pourtant pas toujours été la priorité. Depuis les années 80, une grande partie des efforts de recherche s'est focalisée sur l'augmentation des performances matérielles, l'amélioration de la qualité des logiciels, ainsi que sur la baisse des coûts en général. De réels progrès ont été accomplis. En particulier, les performances matérielles ont crû de façon exponentielle [1] et, bien qu'elles commencent à être remises en question, les lois empiriques de Moore se vérifient encore. On constate que les capacités de stockage et la puissance des processeurs doublent approximativement tous les 18 mois. Ainsi, les ordinateurs ont atteint la puissance nécessaire pour faire fonctionner très confortablement en parallèle les logiciels « traditionnels » que sont, par exemple, les logiciels de bureautique, les jeux ou les applications multimédias. En parallèle, une importante chute du prix du matériel a permis l'apparition les ordinateurs dans tous les lieux de vie. La plupart des foyers de sociétés industrielles possèdent un ordinateur et chaque individu est amené à utiliser plusieurs ordinateurs [2]. Au sein des entreprises, les parcs informatiques ont changé de façon radicale. Jusqu'à la fin des années 70, l'informatique était très centralisée avec généralement l'usage d'un nombre limité de serveurs accédés à distance par des terminaux. La baisse du coût des équipements a permis le passage progressif des mainframes au modèle de l'informatique individuelle. Une entreprise possède aujourd'hui des parcs constitués de centaines, voire de milliers d'ordinateurs connectés en réseaux. De nos jours, un ordinateur qui n'appartient pas à un réseau fait figure d'exception, et ce même dans le grand public.

Cette augmentation des performances s'est accompagnée d'une meilleure maîtrise des techniques de développement. Le Génie Logiciel [3], au travers de la définition de processus logiciel, de méthodes et de langages, a contribué à l'amélioration de la qualité globale des applications logicielles. L'apparition des langages orientés objet dans les années 90 a, par exemple, permis de le rapprochement des phases de conception et de développement. Récemment, l'émergence de l'informatique dirigée par les modèles a apporté des outils facilitant la conception des logiciels et leur projection vers des plates-formes d'exécution. Néanmoins, la rapidité de changement des paradigmes utilisés pose problème : elle laisse les systèmes d'information dans des états de forte hétérogénéité.

Tous ces progrès ont changé la place de l'informatique dans notre société. De nombreux pans de notre économie et de notre vie sociale reposent sur l'usage de logiciels. Au sein des entreprises, la quasi-totalité des fonctions sont aujourd'hui supportées par des solutions informatiques, aussi bien au niveau des processus internes que des interactions extérieures. Au niveau social, l'avènement d'Internet, la multiplication des ordinateurs fixes et mobiles, et l'émergence d'acteurs non-traditionnels tels que Google, Yahoo, ou Facebook ont profondément modifié nos habitudes de vie.

Tout cela a néanmoins une contrepartie : les usagers demandent toujours plus de services de la part des logiciels, tandis que la pression sur les prix ne faiblit pas et les délais de mises sur le marché se réduisent sans cesse. De plus, les services fournis par les logiciels doivent être accessibles de partout, avec un haut niveau de performance. Les logiciels deviennent ainsi de plus en plus :

- nombreux et de taille importante afin de fournir un nombre croissant de services,
- interdépendants car les nouveaux développements doivent interopérer avec des applications patrimoniales, développées au fil du temps à l'aide de technologies variées,
- hétérogènes et parfois hétéroclites car construits à partir de composants réutilisés ou achetés,
- distribués suite à l'utilisation massive des réseaux et à l'arrivée de nouveaux paradigmes de mise à disposition des logiciels tels que le « cloud computing » [4] ,

Toutes ces évolutions ont des conséquences directes sur l'activité de maintenance. Examinons en premier lieu l'interdépendance des logiciels. Avec le temps, les entreprises ont en effet constitué des topographies logicielles complexes, composées de nombreuses applications réparties, fondées sur des technologies différentes et interdépendantes. Ces applications doivent ainsi communiquer pour fournir les services attendus par l'entreprise. Ces structures ne cessent de grandir : la refonte d'un système d'information est très couteuse et implique souvent des durées d'indisponibilité rédhibitoires. La variabilité entre les technologies utilisées au sein d'un système d'information est ainsi très grande : méthode de conception, langage de programmation, concepts manipulés, outils d'administration varient au fil du temps. Ainsi, les logiciels fournissant des services complémentaires proposent fréquemment des méthodes de configuration et un vocabulaire associé radicalement différents. Un grand nombre d'applications utilisent des fichiers et des formats de configuration qui leur sont propres. Certaines applications se configurent via XML, d'autres en ligne de commande ou via une interface Web. Qui plus est, les logiciels actuels permettent la configuration d'un grand nombre de paramètres, souvent plusieurs centaines [5].

Voyons maintenant l'impact de l'hétérogénéité grandissante des logiciels. Le temps où les logiciels étaient développés en totalité pour un projet est révolu. Dorénavant, pour des raisons de coût et de délai toujours plus serrés, développer un logiciel revient plutôt à assembler divers constituants existants, appelés composants ou COTS (pour *Components Off The Shelves*). Ces derniers peuvent être des exécutables fournissant des capacités fonctionnelles ou d'administration, ou encore des sources de données. Il va de soi que les différents composant d'un système utilisent des technologies diverses, et ont des objectifs et des rythmes d'évolution différents. Ainsi, pour assurer la maintenance de tels assemblages, il faut être capable de maîtriser diverses technologies, comme expliqué précédemment, mais aussi de s'adapter à des évolutions de composants que l'on ne contrôle pas. L'activité de maintenance devient, dans de telles conditions, un véritable défi technique et humain

Examinons enfin les aspects liés à la prolifération des réseaux. Les logiciels sont de plus en plus distribués. Les composants constituant un logiciel se répartissent fréquement sur des réseaux de différentes natures : locaux ou à grande échelle, filaires ou radio, bas débit ou haut débit, fiables ou pas, etc. Ainsi, maintenir une application inclut désormais la maintenance des réseaux qu'elle utilise. Cela demande, une fois de plus, la maîtrise de nouvelles compétences technologiques pour configurer ces réseaux, par exemple. Cela exige également d'intégrer les évolutions d'un constituant que l'on ne contrôle pas : le réseau dans un bâtiment par exemple évolue au gré de son propre calendrier, et non pas au rythme des besoins des applications qui l'utilisent.

La prolifération des réseaux, et notamment d'Internet, conduit également au développement de nouveaux modes de distribution des logiciels. De nombreuses applications sont aujourd'hui accessibles à distance. C'est, par exemple, le cas de suites bureautiques telles que

Microsoft Office ou *OpenOffice*. Ces nouvelles applications constituent le cœur de métier d'entreprises récentes comme Google. Cependant, l'externalisation des applications et du stockage de données apporte de nouvelles exigences extrêmement fortes sur le logiciel, et notamment sur la phase de maintenance. Les consommateurs de services, surtout lorsqu'il s'agit d'entreprises, demandent des taux de disponibilité de 99.999% bien difficiles à atteindre. Il faut comprendre que l'indisponibilité a de lourds impacts financiers. En 2000, le coût horaire d'une indisponibilité se chiffrait déjà en millions de dollars [6]. En 2008, ce coût n'a pas diminué ; ainsi, une interruption de services pour une entreprise comme Amazon se chiffre encore en dizaine de milliers de dollars par minutes[1]. Actuellement, une entreprise aussi importante que Google ne peut pas s'engager à fournir plus de 99.9% de disponibilités pour ces applications[2]–soit plusieurs heures d'interruption de service par an. A long terme, une succession d'anomalies peut détériorer l'image de l'entreprise et lui faire perdre la confiance des clients. Or, le développement du *cloud computing* ne pourra se continuer sans la confiance des consommateurs. Ceux-ci n'accepteront de confier leurs données personnelles et professionnelles qu'à la condition expresse que leur intégrité et leur disponibilité soient garanties. Les remous[3] engendrés par les récentes perturbations de Google mail témoignent de cette angoisse des utilisateurs de perdre leurs données. Pour atteindre une telle qualité en termes de performance et de disponibilité, il faut être capable d'effectuer des opérations de maintenance de façon rapide et sûre ; ce n'est pas le cas aujourd'hui sur des logiciels hétérogènes de grande taille.

Ainsi, les applications logicielles sont devenues hétérogènes, distribuées, interconnectées et vitales, au niveau économique ou sociétal. Elles appartiennent à des ensembles informatiques de plus en plus sophistiqués et évoluent dans des environnements instables, voire imprévisibles. En conséquence, l'activité de maintenance d'un parc de logiciels devient de plus en plus complexe. Les ingénieurs doivent avoir des temps de réaction très rapides pour garantir la disponibilité des logiciels et maîtriser un nombre important de technologies, toujours plus pointues, pour intervenir adéquatement. Par exemple, l'hétérogénéité des outils d'administration a nécessité la spécialisation du personnel administratif et la mise en place de formations spécifiques pour obtenir des certifications : les certifications CISCO ou Oracle pour ne citer qu'elles. De fait les logiciels deviennent de plus en plus difficiles à installer, configurer et maintenir, sans parler de l'optimisation, et les ingénieurs, quel que soit leur niveau de compétence, arrivent aux limites de leurs capacités.

Le nombre de personnes nécessaire au déploiement et au maintien des logiciels s'est ainsi fortement accru ces dernières années. IBM a, par exemple, dû recruter des milliers de personnes par an pour gérer cette complexité [7]. Il est notable que l'aspect humain joue un rôle important dans l'informatique actuelle. Les experts en sécurité ont depuis longtemps souligné l'importance de ce facteur dans la sécurisation des systèmes. Cependant, jusqu'alors les erreurs humaines pour l'administration des systèmes n'étaient que peu prises en compte et l'on considérait que l'administrateur ne commettait pas d'erreur durant le déploiement, les mises-à-jour et plus généralement la maintenance et la réparation des problèmes. Cette hypothèse n'est aujourd'hui plus raisonnable au vu de la taille, la complexité et la distribution des applications. Au début des années 2000 de nombreuses études [8] ont été menées sur les causes d'erreurs et la répartition des coûts dans les systèmes d'information. La figure 1 montre les résultats d'une étude de six mois conduite en 2001 et portant sur trois sites internet de taille moyenne souhaitant rester anonymes. Elle montre très clairement que les erreurs des opérateurs sont majoritaires. Ainsi on estime aujourd'hui que 80% des dépenses d'une entreprise du secteur des technologies de

[1] D'après CNET : http://news.cnet.com/8301-10784_3-9962010-7.html

[2]Google Apps Service Level Agreement : http://www.google.com/apps/intl/en/terms/sla.html

[2]Google Apps Service Level Agreement : http://www.google.com/apps/intl/en/terms/sla.html

[3]"Google Outage Damage Could Credibility": http://www.infoworld.com/d/cloud-computing/google-outages-damage-cloud-credibility-279

l'information sont effectuées pour de la maintenance [9]. Approximativement 40% des erreurs conduisant à des arrêts de fonctionnement sont causées par les administrateurs eux-mêmes [10].

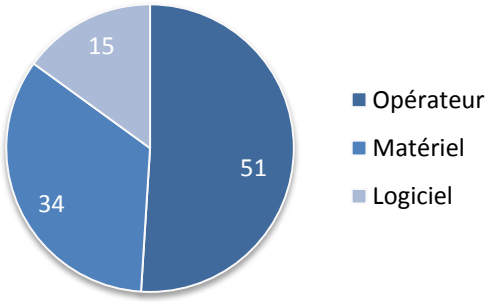

Figure 1 : Pourcentage d'erreurs par cause pour trois sites internet en 2000.

Et les problèmes d'administration ne font que commencer ! En effet, les profonds changements induits par la multiplication des réseaux ne se limitent pas aux entreprises. En 1991, Mark Weiser a dépeint un monde dans lequel l'informatique serait omniprésente et surtout transparente pour l'utilisateur [2].Cette vision qui a donné naissance au domaine de l'informatique ubiquitaire devient progressivement une réalité (figure 2). Ainsi les appareils du quotidien sont de plus en plus connectés avec la démocratisation des technologies d'accès au réseau (notamment sans fil : Wifi, Bluetooth, 3G). Cette capacité d'accéder au réseau en tout lieu a donné naissance au terme marketing ATAWAD (« Any Time, AnyWhere, AnyDevice »). Les GPS, les « smartphones » et l'apparition de nouveaux objets communicants tel le Nabaztag sont autant d'exemples de la multiplication de ces systèmes dans la vie quotidienne. Nous pouvons également prendre le cas plus anecdotique, car encore peu répandu dans la population, des aspirateurs ou des réfrigérateurs vendus comme« intelligents ». Les dispositifs d'interaction avec les systèmes sont polymorphes : geste, voix, mesures biométriques. Cette multi-modalité contribue à l'effacement progressif du système informatique.

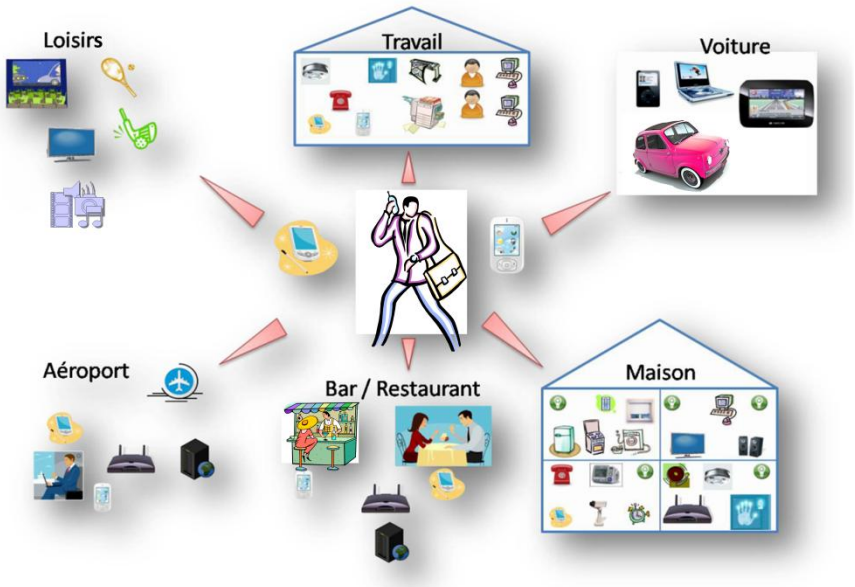

Figure 2 : L'informatique pervasive (tiré de [11])

De plus en plus d'utilisateurs non experts se trouvent ainsi confrontés à l'informatique. Le niveau de connaissance de ces personnes est faible et si l'on peut raisonnablement penser qu'il

augmentera avec le temps, il est peu probable qu'il évolue suffisament pour faire face à la complexité actuelle des systèmes. Mais plus qu'une question de compétence, les utilisateurs ne veulent pas avoir à se soucier des problèmes de configuration. D'autre part, l'environnement dans lequel évoluent les applications pervasives reste extrêmement fluctuant et tributaire, par exemple, du bon fonctionnement des infrastructures réseaux, de la disponibilité d'énergie et des conditions extérieures (niveau sonore, température, ...). Ainsi les systèmes informatiques du quotidien sont non seulement plus complexes à construire et à destination d'un public non-expert, mais doivent également pouvoir fonctionner dans des environnements extrêmement variés et imprévisibles par essence.

C'est IBM qui est à l'origine de l'utilisation du terme « autonomique » pour qualifier un système informatique capable de s'adapter à des évolutions internes et externes avec un minimum d'interventions humaines. Paul Horn, directeur de recherche d'IBM, a détaillé ce concept d'informatique autonomique dans un manifeste rédigé en octobre 2001 [12]. Il identifie la complexité logicielle comme le défi majeur de l'informatique en ce début de siècle. Selon lui, cette complexité constituera à court terme un frein à l'évolution des services et des logiciels. Si rien ne change la complexité des logiciels continuera à augmenter jusqu'au point où il ne sera plus possible pour les ressources humaines de faire face. Les administrateurs devront apporter des solutions à des problèmes dont les causes seront difficiles à diagnostiquer. En conséquence, cela conduira à la mise en péril des performances et de la fiabilité des systèmes, ce qui aura évidemment des conséquences financières. Dans une période de rationalisation des dépenses des entreprises, il est peu probable que les comités de direction avalisent les budgets colossaux nécessaires au maintien des systèmes. Le problème de la complexité concerne aussi la sécurité : comment, dans un monde si ouvert, garantir la sécurité des informations si l'on n'arrive plus à configurer les logiciels qui les manipulent ?

En conséquence, IBM propose que, puisque les capacités d'adaptation de l'homme aux machines qu'il utilise sont limitées, les machines évoluent de façon autonome en fonction des enjeux et de leur environnement. Il est nécessaire d'introduire une certaine marge d'autonomie dans les systèmes informatiques pour absorber la complexité des tâches administratives. Les administrateurs doivent pouvoir déléguer une partie de leur charge de travail au système. Ils pourront ainsi focaliser leur attention sur les fondamentaux du système en se déchargeant des tâches rébarbatives. Les bénéfices attendus de l'informatique autonomique sont donc multiples et incluent notamment :

- **une diminution des coûts de maintenance des ressources de l'entreprise**. L'objectif est en effet l'obtention de systèmes capables de se configurer automatiquement et de tendre vers le zéro-configuration de la part de l'administrateur. En corolaire, l'informatique autonomique permettra une revalorisation des tâches administratives en permettant aux administrateurs de se focaliser sur les manipulations vraiment importantes.
- **une diminution des erreurs dues à des opérateurs humains**.
- **une augmentation de la disponibilité des services fournis en élevant leur fiabilité.** L'anticipation des problèmes potentiels et l'affinement des diagnostics par les systèmes permettront d'augmenter la fiabilité des applications.
- **une réactivité et une sécurité accrues**, qui prépareront mieux les systèmes aux éventuels actes de malveillance.
- **une meilleure répartition et économie des ressources** en réduisant leur utilisation au strict nécessaire.
- **une offre de service plus adaptée, voire personnalisée, à l'utilisateur**. Cela comprend l'adaptation au niveau de l'utilisateur et permettra l'adoption en douceur de l'informatique par le grand public.
- **l'adaptation des services à des environnements fluctuants** dans lesquels évoluent les applications pervasives.

2 PROBLEMATIQUE

A terme, la mise en œuvre de solutions autonomiques robustes et fiables est donc incontournable pour les entreprises voulant rester concurrentielles : la gestion de systèmes complexes constitue un défi majeur de l'informatique moderne, elle ne pourra se résoudre sans un développement conséquent de l'informatique autonomique.

En dépit des travaux effectués dans le domaine, le développement d'applications autonomiques demeure très complexe. Si la complexité des activités administratives est effectivement réduite par la mise en œuvre de l'informatique autonomique, la complexité ne disparaît pas pour autant : elle est transférée au système. Pour cette raison les systèmes autonomiques sont plus complexes à développer que les systèmes traditionnels. Ils font appel à un panel plus large de techniques, en particulier celles d'intelligence artificielle, et nécessitent des compétences importantes de la part des développeurs et des experts. Il n'en demeure pas moins que les systèmes autonomiques seront soumis aux mêmes exigences et aux mêmes délais que les systèmes classiques.

Dans ce contexte, proposer des techniques de conception et des plateformes permettant la conception de systèmes autonomiques est un enjeu de taille. IBM a proposé une architecture de système autonomique généralement acceptée par la communauté. Nous le verrons, la conception d'un système autonomique demande la réalisation d'une ou plusieurs boucles de contrôle implémentées par un ou plusieurs modules spécialisés, les gestionnaires autonomiques, qui prennent en charge l'administration de tout ou partie du système.

Il existe actuellement de nombreux projets assistant les développeurs dans la construction de systèmes autonomiques. Soulignons cependant que ces derniers définissent majoritairement des méthodes spécifiques à un domaine et ne s'appliquent qu'à une classe réduite d'applications. Les plateformes réellement génériques sont plus rares. Elles mettent généralement l'accent sur la définition des relations entre les différents gestionnaires et assez peu sur l'architecture interne de ces derniers.

Lorsqu'ils ne sont pas conçus de façon monolithique, l'architecture interne des gestionnaires autonomiques ressemble à l'architecture de référence proposée par IBM. Or, nous verrons que cette architecture est davantage une architecture logique qu'une architecture d'implantation. Son application systématique et littérale produit des gestionnaires difficiles à maintenir et à étendre. Très souvent, l'architecture d'IBM est implémentée sous forme de règles ou de code – éventuellement de composants – difficiles à maintenir ou à faire évoluer dynamiquement.

Il faut également souligner que les systèmes autonomiques doivent tenir compte de nombreuses préoccupations et d'objectifs parfois divergents si bien qu'il est difficile d'assurer un comportement cohérent au système. Les travaux actuels fournissent peu d'indications sur la façon d'implémenter ces différentes préoccupations et de faire évoluer les priorités entre les différents objectifs. L'implémentation de ces aspects a toutefois des conséquences importantes sur les capacités d'évolution et la maintenabilité des systèmes autonomiques.

Nous considérons que la possibilité de faire évoluer l'architecture interne des gestionnaires autonomiques est trop souvent ignorée par la littérature ou que les solutions apportées pour ce faire sont insuffisantes. L'évolution et le dynamisme sont pourtant des propriétés importantes. D'une part les applications sont de plus en plus dynamiques, hétérogènes et par conséquent imprévisibles. Il n'est souvent pas possible de déterminer à l'avance l'ensemble des comportements que devra adopter le gestionnaire. Pouvoir l'étendre dynamiquement apporte une solution élégante à ce problème. D'autre part les gestionnaires autonomiques sont des applications à part entière qui consomment des ressources et ont besoin d'être administrées. Ils peuvent également connaître des anomalies de fonctionnement dues à des erreurs de

développement. On ne devrait pas avoir à arrêter les gestionnaires pour pouvoir modifier leur comportement ou corriger leurs algorithmes.

Il nous paraît donc important d'introduire de nouveaux éléments architecturaux pour prendre en compte ces problèmes.

3 OBJECTIFS DE CETTE THESE

L'objectif de cette thèse est de fournir un cadre conceptuel et programmatique pour faciliter le développement de gestionnaires autonomiques. Nous visons à définir un modèle architectural modulaire et homogène permettant le développement de gestionnaires autonomiques flexibles, évolutifs, dynamiques et administrables.

Nous l'avons dit, nous considérons que le dynamisme et l'extensibilité sont deux propriétés essentielles pour faire face à l'hétérogénéité et au dynamisme des applications modernes. Elles sont particulièrement nécessaires lorsque l'ensemble des contextes d'utilisation est mal défini ou imprévisible. C'est pourquoi nous voulons faciliter l'évolution et l'extension des stratégies de gestion globale en supportant l'ajout, la mise-à-jour et la suppression de fonctionnalités aux gestionnaires, ainsi que la modification dynamique de leur architecture interne.

Notre approche est modulaire et homogène, donnant ainsi la possibilité d'étendre et de modifier facilement le comportement du gestionnaire, apportant aux experts maîtrise et flexibilité. Il existe actuellement de nombreux projets se focalisant sur un aspect particulier de la gestion autonomique (collecte, analyse, planification, analyse). Par exemple, il existe de nombreux algorithmes d'inférence ou de planification. Notre approche se veut compatible avec ces outils et devrait permettre leur intégration et réutilisation.

Un des principaux objectifs de ce travail est de définir et séparer clairement les concepts appartenant de façon générique au domaine de l'informatique autonomique et les aspects métiers définis pour chaque nouveau système. Nous voulons permettre et favoriser la réutilisation de fonctionnalités d'administration entre plusieurs solutions autonomiques pour diminuer le temps de développement et augmenter la fiabilité. L'expert doit pouvoir se focaliser sur l'écriture des fonctions autonomiques et des activités de gestion, mais ne pas avoir à gérer leur enchainement ni le contrôle.

Pour atteindre nos objectifs, nous proposons d'introduire un nouvel élément architectural d'un niveau d'abstraction et d'une granularité plus faible et complémentaire aux éléments traditionnellement utilisés dans les systèmes autonomiques : les tâches d'administration. Ce sont des unités fortement spécialisés et cohérentes, ce qui facilite leur implémentation. Nous fournissons un cadre architectural permettant l'intégration et l'administration de ces dernières. Dans notre approche, le comportement global d'un gestionnaire résulte de la composition opportuniste de ces tâches d'administration ; les combinaisons entre les tâches évoluent dynamiquement au fil du temps et des besoins pour s'adapter aux contextes d'utilisation variés et aux modifications effectuées sur le système.

4 STRUCTURE DU DOCUMENT

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

- le **chapitre 2** présente l'informatique autonomique, en commençant par une définition du domaine et des systèmes autonomiques. Après l'étude des propriétés des systèmes autonomiques, nous en présentons l'architecture. Nous concluons par la présentation de projets importants.
- le **chapitre 3** introduit l'approche orientée service. Après la définition de la notion de service, nous décrivons les principes de ce modèle d'interaction. Nous montrons ensuite que la mise en œuvre de cette approche est variée en présentant différentes plateformes : Web Services, OSGi, SCA puis iPOJO. Les services jouent un rôle de premier plan dans la mise en œuvre de notre approche.

La contribution se divise en quatre parties :

- le **chapitre 4** expose la problématique, les objectifs et donne une vision d'ensemble de notre approche. Nous expliquons le concept de tâche administrative et le rôle des grands blocs architecturaux de notre approche.
- le **chapitre 5** fournit une description détaillée de l'architecture de notre framework. Ce chapitre discute des techniques permettant l'administration et l'activation opportuniste des tâches, ainsi que la gestion des conflits potentiels. Nous présentons d'abord l'architecture des tâches atomiques. Nous détaillons ensuite l'architecture de contrôle et en particulier les mécanismes de gestion de conflits. Puis nous abordons l'adaptation et la construction du gestionnaire. Pour finir, nous introduisons le concept de composite qui facilite le passage à l'échelle.
- le **chapitre 6** propose une implémentation de cette architecture sous la forme d'un modèle à composants nommé Ceylan. Nous montrons d'abord l'importance des services pour notre implémentation, puis nous détaillons l'implémentation de chaque bloc architectural.
- le **chapitre 7** illustre l'utilisation de notre framework par un cas d'application tiré de la domotique : une application de vidéosurveillance. Nous donnons notamment des exemples d'utilisation des techniques de gestion de conflit que nous avons utilisées. Nous construisons un gestionnaire en implémentant chacune des préoccupations de gestion de façon séparée et nous montrons leur intégration pour obtenir un comportement globalement complexe.

Enfin le **chapitre 8** synthétise les idées principales de notre proposition. Nous rappelons les points principaux de notre contribution et nous décrivons les perspectives envisagées pour de futurs travaux.

Chapitre 2 - **Informatique autonomique**

L'informatique autonomique propose donc de faire reposer la complexité de la maintenance non plus seulement sur l'humain, mais en large partie sur du logiciel. Elle peut se définir, d'une manière générale, comme étant l'approche qui vise à construire des systèmes autonomes autogérés dans l'objectif d'assister l'opérateur humain dans sa configuration, sa maintenance et son utilisation.

Dans ce chapitre, nous définissons précisément la notion d'informatique autonomique. Nous présentons les propriétés attendues d'un système autonomique, puis l'architecture logicielle de tels systèmes et enfin quelques exemple et critères d'évaluation des solutions autonomiques.

1 INFLUENCES

Dans ce qui précède, nous avons vu que la complexité des logiciels et les coûts qu'elle engendre sont les principales motivations du domaine de l'informatique autonomique. Attachons-nous maintenant à en étudier les influences. L'informatique autonomique peut être vue comme une tentative de rapprochement entre différents domaines [13] jusqu'alors parfois assez hermétiques. Tout d'abord, elle s'inscrit dans la lignée des techniques inspirées de la biologie. Des algorithmes biomimétiques et socio-mimétiques sont notamment utilisés dans certains projets. D'autre part, la robotique et la théorie du contrôle sont des domaines dont l'influence se fait sentir depuis l'origine. Enfin, les besoins d'adaptation et d'apprentissage nécessaires au succès de l'approche font naturellement écho au domaine de l'intelligence artificielle.

Si nous utilisons le terme autonomique en lieu et place d'autonome, c'est pour rappeler la première influence de l'informatique autonomique : la biologie. En langue anglaise, le terme « autonomic » se réfère au système nerveux autonome, responsable de la régulation des fonctions vitales. La complexité du corps humain, de par le nombre et la variété de ses constituants, pourrait rendre impossible la coordination de toutes les actions nécessaires au maintien du système dans un état stable. L'homéostasie est pourtant une condition indispensable à la survie de l'organisme. Néanmoins, en dépit de cette complexité, les organismes vivants témoignent de capacités d'adaptation à leur environnement jusqu'à présent inégalées. Ces capacités s'expliquent par l'organisation du contrôle dans l'organisme. Il n'est, en effet, pas possible à un organe central unique de régner sans partage. Deux parties distinctes s'accordent pour garantir l'homéostasie : l'une, le cerveau, commande les actes conscients tandis que l'autre, le système nerveux autonome, pilote les activités inconscientes. Le nom de ce dernier vient donc de ce que ses actions sont indépendantes de la volonté. Il est aussi connu sous le nom de système neurovégétatif. Il régente la régulation des fonctions automatiques de l'organisme comme par exemple le contrôle de la pression artérielle, la respiration, la digestion ou la sudation. Il s'étend du cerveau à la moelle épinière en passant par le tronc cérébral et possède des ramifications vers chaque glande et organe du corps humain. Il se compose de plusieurs sous-systèmes qui s'influencent mutuellement pour assurer le fonctionnement de notre corps sans que nous en ayons conscience. Nous pouvons alors mener à bien des activités intellectuelles sans nous préoccuper des contingences corporelles. Il est important de souligner que le cerveau, et donc les actions conscientes, ont des répercussions sur le système neurovégétatif. Par exemple une pensée angoissante, génératrice de stress, entraine des modifications du rythme cardiaque et une sudation plus élevée. Dans la proposition d'IBM, l'administrateur joue donc le rôle du cerveau. Il n'intervient qu'en cas de défaillance ou lorsqu'il désire changer le comportement du système. Il se contente d'une vision générale et synthétique de son environnement pour prendre des décisions. Une large partie de la charge d'administration est quant à elle gérée par une quantité variable de subordonnés autonomes. Ceux-ci disposent d'une vision très localisée et détaillée de la situation d'une ressource ou d'un pool de ressources particulier. La somme de leurs actions définit le comportement général du système.

Mais les influences de la biologie ne se limitent pas à l'imitation du corps humain. Des milliards d'années d'évolution ont permis à des organismes biologiques au départ très simple d'évoluer vers des organismes sophistiqués disposant de mécanismes complexes destinés à accroître leur résistance et leur capacité d'adaptation. Avec l'augmentation des connaissances en biologie, de nombreux chercheurs se sont naturellement inspirés du fonctionnement de ces organismes pour développer des modèles informatiques [14]. Ce sont les algorithmes dits biomimétiques. De la même manière les organismes biologiques ont structuré leur collaboration autour d'activités sociales complexes qui ont à leur tour inspiré les chercheurs pour l'élaboration d'algorithmes dits socio-mimétiques. Ils forment un ensemble d'algorithmes utilisables dans l'informatique autonomique. Dans [15], les auteurs dressent par exemple une liste de stratégies bio-inspirées utilisées pour le cas particulier de l'autoréparation. Le point commun des systèmes

bio et socio-mimétiques est la forte distribution et la décentralisation du contrôle. Ils mettent en œuvre la collaboration d'un grand nombre d'acteurs très spécialisés ne possédant qu'une connaissance limitée de leur environnement. De la somme des comportements à l'échelle microscopique résulte un comportement plus intelligent à l'échelle macroscopique. On parle alors d'intelligences collectives. C'est particulièrement le cas des colonies et essaims d'insectes sociaux [16], tels les termites ou les fourmis, qui servent régulièrement de métaphores [17]. D'autres sources d'inspiration sont la vie quotidienne et l'économie avec, par exemple, l'utilisation de monnaie virtuelle et la négociation entre différentes parties. Ces modèles s'appliquent souvent à l'allocation de ressources comme dans [18] ou [19]. Le domaine du multi-agent résulte de ces réflexions et il est au cœur de nombreux projets en informatique autonomique (voir [20], [21] ou [22]).

L'informatique autonomique s'appuie sur deux autres domaines : la robotique et l'automatique. Les robots évoluent souvent dans des environnements très dynamiques et les besoins d'adaptations sont importants. L'informatique autonomique a ainsi emprunté plusieurs techniques de la robotique, telles que celles liées à la planification en milieu incertain [23].

L'objectif traditionnel des systèmes automatiques est le maintien sans intervention humaine de paramètres dans un cadre de référence, en général une grandeur physique telle que la vitesse ou la température. Les techniques de régulation ont été développées pour répondre à ces besoins avec notamment l'usage de boucles de rétroaction. Comme illustré par la figure 3, ci-dessus, une boucle de rétroaction se compose de trois éléments : le système que l'on souhaite réguler, un ensemble de sondes et un contrôleur. A partir d'une valeur de référence fixée par l'utilisateur, la consigne, le rôle du contrôleur est d'observer le système via les sondes pour effectuer les modifications garantissant la stabilité du système et le respect de la consigne. Pour cela, il faut posséder une représentation précise des systèmes ; c'est pourquoi les techniques de régulations font appel à une modélisation mathématique de l'environnement qui défini l'état du système et les valeurs d'entrée et de sortie possibles. Ce concept de boucle de rétroaction est présent dans l'informatique autonomique depuis l'origine [24].

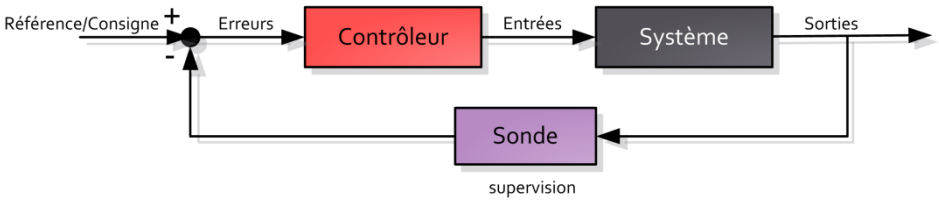

Figure 3 : boucle de rétroaction

Lorsque l'on évoque l'informatique autonomique, le domaine de l'intelligence artificielle est le premier qui vient à l'esprit. A l'origine, certains esprits critiques ont d'ailleurs accusé IBM d'avoir introduit un néologisme pour endormir les réticences des entreprises souvent peu enjointes à mettre en œuvre les techniques issues de l'intelligence artificielle. C'est un procès d'intention : en effet, les champs d'application couverts par l'intelligence artificielle sont beaucoup plus vastes et l'objectif bien plus ambitieux. D'abord, le but initial de l'intelligence artificiel était/est de construire des machines capables de rivaliser intellectuellement avec l'homme [25]. L'objectif de l'informatique autonomique demeure beaucoup plus modeste puisqu'il s'agit « simplement » de donner de l'autonomie aux systèmes informatiques. Cette notion d'autonomie est, en particulier, bien plus importante pour un système autonomique que l' « intelligence » relative du système. D'autre part, le niveau d' « intelligence » se quantifie mal et reste très dépendant du domaine ; la notion d'autonomie s'évalue beaucoup plus facilement. L'intelligence n'est donc pas ici la finalité mais un moyen de donner de l'autonomie au système. Enfin un système autonomique se définit comme étant nécessairement subordonné à un administrateur humain dans le but bien précis de l'assister dans les tâches d'administration, ce qui n'est pas nécessairement le cas d'un système intelligent. Toutefois, les travaux menés dans le

domaine sont évidemment indispensables à la compréhension et la construction de systèmes autonomiques [26].

Comme dans de nombreux domaines, l'action a précédé la qualification. Les besoins d'autonomie, si la situation actuelle les met particulièrement en lumière, ne sont pas nouveaux et divers systèmes possédaient des propriétés autonomiques avant qu'IBM ne propose son approche. A ce titre, les algorithmes utilisés dans les réseaux font figures de précurseurs : la construction des tables de routage ou le protocole SpanningTree en sont de bons représentants. Un grand nombre de tâches administratives ont déjà été automatisées. On peut ainsi citer l'exemple des gestionnaires de packages sous UNIX qui résolvent automatiquement les dépendances, ou encore les utilitaires de mises à jour automatique. L'USB et l'auto-configuration des réseaux (Wifi par exemple) sont des exemples d'adaptation qui ont permis l'adoption de ces outils par le grand public. Enfin, en même temps que l'informatique autonomique, ont été proposés des domaines connexes aux besoins similaires. Citons par exemple l'informatique proactive d'Intel [27] et l'intelligence ambiante [28].

Enfin, il est fondamental de souligner que, si l'informatique autonomique s'inspire et accueille des approches et des techniques issues d'autres domaines informatiques, elle constitue bien un domaine à part entière et possède ses propres « points durs ». En particulier, nous pensons que les problématiques architecturales constituent des défis particulièrement importants. Il s'agit, par exemple, de répondre aux questions du type :

- comment organiser un système autonomique ?
- comment séparer code applicatif et code d'adaptation ?
- comment construire un gestionnaire autonomique ?
- combien faut-il de gestionnaires autonomiques et comment les faire collaborer ?

La suite de ce chapitre tente d'apporter quelques réponses à ces questions fondamentales.

2 DEFINITIONS

2.1 INFORMATIQUE AUTONOMIQUE

De nombreuses formulations ont été proposées pour définir le domaine de l'informatique autonomique dans son ensemble. Nous retenons ici les définitions suivantes qui émanent de chercheurs reconnus dans le domaine :

- *« The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7 » [24],*
- *« The Autonomic Computing Paradigm has been inspired by the human autonomic nervous system. Its overarching goal is to realize computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans » [29] ,*
- *« Autonomic computing is the ability to manage your computing enterprise through hardware and software that automatically and dynamically responds to the requirements of your business » [30],*
- *« The aim [is to] create the self-management of a substantial amount of computing function to relieve users of low level management activities, allowing them to place emphases on the higher level concerns of running their business, their experiments or their entertainment »[31].*
- *« The vision of autonomic computing is to create selfware through self-* properties» [32].*

Nous constatons que ces définitions s'accordent bien, ce qui n'est pas le cas dans tous les domaines de recherche en informatique comme nous le verrons dans le chapitre suivant. Le but de l'informatique autonomique est ainsi clairement de soulager le travail des ingénieurs chargés de la maintenance en élevant le niveau d'abstraction des tâches administratives, notamment en les exprimant sur la base de buts de haut-niveau. Elle vise, en outre, l'intégration des solutions autonomiques dans des solutions métiers.

La cohérence de vue entre les chercheurs nous permet de proposer une définition synthétique, en français, que nous utiliserons dans la suite de ce document :

- *« L'informatique autonomique est le domaine qui s'intéresse à l'ensemble des techniques d'adaptation pour proposer des approches et des outils permettant la construction de systèmes auto-administrés, dits autonomiques »*

Notons, néanmoins que les définitions mettent souvent en avant la notion de propriétés auto-*. Ce raccourci, très fréquemment utilisé en informatique autonomique, est intéressant car il va droit à l'essentiel en se focalisant sur les propriétés que les systèmes autonomiques doivent impérativement posséder. Les propriétés envisagées ici sont, par exemple, l'auto-configuration, l'auto-optimisation ou l'auto-réparation. Utiliser ce raccourci a néanmoins une faiblesse : définir l'informatique autonomique sur la seule base de ces propriétés ne permet pas d'en saisir facilement une vision d'ensemble. Par ailleurs, il faut bien noter que les experts ne s'accordent pas toujours sur l'ensemble exact de propriétés définissant l'autonomie. Nous reviendrons sur ces propriétés auto-* dans la suite de ce chapitre.

Les acteurs du domaine se sont également efforcés de situer l'informatique autonomique par rapport aux travaux antérieurs. En effet, par le passé, de nombreuses applications auto-adaptables ont été construites de manière *ad-hoc* et elles ont souvent répondu avec succès à des besoins spécifiques d'adaptation au contexte d'exécution. L'informatique autonomique vise à répondre à un besoin de généraliser les méthodes utilisées de façon à faciliter la construction de ces applications auto-adaptables :

- *« Autonomic computing is not a new field but rather an amalgamation of selected theories and practices from several existing areas including control theory, adaptive algorithms, software agents, robotics, fault-tolerant computing, distributed and real-time systems, machine learning, human-computer interaction (HCI), artificial intelligence, and many more. »[33].*
- *« What is new is AC's holistic aim of bringing all the relevant areas together to create a change in the industry's direction »[31]*
- *« what is needed is a new computing architecture and programming paradigm that takes a holistic approach to the design and development of computing systems and applications. Autonomic computing provides such an approach by enabling the design and development of systems/applications that can adapt themselves to meet requirements of performance, fault tolerance, reliability, security, etc., without manual intervention.»[29].*

Le but de l'informatique autonomique est ainsi de fournir une approche holistique et les techniques associées pour permettre l'auto- gestion de systèmes informatiques.

2.2 SYSTEME AUTONOMIQUE

Le but ultime de l'informatique autonomique est, bien évidemment, de construire des systèmes autonomiques. De façon surprenante, il n'y a pas toujours accord sur la définition précise d'un système autonomique. Deux caractéristiques sont cependant communément admises. Premièrement les systèmes autonomiques sont guidés par des politiques de haut-niveau fixées par l'administrateur, comme nous le mentionnions précédemment [29] [34] ; il existe cependant plusieurs visions de ce qu'est une politique dans la littérature. Deuxièmement une des

caractéristiques essentielles des systèmes autonomiques est la nature des événements auxquels ils réagissent.

Nous retenons les deux définitions suivantes pour la notion de système autonomique :

- *« The Self-Managing systems' community are coming to an agreement that the term autonomic computing is […] being used to describe […] systems […] in which […] decision changes to reflect the current environmental context; reflecting dynamism in the system »[35],*
- *« Each autonomic element will be responsible for managing its own internal state and behavior and for managing its interactions with an environment » [24].*

Ces deux définitions indiquent, d'une part, qu'un système autonomique est avant tout un système qui s'adapte à son contexte d'environnement et, d'autre part, qu'un système autonomique doit aussi se préoccuper de son état interne.

Sur la base de ces définitions et des différents projets se réclamant autonomiques, nous utiliserons cette définition générique et neutre :

- *« Un système autonomique est un système traditionnel – un composant, un service, une application ou une ressource matérielle – qui possède les informations suffisantes sur son environnement et sur lui-même pour pouvoir anticiper les situations qu'il rencontre et s'y adapter dans l'objectif de délivrer ses services de la meilleure façon possible et de respecter les politiques fixées par l'administrateur».*

Notons, enfin, que dans la vision proposée par IBM [24], un système autonomique est composé d'un ensemble d'éléments autonomiques qui se coordonnent. Ici nous considérerons qu'un élément autonomique n'est qu'un système autonomique particulier. La taille d'un système autonomique (du composant à l'application) varie en fonction des domaines d'application, des besoins et du degré d'autonomie souhaité.

2.3 CONTEXTE

Nous avons introduit dans les définitions précédentes la notion de contexte. Cette dernière, essentielle en informatique autonomique est très fréquemment mentionnée, parfois avec des sens différents. Mais, finalement, qu'entendons-nous par contexte ou plus généralement quelles sont les informations pertinentes décrivant l'environnement ? Une définition minimaliste serait l'ensemble des informations sur lesquelles le système autonomique raisonne pour s'autogérer. Evidemment, cet ensemble varie selon le système considéré et sa finalité. Une base de données dans une entreprise ne sera pas sensible aux mêmes informations qu'un téléphone portable par exemple. La caractérisation du contexte constitue à elle seule un domaine de recherche et il ne s'agit donc pas ici de couvrir l'ensemble des définitions. Les auteurs de [36] proposent une définition générale du contexte :

- *« Le contexte est constitué d'éléments représentant directement l'environnement d'exécution et d'éléments plus complexes dérivés de ces éléments ou issus de leur agrégation ».*

Ils donnent comme exemple les éléments décrivant l'infrastructure et le système (niveau de batterie ou ressource réseau) ou l'utilisateur (position, bruit, besoins). La notion de contexte n'est que rarement explicite dans les documents d'IBM mais on y trouve, ainsi que dans la littérature, des éléments constitutifs de la description de l'environnement ou du système : état des composants ou plus généralement modification de l'architecture, variation de la disponibilité des ressources matérielles (processeur, mémoire) qui concordent avec la définition de [36].

Si l'on cherche une classification plus élaborée du contexte, on peut se tourner vers un domaine proche de l'informatique autonomique : l'informatique pervasive [2]. Un grand nombre de travaux ont, en effet, été menés dans ce domaine sur la notion de contexte [37][38]. Ces définitions sont cependant très centrées sur l'utilisateur, telle celle de Dey [39] :

- *« Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves ».*

Une classification généralement retenue [39][40] propose trois catégories :

- **l'environnement d'exécution** (computing environment) regroupe l'ensemble des variables qui décrivent les technologies informatiques disponibles telles que les ressources matérielles utilisées, les périphériques et ressources accessibles, la capacité du réseau, etc.
- **l'environnement de l'utilisateur** (user environment) regroupe l'ensemble des informations sur l'utilisateur : sa localisation, ses besoins immédiats, des informations sur les personnes environnantes, sa situation sociale, etc.
- **l'environnement physique** (physical environment) correspond à la description du lieu dans lequel évolue l'utilisateur et intègre, notamment, le niveau sonore ou lumineux.

Le contexte est donc essentiellement dépendant de la finalité de l'application. Outre l'usage d'une base de données pour recueillir ces informations sur l'environnement, les concepteurs de systèmes autonomiques ont parfois recours à des ontologies [41][42] permettant de couvrir un plus large spectre de descriptions.

Ainsi, si l'on tient compte de la définition du contexte précédente, il apparaît qu'un système autonomique doit prendre en considération les modifications intervenant au niveau de cinq acteurs :

- **l'administrateur** qui définit les objectifs et les limites que doit respecter le système. Il peut à tout moment choisir de redéfinir les objectifs sur la base de rapports d'activité que le système autonomique lui transmet.
- **les utilisateurs** bienveillants ou malveillants : le système doit pouvoir s'adapter aux conditions dans lequelles évolue l'utilisateur (contexte d'utilisation) et se protéger des attaques.
- **la plateforme d'exécution** c'est-à-dire l'ensemble des contraintes matérielles et logicielles. Le système doit être en mesure s'adapter aux perturbations qui peuvent être causées par d'autres applications, la perte de réseau ou la défaillance d'un équipement.
- **les autres systèmes autonomiques** : lorsqu'il appartient à des infrastructures importantes qui nécessitent par exemple le partage de ressources, un système autonomique peut être amené à négocier avec d'autres systèmes autonomiques.
- **le système** lui-même : le système agit sur lui-même et doit évaluer les modifications qu'il a effectuées pour vérifier qu'elles conduisent à la réalisation des buts de l'administrateur.

Les interactions avec un système autonomique sont classiques en ce qu'elles font appel aux acteurs traditionnels du génie logiciel : utilisateur et administrateur. Cependant le nombre et la nature de ces interactions sont différents, et il faut ajouter l'interaction avec l'environnent et d'autres systèmes autonomiques. Du coté administrateur, le gain espéré avec l'informatique autonomique est de réduire les interactions entre le système et d'en augmenter le niveau d'abstraction. Le système autonomique est donc capable de comprendre des objectifs de plus haut-niveau fixés par l'administrateur. Nous détaillerons plus loin ce qui est entendu par « plus

haut-niveau » mais intuitivement on souhaiterait par exemple pouvoir ordonner de « baisser la consommation de processeur » plutôt que de changer manuellement la méthode de compression dans une base de données. Ainsi l'administrateur n'agit plus sur les interfaces de contrôle mais fixe des objectifs. Du coté utilisateur, les interactions ne changeront pas significativement, mais la valeur du service proposé devrait s'accroitre avec, par exemple, l'augmentation de la disponibilité et l'adaptation au besoin immédiat.

3 PROPRIETES D'UN SYSTEME AUTONOMIQUE

3.1 CAPACITES AUTONOMIQUES ET QUALITES

Dans son manifeste initial [12] [24], IBM a délimité un ensemble de propriétés qu'un système autonomique doit posséder pour être qualifié d'autogéré (*self-managed*). Selon IBM, l'autogestion se décline sous forme d'un groupe de quatre capacités autonomiques.

Il s'agit, plus précisément, des capacités suivantes :

- **l'auto-configuration** (self-configuring) est la capacité du système à se configurer automatiquement et de façon autonome. Le système doit pouvoir adapter sa configuration à l'environnement, à partir d'objectifs de haut-niveau : « baisser la consommation de processeur », « augmenter la précision ». Admettons qu'il manque au système un composant : il faut que le système soit capable de le trouver et de l'ajouter, puis le composant doit être configuré. Si le composant est autonomique il lui faut adapter sa configuration à l'environnement, s'il ne l'est pas le système doit le faire pour lui.
- **l'auto-optimisation** (self-optimizing) est la capacité du système à utiliser de façon optimale les ressources dont il dispose. Ce peut être, par exemple, l'emploi d' un algorithme plus précis mais plus gourmand lorsque les ressources sont sous-utilisées. Les optimisations concernent également l'allocation et le partage des ressources entre les différents composants. La machine virtuelle java avec son ramasse-miettes est déjà un exemple de système qui s'auto-optimise.
- **l'auto-protection** (self-protecting) est la capacité d'un système autonomique à se protéger de son environnement. Pour se protéger, un système doit anticiper, détecter et identifier les situations, notamment les situations à risque. Celles-ci peuvent provenir d'une maladresse d'un utilisateur, lorsqu'il n'a pas encore la maîtrise complète du système par exemple. Elles peuvent également être liées à des pannes de matériel. Le système peut alors prévenir une panne de disque dur ou de l'anticiper par des sauvegardes régulières. Bien sûr, les dangers potentiels peuvent être des actes volontaires de malveillance. En particulier un système autonomique doit prendre les mesures de sécurité nécessaires pour parer des attaques par des dénis de services ou des infections par virus. Ce comportement est essentiel pour garantir la sécurité des informations sensibles du système.
- **l'auto-réparation** (self-healing) est la capacité du système à découvrir, diagnostiquer et réagir à une panne. Elle sert notamment lorsque le système n'a pas été capable de s'auto-protéger. C'est le domaine de la tolérance aux fautes. Il n'est pas possible d'anticiper toutes les erreurs qui sont provoque parfois une mauvaise utilisation du système, il faut donc pouvoir réparer. C'est un enjeu important pour augmenter la disponibilité du système. En effet, diviser le temps moyen passé à réparer (Mean Time To Repair) par deux équivaut à diviser le temps entre deux pannes par deux (Mean Time BetweenFailure).

Ces propriétés sont parfois désignées sous le terme CHOP (Configure, Heal, Optimize and Protect) [43] ou auto-*.

Comme le rappelle [2][35], ces propriétés s'inspirent de celles définies pour les agents identifiées dans [44]:

- **l'autonomie** : la capacité des agents à agir sans intervention humaine. Elle est donc très proche de l'autonomie souhaitée par l'informatique autonomique.
- **la sociabilité** se définit comme la capacité des agents à communiquer avec d'autres agents ou des humains (via un langage de communication).
- **la réactivité** : un agent doit répondre rapidement au changement de son environnement, c'est-à-dire être capable d'observer ces changements.
- **la pro-activité** : les agents ne font pas que réagir aux conditions extérieures, ils prennent des initiatives pour mener à bien leurs objectifs.

Les quatre propriétés CHOP sont ici mises en avant car ce sont les plus fréquemment utilisées dans la littérature. Mais on trouve selon les auteurs des variations dans ce groupe de propriétés de base. Ainsi lors de sa présentation, Paul Horn relevait quatre autres caractéristiques rendant l'autogestion possible, progressivement complétées par d'autres chercheurs, l'ensemble formant un groupe de capacités plus ou moins pertinentes et redondantes :

- **l'auto-connaissance** (self-awareness) : le système doit posséder une représentation de son état, de ses comportements, et de celui de ses constituants. Il doit également connaitre ses relations avec les autres systèmes avec lesquels il est en relation. Il faut donc qu'il posséde des mécanismes d'introspection permettant de remonter les informations sur son état et sur ses comportements. Une propriété très proche est l'auto-description, c'est-à-dire la capacité du système à fournir une description de ses capacités et de son état.
- **la connaissance de l'environnement** (environment-aware) : le système doit posséder une description de son environnement.
- **l'auto-apprentissage** (self-learning): le système doit pouvoir découvrir/forger de nouvelles informations sur lui-même ou son environnement.
- **l'auto-supervision** (self-monitoring) et l'auto-ajustement (self-adjusting) : capacité du système à observer et ajuster sa configuration. Ces deux capacités sont au cœur du principe de boucle de contrôle que nous présentons plus loin.
- **la capacité à fonctionner en environnement hétérogène** car les applications sont de plus en plus hétérogènes dans le milieu industriel - conséquence de l'évolution rapide des technologies et de la pluralité des fournisseurs de services.
- **l'anticipation et la pro-activité** : pour permettre la réalisation des propriétés CHOP, il est nécessaire que le système puisse anticiper les états accessibles à partir de l'état courant et des opérations possibles. Cette capacité à anticiper joue un rôle important sur la réactivité du système, une meilleure anticipation rend en effet les systèmes plus réactifs. Elle est en revanche coûteuse en ressources et doit ainsi se faire avec parcimonie.
- **l'anticipation et l'adaptation aux besoins de l'utilisateur**, c'est-à-dire être capable de s'adapter à un environnement humain pour l'humain, ce qui nécessite un modèle de situation sociale.

3.2 TAXONOMIE

Il existe des tentatives de classification de ces propriétés. Il faut d'abord noter que la plupart des propriétés sont simplement dérivées des huit caractéristiques décrites par Horn. Les auteurs de [45] établissent une relation entre les facteurs de qualité d'un logiciel et les propriétés autonomiques. Ils établissent deux groupes de propriétés : les caractéristiques majeures et les caractéristiques mineures.Les premières correspondent aux propriétés CHOP. Les secondes sont un sous-ensemble de celles détaillés ci-dessus (l'auto-connaissance, l'ouverture, l'anticipation, la sensibilité au contexte) ; pour les auteurs, ces caractéristiques mineures sont à la base de la pro-

activité. L'intérêt de cet article est d'établir une relation entre ces huit caractéristiques et les facteurs de qualité (voir Tableau 1).

On retrouve un classement similaire chez Steritt [31] qui organise les propriétés en plusieurs niveaux d'abstraction. Au sommet se trouve la vision : l'autogestion ; en dessous les objectifs : les propriétés CHOP du premier groupe qui donnent du sens à l'autogestion mais sont encore trop générales pour attaquer la conception d'un gestionnaire. Ces objectifs se concrétisent par les capacités suivantes : l'auto-connaissance, la connaissance de l'environnement, l'auto-ajustement et l'auto-supervision qui appartiennent au second groupe. Enfin, à la base, les techniques dont on dispose pour donner des capacités comme l'utilisation d'algorithmes du domaine de l'intelligence artificielle ou la mise en œuvre de techniques de génie logiciel. Les auteurs de [46] proposent une taxonomie de ces propriétés dites autonomiques. Cette classification se destine plus particulièrement aux systèmes autonomiques distribués faisant intervenir un grand nombre d'éléments autonomiques qui collaborent. Son intérêt et son défaut sont de fournir un ensemble de critères permettant de classer les propriétés en fonction d'un système en particulier. Ainsi la classification d'une propriété, par exemple l'auto-protection, varie d'un système à l'autre.

	Fiabilité	Rendement	Maintenabilité	Utilisabilité	Capacité fonctionnelle	Portabilité
Auto-configuration			X	X	X	X
Auto-réparation	X		X			
Auto-optimisation		X	X		X	
Auto-protection	X				X	
Auto-connaissance					X	
Ouverture						X
Sensibilité au contexte					X	
Anticipation		X	X			

Tableau 1 : Relation entre huit caractéristiques autonomiques importantes et les facteurs de qualité [45]

Différents critères entrent en jeu pour la classification d'une propriété, par exemple et de façon non exhaustive :

- **le fait qu'une collaboration soit nécessaire pour obtenir la propriété** : une propriété est macroscopique si elle est obtenue via la coordination de nombreux sous-systèmes autonomiques. Ainsi l'auto-configuration est généralement macroscopique car elle nécessite souvent la coordination de plusieurs composants d'un système. Une propriété est dite microscopique si elle est obtenue sans coordination globale. Pour garantir une propriété microscopique il n'est pas nécessaire que les composants collaborent. Notamment, l'auto-protection peut être microscopique si l'auto-protection d'un seul composant, par exemple le seul point d'entrée du système exposé, suffit à garantir la protection du système dans son ensemble.
- **la durée pendant laquelle une propriété s'applique** : une propriété est continue si elle doit être garantie pendant toute la durée du fonctionnement du système. Par opposition, une propriété est ponctuelle si elle est n'est nécessaire qu'un bref instant.
- **le fait qu'une propriété dépende de l'histoire du système** : Une propriété peut ou non dépendre de l'évolution du système dans le temps et des états antérieurs.

Il est important de noter que si elles aident à mieux comprendre les objectifs d'un système autogéré, ces propriétés ne sont cependant pas indépendantes. Par exemple l'auto-réparation peut

être vue comme une sous-catégorie de l'auto-optimisation (minimisation des problèmes pouvant survenir). Par ailleurs, protéger un système c'est aussi réparer les failles de sécurité. Les limites sont floues. Ici encore les implications inter-domaines de l'informatique autonomique apparaissent. La mise en œuvre de ces propriétés n'implique donc pas toujours une séparation au niveau du code [32]. Concevoir un système possédant toutes ces qualités est une affaire de compromis entre performance, fiabilité et sécurité. Sur les quatre propriétés initialement proposées, deux permettent d'augmenter les performances : l'auto-configuration et l'auto-optimisation ; deux mettent en avant la sécurité et la fiabilité : l'auto-protection et l'auto-réparation. D'autre part les systèmes sont dynamiques, les besoins varient dans le temps et les priorités changent : être performant peut signifier prendre des risques. Il faut donc pouvoir hiérarchiser ces propriétés en fonction du contexte et avoir une vision d'ensemble de l'action autonomique ; les taxonomies sont utiles en cela car elles aident à mieux cerner les besoins d'une application donnée. L'informatique autonomique ne cherche pas à mettre en avant certaines qualités plus que d'autres mais plutôt à trouver un équilibre.

4 ARCHITECTURE DES SYSTEMES AUTONOMIQUES

4.1 PRINCIPAUX ELEMENTS

Vue générale

IBM a proposé un découpage [47] en plusieurs entités généralement accepté par la communauté, en identifiant des blocs architecturaux avec des rôles spécifiques. Dans cette proposition, un système autonomique est constitué d'un ou plusieurs sous-systèmes. Chacun d'eux se voit alors attribuer un certain degré d'autonomie et met en œuvre une boucle de contrôle. Un système ou sous-système autonomique est ainsi constitué de trois parties (figure 4) :

- **un ou plusieurs éléments administrés**.
- **un gestionnaire autonomique** qui établit le raisonnement autonomique. C'est l'entité qui héberge la logique d'adaptation et qui met en œuvre l'essentiel de la boucle de contrôle nécessaire à la gestion des éléments administrés.
- **les points de contrôle** pour permettre la collecte de données et fournir des moyens d'action sur les ressources.

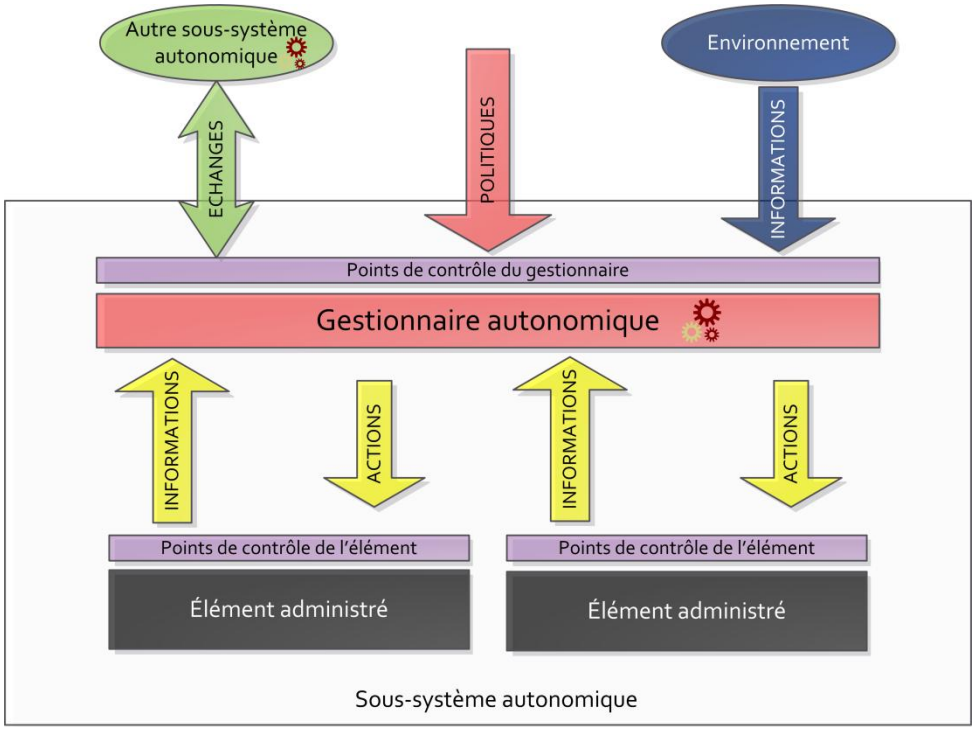

Figure 4 : Architecture générale d'un gestionnaire autonomique

Cette division n'est pas nouvelle et se retrouve dans d'autres domaines inspirant l'informatique autonomique (robotique, théorie du contrôle, …). Aujourd'hui elle est utilisée par la plupart des approches dans le domaine. C'est sur cette architecture générale que nous basons la suite de cette section. Nous décrirons notamment les architectures utilisées pour le gestionnaire et la réalisation points de contrôle.

Eléments administrés

Un élément administré, ou ressource supervisée, représente ce qui est contrôlable par le gestionnaire autonomique. La nature d'un tel élément est très variable : il peut s'agir d'une

ressource physique (par exemple un disque dur ou une interface réseau), d'un composant logiciel ou d'une application patrimoniale. En résumé, un élément administré représente tout ce qui doit être configuré par un opérateur humain en l'absence d'un gestionnaire autonomique. Si l'on se réfère aux objectifs premiers de l'informatique autonomique (faciliter et optimiser la gestion), le fonctionnement de cette ressource ne devrait pas dépendre du gestionnaire autonomique – sauf lorsque l'application est intrinsèquement autonomique. D'une manière générale, un élément administré doit pouvoir fonctionner sans gestionnaire autonomique et être configuré de façon classique. La raison est pragmatique : dans la pratique, il est très difficile de prévoir toutes les conditions d'utilisation d'un élément et il est probable que le gestionnaire autonomique se trouve dans l'incapacité d'apporter une solution adéquate à un problème radicalement nouveau. Il est même envisageable qu'il tombe en panne ou se trompe, il faut donc pouvoir le désactiver dans ces situations.

Gestionnaires

IBM divise les systèmes en deux parties. La première réalise les services traditionnels du système, elle est constituée des éléments gérés. La seconde s'occupe spécifiquement de l'administration et de l'adaptation du système : ce sont les gestionnaires autonomiques.

Historiquement, de nombreuses applications ont été conçues de manière ad-hoc pour répondre à un besoin spécifique sans établir de séparation entre la logique d'adaptation et le code applicatif. Cette confusion vient essentiellement de ce que l'adaptation n'est pas prise en compte comme étant une préoccupation à part entière. La logique d'adaptation se trouve alors mêlée au code applicatif, avec toutes les conséquences négatives que cela implique sur la flexibilité, la maintenabilité et la réutilisabilité.

Depuis l'annonce d'IBM, et à cause de ces nombreux défauts, les concepteurs d'applications ont tendance à vouloir clairement séparer la logique applicative de la logique d'adaptation. Aujourd'hui, la majorité des projets menés dans le domaine prônent une séparation. L'adaptation est complètement découplée et généralement réalisée par un ou plusieurs gestionnaires autonomiques. Grâce à ce cloisenement, les technologies utilisées pour la construction des gestionnaires et des applications peuvent être hétérogènes.

Actuellement, la plupart des systèmes autonomiques sont composés d'un ou plusieurs gestionnaires autonomiques qui administrent le (les) élément(s) administré(s) en s'appuyant sur des points de contrôle et les interfaces d'administration. Le gestionnaire autonomique est la partie du système autonomique qui prend les décisions et qui, pour ce faire, met en œuvre des algorithmes de prise de décision et d'analyse souvent sous la forme d'une boucle de contrôle comme nous le verrons plus loin.

Nous verrons notamment que l'architecture interne d'un gestionnaire est variable en fonction des besoins et de la complexité de l'adaptation : parfois un algorithme dans un bloc monolithique suffit, souvent une architecture plus modulaire est requise.

Il faut également souligner que l'autonomie peut provenir de l'action combinée, voire concertée, de plusieurs gestionnaires dans le cas d'approches décentralisées (très utilisées pour les solutions émergentes) ou de l'action d'un gestionnaire centralisé qui régit alors tout le système.

Politiques

L'un des points les plus délicats lors de la mise en œuvre de solutions autonomiques est la définition de politiques. Ces dernières sont définies par l'administrateur ou d'autres gestionnaires autonomiques et guident le gestionnaire autonomique dans le choix de ses actions face aux évolutions de son Environnement. Un problème récurrent de l'informatique autonomique est la transformation d'objectifs dits de haut-niveau – i.e. au niveau d'abstraction de l'administrateur – en objectifs dits de bas-niveau –i.e. interprétables par un système informatique.

L'étude des techniques permettant de définir des politiques et la façon dont les systèmes les interprètent représente un domaine à part entière et dépasse le cadre de cette thèse. Nous retiendrons ici en particulier les travaux de Kephart et Walsh [34], qui ont permis de dégager trois façons d'exprimer une politique :

- **les politiques d'action** qui à partir d'une situation donnée déterminent les actions à effecteur. En d'autres termes, elles définissent comment revenir dans un état souhaitable lorsque le système se trouve dans une situation non désirable prédéterminée. Les politiques d'action sont généralement exprimées sous forme de règles ECA (Evénement, Condition, Action), qui s'écrivent sous la forme : «Si tels *Evénements* se produisent et ces *Conditions* sont remplies, alors effectuer telles Actions » ; par exemple : « si CPU<10% alors supprimer un serveur ». Cette simplicité les rend très populaires, elles sont utilisées depuis longtemps dans les systèmes adaptables, voir par exemple [48][49] ou [50].
- **les politiques de but** qui caractérisent l'ensemble des états considérés comme valides. Elles se focalisent donc sur la définition des états désirables et c'est au système de déterminer quelles sont les décisions à prendre pour y arriver. Les buts peuvent se décrire en intension ou en extension. Lorsque les politiques d'action sont inadaptées du fait du trop grand nombre d'états possibles, les politiques de but offrent une alternative souvent mise en avant dans l'informatique autonomique [24]. Les politiques de but s'appliquent parfaitement aux systèmes qui raisonnent sur l'architecture du système (notamment Rainbow[51]).
- **les fonctions d'objectif** [52] qui permettent d'estimer le degré de satisfaisabilité d'une situation donnée. Comme les politiques de but, elles permettent de déterminer les états acceptables (ici en intension), mais également de comparer deux situations. Le principal intérêt des fonctions d'objectif est de nuancer les politiques de but. Les états ne sont plus considérés comme valides ou invalides, mais reçoivent une note indiquant le degré de satisfaction. A partir d'un ensemble de paramètres caractérisant un état, la fonction d'objectif est capable de calculer cet indice. Elle généralise la notion d'objectif à atteindre en hiérarchisant les buts, elle supprime ainsi les ambigüités qui peuvent se présenter lorsque deux états différents sont souhaités à partir du même état. Ce type de fonction est par exemple utilisé dans le projet Unity [53] pour calculer le nombre de serveurs à attribuer à chaque consommateur en fonction du taux d'utilisation.

Les travaux sur les politiques s'inspirent fortement des techniques développées dans le domaine de l'intelligence artificielle [54]. Citons par exemple les travaux visant à l'apprentissage automatique des règles et des politiques [55], [56] ou [57].

Quel que soit le type de politique utilisée, les développeurs sont face à l'alternative : intégrer directement les politiques au code, ou les séparer de ce dernier, avec par exemple un moteur de règles ou un langage spécifique. L'intégration directe au code est fréquente car elle ne nécessite pas l'apprentissage d'une nouvelle technologie et que le pouvoir d'expressivité est maximal. Jade [58] est un exemple de plateforme qui ne contraint pas le développeur. D'autres plateformes proposent un langage spécifique comme Rainbow[51], ou utilisent des règles définies séparément (Autonomia[59], Automate).

Points de contrôles

Un élément administré est modifié via des points de contrôle comme le montre la figure 4. Ils permettent les échanges entre l'élément administré et le gestionnaire autonomique. Deux types d'opérations doivent être effectués sur un élément : l'obtention de données et la modification du comportement. On distingue par conséquent deux types de points de contrôle : les capteurs, aussi appelés sondes, et les effecteurs.

Ces points de contrôle sont, de par leur nature, très souvent spécifiques à la ressource à laquelle ils sont rattachés. Ils permettent d'augmenter le niveau d'abstraction du gestionnaire autonomique, contribuant ainsi à supprimer la dépendance entre le gestionnaire et l'implémentation de la ressource. La nature des points de contrôle conditionne celle des interactions possibles sur les ressources supervisées.

Les sondes ou capteurs permettent de collecter des données caractérisant un élément à administrer. Ces données peuvent être simples ou complexes. Par exemple, pour des raisons d'efficacité ou pour des contingences réseau, il est souvent nécessaire de remonter des données agrégées. Ces valeurs peuvent être des moyennes (moyenne d'une température sur une période donnée) ou des informations de qualité de service par exemple

La nature des données collectées par les gestionnaires autonomiques dépend fortement du domaine. L'une des premières étapes lors de la création d'un gestionnaire est la détermination des données qui seront utilisées pour construire le raisonnement. Nous l'avons évoqué en introduction, le gestionnaire autonomique peut être influencé par un grand nombre de variables en provenance par exemple de l'environnement de l'utilisateur ou de l'environnement d'exécution. Ces données sont souvent quantifiables : la quantité de mémoire disponible, la quantité de processeur utilisé, la charge actuelle du réseau. Elles peuvent également concerner des facteurs plus difficiles à évaluer : l'humeur de l'utilisateur, par exemple.

Le mode de distribution des informations au gestionnaire autonomique n'est pas non plus figé. Cela se fait souvent à l'initiative du gestionnaire en utilisant les interfaces fournies par le point de contrôle. Dans certains cas, il est cependant préférable de laisser l'initiative au capteur – via un mécanisme de publication/souscription par exemple – pour économiser les ressources réseau en n'envoyant que les données jugées pertinentes. Enfin, le point de contrôle peut diffuser les informations sur un bus ou un MOM, ce qui permet le découplage du gestionnaire de la ressource et la transmission de l'information à plusieurs gestionnaires.

La standardisation des échanges [60] entre le gestionnaire et les sondes est importante car elle simplifie le développement et la maintenance des systèmes autonomiques. Elle permet, par exemple, d'échanger le gestionnaire autonomique avec celui d'une entreprise concurrente. Elle permet aussi de faciliter la communication avec d'autres gestionnaires autonomiques.

Un exemple intéressant de données utilisées est l'usage de la description de l'architecture comme le montre la figure 5. L'utilisation de l'architecture comme base de l'adaptation dynamique est un thème de recherche important [61]. Il existe aujourd'hui de nombreux travaux autour de ce sujet dont Rainbow [51] ou Jade [62] par exemple.

Ici, le système collecte des données sur la topologie et l'état de l'architecture. En général le raisonnement fait intervenir plusieurs modèles. Pour simplifier, on trouve généralement la distinction entre deux modèles : la spécification/description écrite à la conception, et le modèle de l'application en train de s'exécuter. En effet, l'une des caractéristiques fondamentales des applications dynamiques (celles à base de services par exemple) est que l'assemblage peut intervenir à l'exécution. Les liaisons ne sont donc pas statiques et peuvent évoluer entre les constituants du système. La spécification écrite à la conception permet de contraindre les différentes liaisons possibles et définit les relations potentielles entre les constituants de l'application. Le modèle de l'architecture à l'exécution est la représentation de l'architecture réelle du système à un instant donné du système. C'est généralement un graphe dans lequel les nœuds sont les constituants (au sens très large : un équipement, un composant...) du système et les arcs définissent les relations entre ces constituants. Ces arcs peuvent êtres valués pour qualifier la liaison : la vitesse d'un lien par exemple. Le gestionnaire utilise ces deux modèles : le modèle contenant la description lui est fournie au démarrage de l'application, le second est maintenu par le moniteur lorsque des changements interviennent sur le système. Les effecteurs permettent d'agir sur les points clefs des systèmes. Le gestionnaire peut par exemple : casser des liaisons, changer un constituant, modifier des propriétés. L'usage de composants ou services

pouvant être changés à l'exécution est un avantage pour ce genre d'approche, ce qui explique qu'ils soient généralement utilisés.

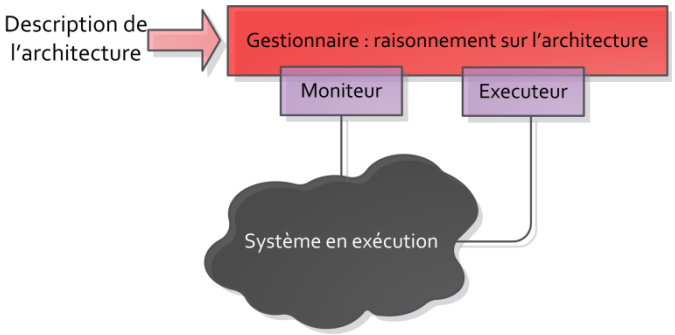

Figure 5 : Raisonnement sur l'architecture

Les effecteurs jouent le rôle complémentaire des capteurs en permettant d'agir sur le comportement de la ressource. Comme les capteurs, ils ont la capacité d'agir sur les interfaces d'administration mises à disposition par la ressource. Ils peuvent être amenés à interpréter des demandes du gestionnaire autonomique, autorisant ainsi à ce dernier l'expression de demandes de plus haut-niveau.

Les gestionnaires autonomiques peuvent reconfigurer un système de plusieurs façons différentes. Les possibilités sont souvent conditionnées par les limites de la plateforme d'exécution. Il faut que l'application soit conçue de manière à supporter l'ensemble des adaptations requises. Il existe trois grands types de reconfiguration possibles. On peut d'abord agir sur la topologie de l'application comme nous l'avons mentionné plus haut. Pour les applications à base de composants et lorsque le modèle à composants le permet, il est alors possible de créer, modifier ou supprimer des liaisons. Un composant peut alors être remplacé par un composant plus performant dynamiquement sans arrêter l'application. Une solution beaucoup plus fréquente consiste à modifier la configuration des composants : les propriétés qui peuvent varier selon le type d'application : par exemple la langue utilisée par l'application, l'encodage choisi pour enregistrer des fichiers, ou la puissance du rétro-éclairage d'un écran. Enfin il est possible de jouer sur le comportement global de l'application. Il s'agit ici de modifier entre autre un algorithme, ou tout autre constituant du système, qui influe sur le comportement de l'application dans son environnement.

La réalisation de l'ensemble de ces points de contrôle peut s'avérer fastidieuse à long terme. C'est pourquoi, actuellement, de plus en plus de plateformes d'exécution fournissent des interfaces d'administration permettant d'agir sur les propriétés ou la topologie des applications. Il est alors facile de construire les points de contrôle sur la base de ces interfaces.

4.2 ARCHITECTURE LOGIQUE D'UN GESTIONNAIRE AUTONOMIQUE

Un système autonomique est un système traditionnel auquel on adjoint une partie permettant d'adapter son comportement aux modifications de l'environnement ou de l'état interne en fonction d'objectifs de haut-niveau fixés par l'administrateur. Implicitement ou explicitement, construire un tel système c'est mettre en œuvre une ou plusieurs boucles de rétroaction qui vont collecter les informations et agir sur le système et son environnement pour respecter les objectifs fixés par l'administrateur. Une boucle de contrôle, telle que représentée figure 6, décrit les étapes nécessaires pour adapter un système. Une telle boucle comprend au moins trois étapes : la collecte d'informations, la prise de décision et l'action. Sur la base des informations collectées dans l'environnement et sur son état interne, le système détermine les actions nécessaires pour respecter un ensemble d'objectifs et agit en conséquence. La mise en œuvre d'une telle boucle n'impose pas d'architecture d'implantation particulière. Dans l'absolu, cette boucle peut être

implémentée d'un seul tenant sous la forme d'un algorithme par exemple, ou via plusieurs blocs modulaires.

Historiquement, ces boucles de contrôle sont implicites : de nombreux systèmes adaptatifs créés avant l'informatique autonomique ne séparent pas la logique applicative du reste du code. Le développement d'applications autonomiques n'est pas nouveau et de nombreux systèmes autonomiques ont été développés de manière ad hoc avant l'annonce de 2001. Le développement de ces applications et leur test constituent un processus fastidieux. Ceci sans compter que les algorithmes utilisés restent souvent difficilement abordables par la moyenne des développeurs. Aujourd'hui encore, de nombreuses applications sont construites de cette façon mais, le thème de recherche se développant, on trouve de plus en plus de préconisations méthodologiques et architecturales. Les boucles autonomiques deviennent de plus en plus explicites.

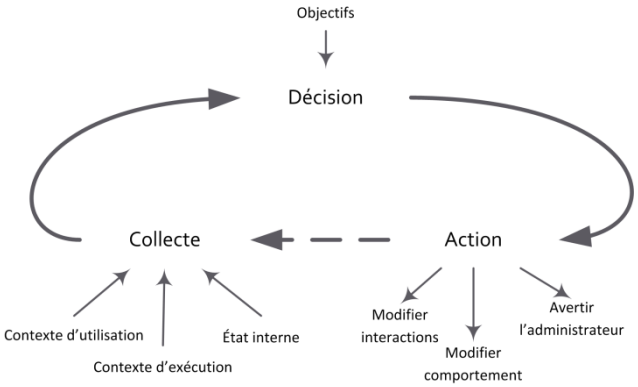

Figure 6 : Boucle de contrôle

IBM a proposé une architecture logique pour la mise en place de cette boucle de contrôle [24][47], en identifiant quatre activités distinctes effectuées par une boucle de contrôle : la collecte des informations, leur analyse, la recherche d'une solution et enfin la mise en œuvre de cette solution. Le fonctionnement est médical : observation, diagnostic, solution, traitement. Dans ce modèle un gestionnaire autonomique est constitué de cinq parties (figure 7) :

- **une partie supervision** est chargée de la collecte des informations en provenance des capteurs. Elle agrège et filtre ces informations. Le composant de supervision produit pour cela un rapport d'activités contenant le résumé des activités, soit à l'expiration dune période de temps déterminé, soit lorsqu'il détecte des symptômes nécessitant une analyse. L'observation des données sur une fenêtre temporelle est nécessaire car la plupart des données ne prennent de sens que sur un intervalle de temps suffisamment long. Le choix d'une période de durée adéquate reste un problème particulièrement ardu dans certaines situations. La partie « analyse » récupère cet ensemble de symptômes.
- **une partie analyse** détermine si des changements doivent être effectués à partir des données produites par la partie supervision. Sur la base des symptômes observés, elle diagnostique les problèmes. Elle possède pour cela un ensemble d'algorithmes pour établir des corrélations, anticiper les situations futures, diagnostiquer les problèmes. Son raisonnement se base sur les derniers rapports reçus, mais également sur des situations antérieures qui lui permettent de déterminer des tendances. En conclusion, elle établit une liste de problèmes à résoudre.
- **une partie planification** résout les problèmes identifiés. C'est la proposition d'un traitement, la construction d'un plan d'action. Ce dernier est une suite d'opérations qui permet, à partir d'un état initial, d'atteindre un état objectif. Cette notion de planification et de plan font directement écho au domaine de

l'intelligence artificielle. Toutefois, l'étape de planification n'implique pas nécessairement l'utilisation de planificateurs de type STRIPS [63] ou Graphplan [64], le plan peut être produit au moyen de règles ECA par exemple.

- **une partie exécution** applique le traitement. Elle agit sur les effecteurs pour réaliser le plan fixé par le planificateur. Elle permet de faire le lien entre le gestionnaire autonomique et les points de contrôle de l'entité administrée.
- **une base de connaissances** permet les échanges entre les différentes parties. Cette base est continuellement mise à jour pour permettre le suivi de l'évolution du système. Elle évite la réplication des données entre les différentes parties et permet de stocker, par exemple, les valeurs mesurées ou un historique de ces dernières, ainsi que des informations sur les politiques et les objectifs de haut niveau.

Cette architecture est connue sous le nom de MAPE-K (*Monitor, Analyze, Plan, Execute, Knowledge*). Le grand intérêt de ce découpage réside dans l'augmentation de la maintenabilité et de la réutilisabilité. Un autre atout de la modularité est la capacité à pouvoir marier différentes technologies de façon transparente. La standardisation des interfaces d'échange entre les blocs telle qu'elle est préconisé par IBM doit permettre d'utiliser des outils différents pour chaque activité. Ces outils peuvent être implémentés séparément par des groupes ou entreprises différentes qui peuvent ainsi se focaliser sur un aspect particulier. Une autre possibilité offerte est celle d'exécuter les différents blocs sur des plateformes d'exécution distinctes. Cependant il faut alors veiller à ce que le temps de circulation des informations, la latence du réseau, ne nuise pas à la réactivité du système.

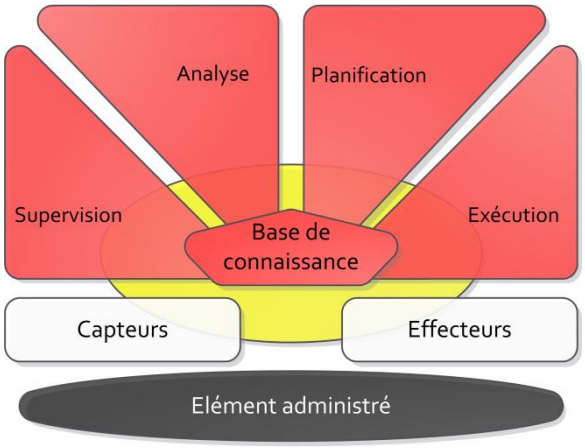

Figure 7 : Un gestionnaire autonomique selon le modèle d'IBM

La production et l'exécution d'un plan peuvent se révéler complexes, il est alors possible que l'action proposée ne corresponde plus au besoin. Prendre une décision demande du temps et le contexte évolue en parallèle. C'est un problème bien connu des systèmes autonomes qu'il faut prendre en compte sans quoi le système peut s'emballer ou boucler. Ainsi, l'un des inconvénients du découpage en bloc, en comparaison avec une approche monolithique, peut être le coût en performance induit par la communication entre les blocs, ce qui risque de diminuer la réactivité du système. Il reste toutefois minime lorsque les blocs sont technologiquement homogènes et qu'ils fonctionnent sur la même plateforme d'exécution. Un autre reproche à l'encontre de cette architecture est le découpage proposé par IBM, parfois trop contraignant. Il s'avère par exemple tout à fait raisonnable de fusionner l'analyse et la planification pour des gestionnaires de taille moyenne.

Le découpage proposé par IBM est donc un modèle à adapter en fonction des besoins. Son mérite est de convenir à une large palette de situations car il est très générique. Le temps et le

coût de développement sont importants à court terme, mais la solution s'avère avantageuse sur le long terme.

Une autre approche architecturale consiste à créer un gestionnaire monolithique. C'était, et c'est probablement encore actuellement, la méthode traditionnellement utilisée par les développeurs. Ici le développeur ne sépare pas les différentes activités d'administration telles que la collecte, l'agrégation de données ou l'identification du problème. Cela peut s'expliquer par plusieurs raisons. Premièrement, lorsque les activités d'adaptation sont relativement simples, il est beaucoup rapide de développer le gestionnaire d'un seul bloc. Corolairement, les optimisations et la factorisation du code sont potentiellement plus importantes. C'est le principal avantage de cette méthode. Ensuite, l'identification des différentes activités n'est pas naturelle en soi. Les concepteurs n'identifient pas toujours ces activités comme étant indépendantes. La rapidité de développement mise à part, cette méthode est désavantageuse sur le long terme. La maintenance du code et les tests sont difficiles. Les possibilités de faire évoluer la logique d'adaptation restent limitées ; en particulier si elle doit être modifiée à l'exécution. La réutilisation du code d'adaptation s'avère plus délicate, voire inexistante. Ainsi, si l'usage de cette méthode se justifie pour la conception de gestionnaires simples dont le code est limité, elle doit être proscrite la plupart du temps.

Lors de la mise en œuvre de l'architecture MAPE-K, les liens entre les composants sont généralement décrits de façon statique à la conception, de sorte qu'il n'est pas possible de les faire évoluer à l'exécution. Certains auteurs proposent de faire évoluer ces liaisons durant l'exécution. Par exemple, dans [65], les auteurs soulignent que la logique d'adaptation peut être réalisée par la composition potentiellement dynamique de plusieurs gestionnaires incomplets comme le montre la figure 8.

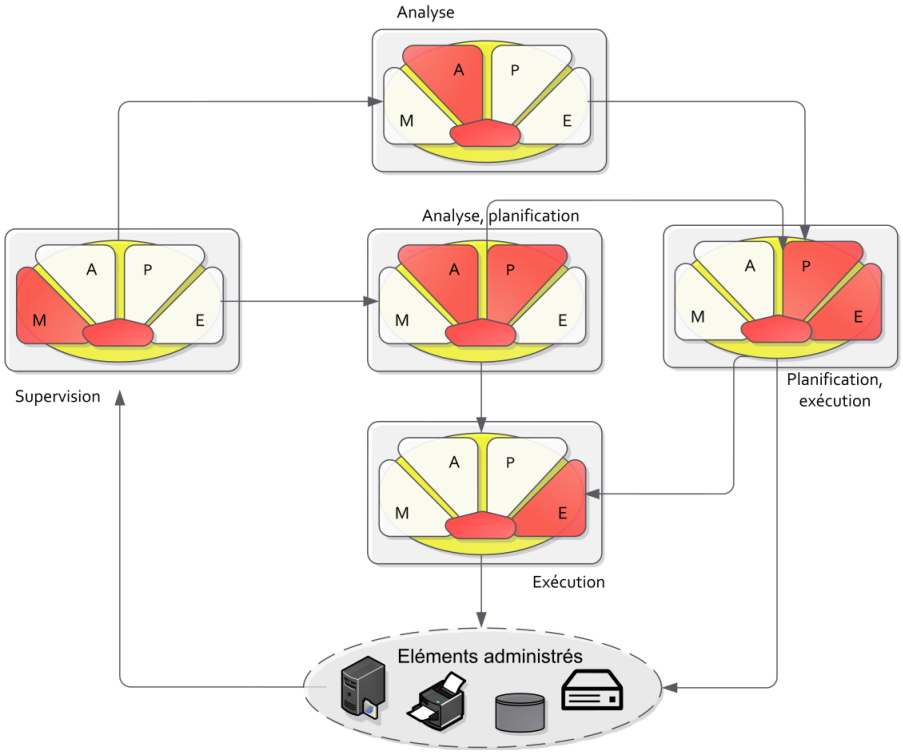

Figure 8 : Composition de gestionnaire autonomique partiel adapté de [65]

Cette solution est donc une variante élégante de la solution précédente qui profite du découpage pour apporter une plus grande flexibilité à l'exécution. Cette approche rend la

standardisation des interfaces obligatoire, sans quoi elle devient rapidement irréaliste. L'un des avantages de cette approche est la possibilité de changer une partie de la logique d'adaptation – par exemple pour introduire de nouveaux algorithmes – durant l'exécution. Cette solution est donc particulièrement adaptée aux environnements ouverts dont les évolutions sont par nature peu prévisibles. Cependant la mise en œuvre de ces solutions présente des difficultés. Le modèle développé dans [65] ne donne que les principes généraux – c'est un patron de conception, insuffisant en soit. Par exemple, il faut déterminer comment les informations sont partagées.

Dans ce type d'approche, le choix de la technologie pivot permettant les échanges entre les gestionnaires autonomiques est important. S'il est tentant d'utiliser les Web Services car c'est une technologie bien maitrisée par les entreprises aujourd'hui, le coût induit par cette solution peut s'avérer trop important pour garantir une réactivité raisonnable du gestionnaire. C'est ce que propose [66] pour l'implémentation des blocs MAPE. Nous verrons lors de la présentation des services qu'il existe des techniques susceptibles de garantir des performances élevées, au détriment toutefois de l'interopérabilité.

4.3 CHOIX D'IMPLANTATION

Nombre de gestionnaires impliqués dans la gestion du système

Lors de l'implémentation d'un système autonomique, l'un des premiers choix est de déterminer le nombre de gestionnaires autonomiques impliqués dans la gestion d'un système. De ce qui précède, nous retenons donc que : sauf besoin particulier, la plupart des systèmes autonomiques sont conçus en séparant clairement la logique d'adaptation du système lui-même. Mais cela n'implique pas qu'il existe un seul bloc de contrôle du système : son autonomie peut provenir de l'action combinée, voire concertée, de plusieurs boucles de contrôle.

Le choix d'un type d'architecture de gestion d'un système ou d'une application autonomique est un des aspects qui a le plus d'impact sur les propriétés des systèmes et la manière de les maintenir. Une architecture de gestion définit le nombre de boucles autonomiques, autrement dit le nombre de gestionnaires autonomiques mis en œuvre dans la réalisation d'un système autonome. Il existe de nombreuses variantes d'architectures qui dérivent de trois grandes catégories.

Les architectures centralisées emploient un seul gestionnaire pour l'ensemble du système considéré. Ce type d'approche est notamment utilisé par les projets JADE [62] et Rainbow [51]. Pour mener à bien l'administration du système, il faut que chaque gestionnaire possède une vision globale, précise et actualisée du système dont il est responsable. Pour ce faire, le gestionnaire doit être en mesure de collecter et comprendre les informations en provenance de tous les points de contrôle présents sur le système. Cela requiert aussi un diagnostic de l'état du système, et la détermination de solutions qui s'appliqueront en plusieurs points du système. L'avantage principal est la cohérence des décisions prises par le contrôleur. Cependant, cet avantage présente un coût : en ressources, en souplesse, en complexité, en fiabilité. Si la prise de décision est théoriquement plus rapide, rapatrier les données et effectuer les analyses est coûteux en ressources. En dehors de la réactivité, qui n'est pas toujours un point critique, la conception de tels gestionnaires s'avère difficile car le concepteur doit être capable de modéliser, ou a minima de prendre en compte tous les états dans lequel le système peut se trouver.

Les architectures distribuées font intervenir plusieurs gestionnaires autonomiques qui se concertent pour faire émerger un comportement globalement cohérent. L'architecture distribuée s'inspire fortement des travaux menés dans le domaine du multi-agents. Depuis la naissance de l'informatique autonomique, de nombreux chercheurs [24] s'intéressent aux systèmes multi-agents (voir [67]). Chaque gestionnaire est responsable de son état et doit accomplir un ensemble d'objectifs précis qui dirigent ses actions. Des politiques expriment ces objectifs de haut-niveau. Pour réaliser les buts fixés, les gestionnaires sont capables de

communiquer avec leurs semblables. Cette concertation permet de faire émerger un comportement global cohérent bénéfique pour le système. L'une des qualités essentielles de cette architecture est qu'elle passe très facilement à l'échelle. L'effort de gestion se répartit entre les différents gestionnaires. De plus, la logique d'adaptation est proche du code qu'elle maintient : cela permet d'augmenter la réactivité du système, car cette proximité limite les coûts de communication. Cette approche, bien que très séduisante et ambitieuse, pose des problèmes. Il est en effet difficile de définir et de séparer clairement les objectifs de chaque gestionnaire pour faire émerger ce comportement cohérent. L'expression des politiques s'avère parfois très délicate et des problèmes de conflits se posent lorsque deux gestionnaires ont des objectifs opposés. D'autre part, les systèmes émergents sont par nature extrêmement imprévisibles. Ainsi est-il difficile de garantir que le comportement du système tendra vers la réalisation des objectifs souhaités.

Les architectures hiérarchiques constituent une variante des architectures distribuées, dans laquelle un rapport hiérarchique s'établit entre les gestionnaires autonomiques. C'est une organisation militaire : au sommet se trouve le gestionnaire principal, qui possède une vision globale des objectifs à atteindre. Il délègue une partie de ses responsabilités à des subalternes en fixant des sous-objectifs sur la base des objectifs généraux. De cette manière, les objectifs deviennent de plus en plus concrets, et correspondent en bout de chaine à des buts très précis. Les décisions sont plus cohérentes que pour les architectures distribuées mais le nombre de communication augmente nécessairement. La mise en œuvre de cette solution a un coût élevé en temps de développement , à court et moyen terme, mais elle constitue la seule alternative viable aux solutions purement distribuées. Cette approche est notamment mise en œuvre dans Auto-Home[11].

Technologies utilisées

Les autres choix sont liés aux technologies utilisées pour l'implantation de la boucle de contrôle et de la remontée des informations.

Souvent les systèmes autonomiques utilisent des règles. L'utilisation de règles et de moteurs de règles est fréquente dans le domaine de l'intelligence artificielle ou du multi-agent. La forme de règle la plus connue est ECA : événement, condition, action. Les règles fonctionnent sur un modèle de programmation événementiel qui est particulièrement adapté aux systèmes proactifs comme le sont les gestionnaires autonomiques. L'avantage principal de l'usage de règles est la flexibilité apportée au système. En effet la plupart des moteurs de règles actuels supportent la modification du jeu de règles à l'exécution, ce qui permet de redéfinir facilement le comportement général du gestionnaire. Il est très facile d'ajouter des règles à un ensemble préétabli, ce qui permet d'adapter le gestionnaire à des situations nouvelles qui n'étaient pas envisagées à sa conception. Il y a toutefois des difficultés inhérentes à l'usage de règles. Lorsque le nombre de règles explose, il devient difficile de prévoir les effets de l'insertion d'une règle. Cette grande souplesse se paie donc par la difficulté à anticiper le comportement d'un système autonomique sur le long terme. Cette complexité est directement corrélée avec le nombre de règles présentes dans les systèmes. Ainsi comme pour le système monolithique à base de code, l'usage de règles ne devrait s'envisager que pour des systèmes de petite taille. L'usage de règles pour la conception de gestionnaires autonomiques entiers est possible, comme le démontre le projet Automate [68] par exemple, mais il est alors nécessaire de découper le gestionnaire en plusieurs entités - chacune régie par un ensemble réduit de règles cohérentes.

L'utilisation de composants et de modèles à composants se popularise ([69] ou [70]), souvent en complémentarité avec l'usage de règles. Les plateformes à composants offrent la possibilité de développer séparément chaque partie de la boucle autonomique. Le gestionnaire autonomiques est souvent implémenté sous forme d'un composant. Lorsque l'architecture n'est pas monolithique, les composants sont utilisés pour développer les blocs MAPE-K des boucles autonomiques. L'avantage de cette approche est le contrôle précis de l'assemblage. Cependant ce

dernier s'effectue au plus tard au déploiement et n'évolue pas au cours de l'exécution. JADE [62] ou Automate [68] sont des exemples pertinents de projets utilisant les composants.

Les plateformes à services et les services sont parfois utilisés. Comme nous le verrons, ces dernières permettent de spécifier les assemblages en termes de fonctionnalités souhaitées. Elles offrent ainsi une plus grande flexibilité au développeur et permettent l'introduction du dynamisme à l'exécution. Ce sont fréquemment la gestion de l'hétérogénéité et la volonté de standardiser les relations entre les constituants des systèmes autonomiques qui poussent à utiliser les services [60]. Le dynamisme à l'exécution proposé par certaines plateformes à services est en général ignoré. De fait les Web Services sont généralement préférés, même si des projets utilisent OSGi [11].

Les approches distribuées et émergentes utilisent fréquemment les plateformes à agents. Celles-ci fournissent des protocoles permettant la mise en relation des agents et leur collaboration. Les projets Unity[53] ou Autonomia[59] utilisent par exemple des approches à agents.

Il existe enfin quelques outils permettant la réalisation de boucles autonomiques complètes, tels que : Autonomic Toolkit[71], ABLE[72] ou Kinesthetics eXtreme[73]. La plupart fournissent des outils facilitant le monitoring ou la communication entre les gestionnaires (ou agents) et des implémentations de fonctionnalités telles que la base de connaissance. Ces outils permettent l'implémentation des boucles MAPE-K en facilitant l'implémentation de chacun des blocs. Toutefois, le dynamisme à l'exécution est pauvre ou n'est possible que par l'utilisation de règles, ce qui pose les problèmes que nous avons mentionnés plus haut.

Dans la section suivante nous donnons des exemples de projets utilisant ces technologies.

5 EXEMPLES DE SYSTEMES EXISTANTS

5.1 RAINBOW

Rainbow [51][74] est un projet initié au Carmegie Mellon University en 2004. L'une des particularités de ce projet est l'utilisation de l'architecture de l'application autogérée comme base de raisonnement par le mécanisme d'adaptation. Le projet Rainbow définit un Framework et un langage pour construire des systèmes auto-adaptables. L'intergiciel définit le support d'exécution du gestionnaire autonomique et les mécanismes permettant de collecter l'architecture de l'application, tandis que le langage permet d'exprimer les politiques d'administration avec des concepts liés à l'adaptation. Historiquement, comme pour le projet ROC [75], Rainbow a été proposé pour construire des applications auto-réparables [76][77].

L'intergiciel propose un ensemble de mécanismes réutilisables permettant l'adaptation des systèmes. Ces fonctionnalités, qui peuvent s'appliquer à de larges classes de systèmes, facilitent la conception du gestionnaire en épargnant le développement des aspects souvent redondants et difficiles. Rainbow fournit également un cadre permettant aux développeurs d'adapter leur gestionnaire à une application particulière. Ces points de personnalisation sont illustrés par les chevrons sur la figure 9. L'objectif est de permettre aux concepteurs de se focaliser sur les politiques de gestion plutôt que sur les mécanismes d'adaptation.

Rainbow définit un support d'exécution des gestionnaires autonomiques indépendant du support d'exécution des applications administrées. L'architecture de gestion est un cas typique d'architecture centralisée avec un seul gestionnaire pour le système entier. La figure 9 représente une vue globale de l'architecture de Rainbow. Le système se compose de trois parties :

- **le système instrumenté au moyen de sondes et effecteurs.** Les sondes et l'effecteur s'exécutent sur le même support d'exécution que l'application et collectent les propriétés et les informations sur les liaisons permettant de générer un modèle de l'architecture à l'exécution.
- **une infrastructure de traduction** qui transforme l'ensemble des informations collectées sur le système pour produire une représentation abstraite indépendante du support d'exécution de l'application. Cette infrastructure fournit pour ce faire des ponts qui permettent d'établir les correspondances entre les informations brutes en provenance de la plateforme et les concepts manipulés par le gestionnaire d'adaptation.
- **le gestionnaire d'adaptation** qui est divisé en quatre parties. Au cœur du gestionnaire se trouve le gestionnaire de modèle ; c'est lui qui maintient le modèle de l'architecture en exécution sur la base des informations remontées par les sondes. L'évaluateur examine l'architecture pour assurer qu'elle est cohérente par rapport à un ensemble de contraintes fixées par le concepteur du gestionnaire. Si l'évaluateur estime que l'architecture ne rentre pas dans le cadre déterminé, il demande au gestionnaire d'adaptation de proposer des modifications. Ce dernier détermine la stratégie à adopter pour revenir à un état acceptable. Cette stratégie est exécutée par le composant d'exécution de stratégie. Ce découpage reste relativement fidèle au découpage MAPE-K proposé par IBM. Ici la base de connaissance et les parties de supervision et d'exécution sont mélangées.

La force de Rainbow est de proposer un ensemble de mécanismes permettant d'automatiser la collecte et la modification de l'architecture de l'application. Il faut également souligner que cet intergiciel reste indépendant de l'infrastructure d'exécution grâce aux mécanismes de traduction. Les gestionnaires construits peuvent ainsi s'adapter à un grand nombre de systèmes différents. Cette propriété facilite également la réutilisation des parties de

gestionnaires. L'ensemble des mécanismes proposés par Rainbow facilite donc grandement la conception de gestionnaires dont le raisonnement se base sur l'architecture.

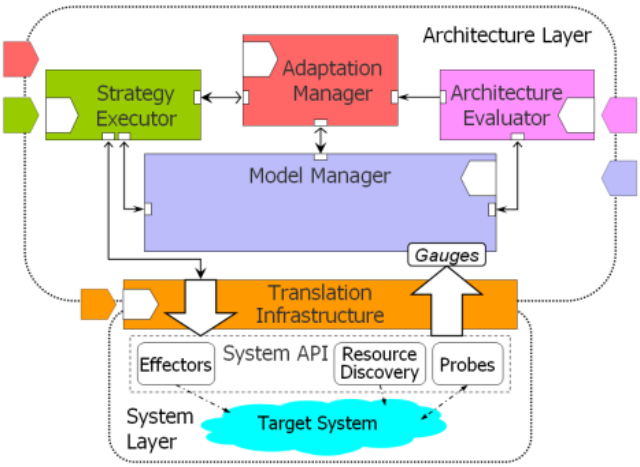

Figure 9 : Architecture du système Rainbow (tiré de [74])

Les limitations de Rainbow sont liées à ses qualités. Comme ils basent leurs raisonnements sur l'architecture, les gestionnaires autonomiques construits avec Rainbow doivent être centralisés. Rainbow possède de fait le défaut des solutions centralisées : point unique de défaillance et problème de passage à l'échelle lorsque les informations collectées deviennent trop importantes. Les auteurs sont conscients de ce problème et ont proposé des solutions pour déléguer une partie des responsabilités du gestionnaire central à des gestionnaires subordonnés [74]. D'autre part, Rainbow propose la construction de gestionnaires indépendants de la plateforme d'exécution, ce qui nécessite d'une part la maitrise d'une nouvelle technologie pour implémenter le gestionnaire et d'autre part l'écriture de transformations pour collecter l'architecture. Enfin Rainbow se focalise sur l'architecture de l'application et les informations collectées passent nécessairement par une traduction pour la représentation d'un modèle abstrait de l'architecture. Il est difficile de baser le raisonnement sur des informations autres que l'architecture, ce qui constitue la principale limitation de Rainbow.

5.2 JADE

JADE [62] est un projet développé à l'Université de Grenoble. Jade est un environnement d'administration autonomique permettant la gestion d'applications autonomiques réparties. Le projet s'est essentiellement focalisé sur la gestion de systèmes complexes et notamment des applications J2EE organisées en grappes et des applications patrimoniales. Ces applications se composent de nombreux éléments ; il est très difficile pour un administrateur d'avoir une vision globale de leur fonctionnement et par conséquent de les administrer. D'autre part, les éléments constituant une application sont très hétérogènes technologiquement. L'un des objectifs de la gestion par des gestionnaires autonomiques JADE est d'augmenter la disponibilité de ces applications en allouant les ressources de façon optimale, et de faciliter le passage à l'échelle en permettant l'administration d'applications de grande taille.

Jade promeut l'utilisation de composants pour la construction de l'application et des fonctionnalités d'autogestion. Jade, illustré figure 10, fournit des mécanismes permettant :

- **l'instrumentation du système administré**. L'application est vue comme la composition d'un ensemble d'éléments administrés logiciels et matériels. L'assemblage de ces éléments et leur configuration évoluent dans le temps. Jade fournit des mécanismes permettant l'encapsulation et l'instrumentation du

système à gérer. L'encapsulation permet d'administrer les éléments gérés à travers des interfaces uniformes. Ces interfaces fournissent des primitives agissant sur le cycle de vie et les liaisons des éléments administrés. Il est ainsi possible d'encapsuler une application patrimoniale.

- **la construction de gestionnaires autonomiques**. Jade divise la tâche d'administration en plusieurs aspects, tels que l'auto-réparation ou l'auto-optimisation, et attribue la responsabilité de chacun de ses aspects à un gestionnaire autonomique. Chaque gestionnaire autonomique est implémenté en respectant la boucle autonomique MAPE-K et respecte des politiques d'administration spécifiques à l'aspect qui l'intéresse.

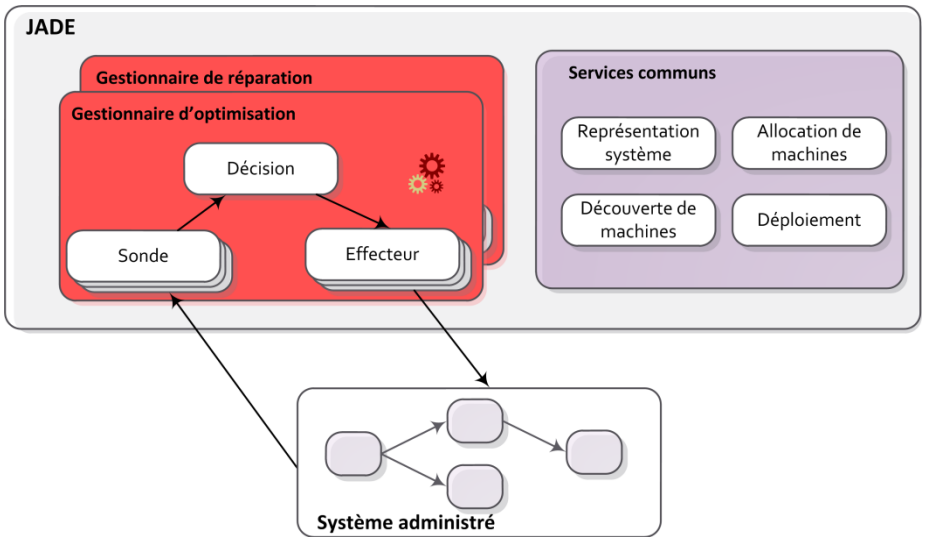

Figure 10 : Vue globale de Jade

Le cadriciel JADE a été implémenté avec le modèle à composants FRACTAL [78]. Ce dernier définit un ADL permettant de décrire l'architecture d'une application à base de composants. Il organise la séparation entre les aspects fonctionnels et non fonctionnels en introduisant la notion de conteneur. De plus, FRACTAL supporte la composition structurelle hiérarchique avec la notion de composite. L'intergiciel supportant ce modèle fournit des mécanismes permettant d'agir sur le cycle de vie des composants durant l'exécution. Il est ainsi possible de modifier les propriétés et les liaisons des composants ainsi que de changer le contenu des composites. Enfin, FRACTAL propose des mécanismes d'introspection, ce qui permet d'observer et modifier les éléments administrés.

Tout comme Rainbow, Jade met l'architecture de l'application gérée au cœur du processus d'autogestion. L'intergiciel fournit une représentation de l'architecture à chaque gestionnaire. Elle est maintenue à jour dans une base de connaissance. Outre cette représentation, JADE fournit un ensemble de mécanismes communs facilitant la création des gestionnaires et permettant de gérer le déploiement d'architecture logicielle, de garder une représentation de l'état du système, de découvrir dynamiquement l'ajout de nouveaux nœuds et de gérer les politiques d'allocation de ressources entre les nœuds. Les politiques de gestion sont codées sous forme d'algorithmes spécifiques au sein de chaque gestionnaire autonomique : c'est le bloc « Décision » de la figure 10. JADE ne fournit donc pas de langage permettant d'exprimer ces politiques dans des concepts métier propres au domaine de l'informatique autonomique.

L'intergiciel a été utilisé pour la conception de gestionnaires autonomiques prenant en charge l'auto-réparation [58]. L'objectif est de réparer l'architecture logicielle lors de la détection d'une panne. L'idée est la mise en place une solution de type heartbeat avec un ensemble de

sondes qui envoient périodiquement des informations sur l'architecture. Le gestionnaire peut alors détecter et remplacer les gestionnaires manquants. Les auteurs ont également proposé un mécanisme pour gérer l'auto-optimisation [79] en allouant automatiquement des ressources en fonction du temps de réponse des composants.

Jade fournit donc un ensemble d'outils permettant la réalisation de gestionnaires autonomiques en respectant le modèle d'IBM. Une des forces de Jade est de permettre l'utilisation d'applications patrimoniales en fournissant les outils nécessaires à leur encapsulation et leur administration. D'autre part, la forte modularisation permet d'augmenter la réutilisabilité et la maintenabilité des gestionnaires, des sondes et des effecteurs. La division de la tâche de gestion en plusieurs gestionnaires est intéressante car elle permet aux développeurs et aux experts de se focaliser sur une tâche particulière de la gestion du gestionnaire. Il faut cependant souligner que ce découpage n'est pas toujours possible. Comme nous l'avons souligné en introduction, les propriétés CHOP ne sont pas orthogonales et il est par conséquent difficile de les implémenter indépendamment sans créer des conflits. Les mécanismes permettant de coordonner l'action de ses gestionnaires restent flous, même si les auteurs tentent de définir les mécanismes mettant en place une telle coordination dans [80].

5.3 UNITY

En tant qu'initiatrice du domaine de l'informatique autonomique, IBM est l'une des entreprises qui développent aujourd'hui le plus de solutions pour la construction de systèmes autonomiques. Depuis l'origine, l'entreprise explore des approches multi-agents [81][53][82] et des outils permettant le développement tels que ABLE[72] puis Autonomic Toolkit [71]. Conformément à la vision décrite par Kephart [24], ces approches font intervenir plusieurs gestionnaires autonomiques qui se coordonnent. Nous présentons ici le projet Unity qui fournit une implémentation de cette vision. Une des particularités intéressantes d'Unity l'utilisation de technologies qui mettent en œuvre les principes de l'approche à services [83] tels que les accords de services et les annuaires.

Le projet Unity [53][82] a été utilisé pour gérer l'allocation de ressources dans un système d'information. Le système est constitué d'un ensemble d'applications qui dépend d'un pool de ressources commun. La charge s'exerçant sur les services fournis par ces applications fluctue en fonction du temps et par conséquent le nombre de ressources nécessaires varie aussi. L'objectif est donc l'allocation du nombre adéquat de serveurs à chaque service pour garantir une bonne qualité de service, et une utilisation optimale des ressources disponibles.

L'architecture d'Unity est illustrée figure 11. Le système se compose de plusieurs acteurs :

- **les ressources qui s'inscrivent** auprès de l'annuaire en fournissant leur localisation et une liste de capacités. Chaque ressource s'engage à respecter les capacités qu'elle décrit. Les serveurs sont des ressources particulières qui s'enregistrent dans l'annuaire.
- **les applications rendues autonomiques grâce à un gestionnaire**. Chaque application décrit une fonction d'objectif qui permet de déterminer son degré de satisfaction par rapport à une solution proposée – ici un nombre de serveurs affectés. L'une des tâches principales du gestionnaire est de déterminer en quoi l'arrivée ou le retrait d'une ressource va affecter l'application et sa capacité à satisfaire ses objectifs.
- **un ou plusieurs arbitres qui sont des entités chargées de répartir les ressources** entre les applications. Un arbitre s'appuie sur le registre pour savoir quels serveurs peuvent être utilisés par l'application. Les arbitres s'enregistrent dans l'annuaire comme des ressources particulières.
- **l'annuaire qui maintient une liste de ressources disponibles.**

- **le dépôt de politique qui contient l'ensemble des politiques s'appliquant au système.** C'est le point unique qui permet à l'utilisateur de fixer des objectifs de haut-niveau au système. Ces politiques sont consultées par les éléments du système pour déterminer leur comportement.

La notion d'arbitre permet ici d'obtenir une répartition intelligente des ressources. L'arbitre est responsable de la gestion du pool de ressources en contrôlant quelle ressource est affectée à quel manager. Il y parvient en demandant à chaque gestionnaire d'application une estimation, exprimée par une fonction d'objectif, de l'impact de l'affectation d'une ressource. Il est alors en mesure de calculer une répartition optimum en maximisant la valeur de chacune des fonctions d'objectif.

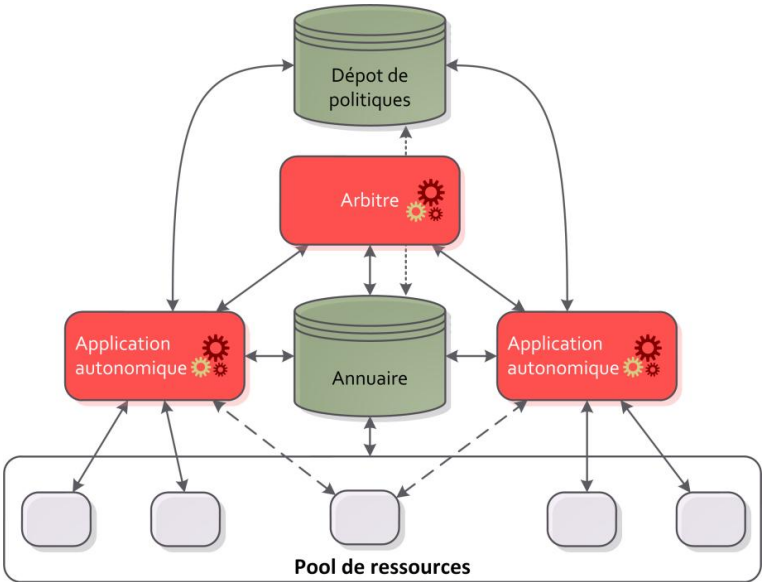

Figure 11 : Architecture du système Unity

L'architecture sépare bien les préoccupations, l'arbitre n'a pas besoin de connaissances particulières sur les applications pour effectuer son calcul. En particulier, il n'est pas concerné par la façon dont chaque gestionnaire d'application produit sa prédiction. De cette façon, il est très facile d'ajouter ou d'enlever des applications du système. Il est également possible de gérer un parc d'applications hétérogène. La souplesse offerte par l'annuaire de service constitue également un avantage de ce système. L'ajout de nouveaux éléments dans le système se fait très simplement en ajoutant une entrée dans l'annuaire. Tout élément du système, y compris l'arbitre et le gestionnaire de politique, est considéré comme une ressource découvrable.

L'intergiciel fournit une interface permettant d'interagir avec le dépôt de politiques. Cette interface et ce dépôt facilitent l'administration en présentant à l'administrateur une vision globale du système et un point unique d'interaction. Le système propose également des outils permettant de superviser le comportement particulier d'une ressource : les sentinelles. Tout élément du système peut s'adresser à une sentinelle pour surveiller un autre élément. En cas de défaillance, la sentinelle avertit son utilisateur, ce qui permet d'augmenter la robustesse du système.

En conclusion, Unity se focalise sur un problème autonomique particulier et y apporte une solution très flexible. L'utilisation d'une approche à services permet de faire évoluer le système très facilement dans le temps. Unity n'est pas une solution complètement décentralisée contrairement à ce qui apparait au premier abord. L'arbitre est au cœur de la répartition des ressources et constitue en cela un point unique de défaillance et d'étranglement. On peut cependant tempérer ces reproches pour deux raisons. D'une part la tâche de l'arbitre reste

complètement indépendante du contenu des applications et il est a priori sans état puisque les informations sont stockées dans un annuaire. De plus, grâce à l'approche à services, il est découvrable et donc fortement découplé des applications. Par conséquent son remplacement par un autre arbitre en cas de défaillance ne devra pas poser de difficultés particulières. D'autre part, les auteurs d'Unity envisagent la mise en œuvre de plusieurs arbitres en fonction du type de ressources à partager, ce qui facilite le passage à l'échelle. Unity présente cependant plusieurs défauts. D'une part la forte spécialisation rend son utilisation difficile pour d'autres catégories de problèmes. D'autre part la détermination d'une fonction d'objectif pour permettre les prédictions de l'arbitre est une tâche qui peut s'avérer ardue.

5.4 AUTONOMIA

Le projet Autonomia [59] [84] a été initié à l'Université d'Arizona. Il essaye de définir une approche holistique permettant la construction de systèmes autonomiques. Le terme holistique fait ici clairement écho aux approches à base de multi-agents dans lesquels la somme de comportements individuels et simples permet de faire émerger un comportement collectif intelligent. L'objectif du projet est de fournir aux développeurs d'applications autonomiques un ensemble d'outils permettant le déploiement d'agents mobiles autogérés.

Le projet se propose de rendre autonomes les éléments logiciels et matériels qui constituent l'application. Il propose d'augmenter les capacités des composants traditionnels (CORBA ou JAVA par exemple) pour fabriquer des agents. Cet intergiciel peut notamment être utilisé pour gérer des applications patrimoniales en ajoutant aux composants des caractéristiques autonomiques. Autonomia fournit l'infrastructure permettant d'organiser le contrôle entre ces différents agents et de les déployer, configurer et superviser. Il est implémenté en utilisant les technologies JAVA et Jini qui facilitent la communication entre les gestionnaires. Le gestionnaire autonomique de chaque agent administré par Autonomia est constitué de deux parties (figure 12) :

- Le CMI (Component Management Interface) contient les politiques de gestion de l'agent.
- Le CRM (Component Runtime Manager) est le composant qui implémente la boucle autonomique en respectant les étapes MAPE.

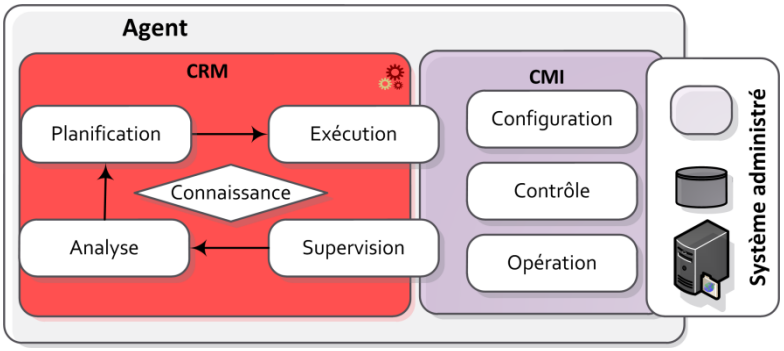

Figure 12 : Un agent dans l'architecture Autonomia

Le CMI est un composant passif qui n'agit pas directement sur la ressource. Il définit un ensemble de ports : le port de configuration (Configuration port), le port de contrôle (Control Ports) et le port opérationnel (Operation Port). Le port de configuration détermine quels sont les attributs nécessaires pour automatiser le déploiement et la configuration de la ressource. Le port de contrôle permet de superviser et d'agir sur la ressource, il détermine quels sont les attributs à observer et l'ensemble des actions possibles sur le composant administré. Les opérations et les attributs proposés par ces deux types de port dépendent fortement de la plateforme d'exécution du système et des mécanismes qu'elle fournit pour modifier les composants et les ressources matérielles. Enfin le port opérationnel permet de définir les politiques de gestion que le CRM doit

appliquer. Les politiques sont divisées en deux ensembles de règles ECA (événement, action, condition) : les politiques de comportement définissent le comportement du gestionnaire et les politiques d'interactions spécifient son comportement avec l'environnement. Ces règles peuvent utiliser les fonctionnalités définies par les deux autres ports.

Le CRM respecte le découpage proposé par IBM. Il est donc constitué de quatre étapes : supervision, analyse, décision, action et d'une base de connaissances. C'est le CRM qui se charge d'observer et de modifier le comportement de la ressource dont il a la responsabilité. Il interprète les règles définies au moyen du port opérationnel du CMI. En cas de changement des politiques au niveau du CMI, le CRM réinterprète les règles et change son comportement pour s'y conformer. Le comportement du gestionnaire est donc décrit dans un langage indépendant du support d'exécution de la ressource, ce qui facilite la réutilisabilité de l'ensemble des règles développées.

En sus de la définition des agents, Autonomia propose un environnement de développement de solutions autonomiques. Il fait intervenir plusieurs modules en proposant notamment un dépôt de composants réutilisables et des outils définissant le comportement général de l'application. Il est notamment possible d'établir des liens de subordination entre les gestionnaires, et donc de créer des architectures hiérarchiques grâce au Compound Component Runtime Manager.

Autonomia offre donc une implémentation conforme à la vision d'IBM avec un ensemble d'éléments autonomiques qui se concertent pour aboutir à une administration globale de l'application. L'architecture est donc distribuée, voire hiérarchique si les développeurs le souhaitent. Le concepteur du système autonomique est libre de choisir les informations sur lesquelles le raisonnement des gestionnaires se base. Il n'est donc plus ici uniquement question de raisonner sur l'architecture, ce qui serait d'ailleurs difficile étant donné la distribution. A ce titre Autonomia laisse beaucoup de liberté aux développeurs. Le choix d'utiliser de règles pour définir les politiques est à double tranchant. S'il apporte beaucoup de flexibilité quand il s'agit de redéfinir le comportement, les règles sont notoirement difficiles à appréhender lorsque leur nombre augmente. D'autant que le pouvoir d'expression limité des règles ECA risque d'entrainer une augmentation de la taille des politiques lorsque des comportements complexes sont souhaités. Enfin comme pour toute solution distribuée, il peut être difficile de garantir la cohérence du comportement du gestionnaire. A ce titre, la spécification ne donne pas d'indication sur la façon dont les échanges, et surtout les liens de subordinations éventuelles, doivent s'opérer pour éviter ce problème.

5.5 AUTOMATE

Automate [68] est un des projets les plus aboutis dans le domaine. Il est conduit à l'université du New Jersey et se compose de nombreux sous-projets (notamment Accord [85], Rudder [86], Meteor [87]) qui définissent un ensemble de technologies permettant la construction de systèmes autonomiques. Nous présenterons ici essentiellement Accord qui est le cœur du projet.

Ce projet a pour ambition de faciliter la gestion des grilles de calculs qui sont des environnements complexes, hétérogènes et dynamiques. Il propose de séparer les préoccupations en établissant une séparation entre les aspects fonctionnels et non fonctionnels, et entre les traitements et les interactions. Il définit la notion d'élément autonomique (figure 13). Dans ce projet, les applications sont construites par la composition dynamique de ces éléments. Il fournit un environnement permettant l'exécution des éléments autonomiques et l'interprétation des règles. Accord fonctionne en effet principalement à base de règles.

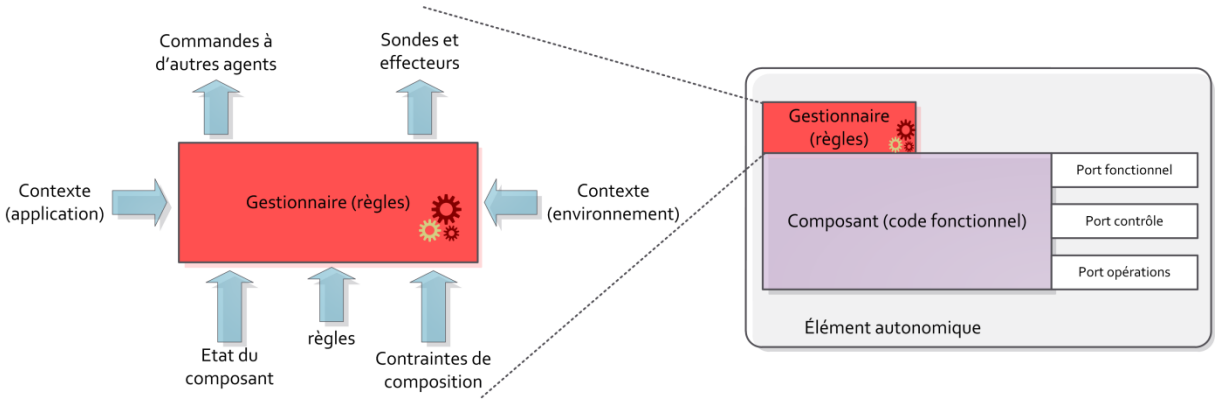

Figure 13 : Un élément autonomique dans AutoMate et son agent [68].

Dans le projet Accord, un élément autonomique est implémenté sous la forme d'un composant. Les constituants traditionnels des systèmes sont étendus par l'attribution des caractéristiques autonomiques. En cela, il ressemble beaucoup aux éléments autonomiques d'IBM. Ces éléments sont alors composés en définissant des politiques d'interaction pour construire le système.

La structure d'un élément autonomique dans Accord comprend les éléments suivants (figure 13) :

- **le composant** qui fournit les fonctionnalités traditionnelles de l'application,
- **un port fonctionnel** qui définit un ensemble de fonctionnalités fournies par le composant,
- **un port contrôle** qui permet la définition de l'ensemble des sondes et effecteurs et les contraintes qui régissent leur accès.
- **un port opération** qui définit les interfaces pour formuler et injecter dynamiquement de nouvelles règles et de les gérer. Ces règles définissent le comportement à l'exécution et les interactions de l'élément.
- **un gestionnaire** responsable du comportement du composant. C'est le gestionnaire autonomique traditionnel de la vision d'IBM ; il fonctionne à base de règles.

La vision d'IBM est ici parfaitement respectée avec la séparation de la logique applicative et de la logique adaptative. Les composants sont régis par un gestionnaire (ou agent) qui surveille son comportement et son état interne par le biais des sondes et effecteurs déclarés (figure 13). Il est sensible aux modifications de son environnement et de l'application ; il peut communiquer avec les autres gestionnaires. Une différence avec la proposition d'IBM est que la boucle MAPE-K n'est pas explicite et que l'architecture gestionnaire est essentiellement monolithique et à base de règles.

Ce gestionnaire est essentiellement un interpréteur de règles. Dans ce projet, les politiques sont en effet exprimées à l'aide de règles de type ECA (événement, condition, action). Elles définissent le comportement et les interactions de l'élément autonomique. L'un des intérêts principaux de l'usage de règles est qu'on peut les retirer et les modifier dynamiquement durant l'exécution de l'application. Le dynamisme est également présent au niveau des constituants de l'application avec la capacité d'ajouter, de retirer ou remplacer des composants.

Pour assurer la cohérence des décisions et donner un comportement stable au système, AutoMate introduit un gestionnaire de composition (figure 14). Il a la tâche de simplifier l'expression des politiques en scindant et transformant des objectifs et des politiques de haut-niveau en buts et règles à réaliser par les éléments autonomiques. Les politiques sont ainsi

définies de façon centralisée, puis analysées et distribuées par le gestionnaire de composition sous forme de règles d'interaction et de contraintes de comportement.

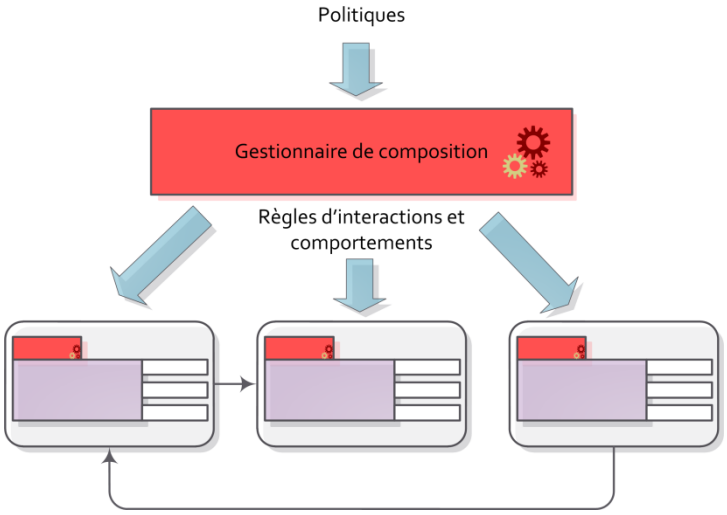

Figure 14 : gestionnaire de composition.

Le projet AutoMate est un excellent exemple de plateforme distribuée qui respecte la vision d'IBM. Il propose une solution intéressante pour la gestion des politiques avec le gestionnaire de composition qui assure la cohérence du comportement des différents gestionnaires. C'est un premier pas vers une architecture hiérarchique. Le fonctionnement général fait penser évoque les solutions multi-agents ; au demeurant le gestionnaire autonomique de chaque élément était nommé agent dans les premieres publications.

L'introduction des règles pour l'expression des politiques et pour le fonctionnement du gestionnaire autonomique est intéressante dans la mesure où elle apporte du dynamisme. Le projet a clairement identifié l'intérêt de faire évoluer le comportement des gestionnaires au cours du temps pour s'adapter au dynamisme des systèmes administrés. Cependant, le pouvoir d'expression des règles ECA apparaît limité et la construction de comportements complexes s'avère parfois extrêmement difficile du fait des conflits qui apparaitront inévitablement entre les différentes règles. Le comportement global du gestionnaire peut être difficile à appréhender pour un administrateur et donc peu aisé à faire évoluer. Les systèmes à règles sont généralement difficiles à maintenir.

5.6 AUTO-HOME

Auto-Home [11] est un projet initié à l'université de Grenoble par l'équipe ADELE. Il propose d'assister le concepteur de solutions autonomiques dans la conception d'une architecture autonomique hiérarchique à services. Il s'agit d'établir une hiérarchie parmi les gestionnaires autonomiques pour permettre de répartir les prises de décision et d'assurer leur cohérence. Ce projet est récent mais un cadre général se dégage.

Au sommet se trouve un gestionnaire principal qui possède une vision globale des objectifs visés. Il délègue une partie de ses responsabilités à des subalternes, en fixant des sous-objectifs sur la base des objectifs généraux. De cette manière, les objectifs deviennent de plus en plus concrets, et correspondent en bout de chaine à des buts très précis. Les feuilles de l'arbre n'ont donc qu'une conscience très limitée des objectifs globaux. Dans le sens inverse, lorsqu'un gestionnaire n'arrive pas à trouver de solution à un problème donné, il peut communiquer le problème à son supérieur. Ainsi les gestionnaires de niveau hiérarchique élevé traitent-ils des informations de nature plus abstraites. Comme les architectures distribuées, cette architecture

passe bien à l'échelle. La réactivité est garantie par les traitements réalisés par les feuilles qui correspondent à de simples traitements réflexes. Ainsi l'organisation interne des différents gestionnaires est potentiellement différente. Par exemple, les gestionnaires se trouvant à la base de la hiérarchie peuvent très bien coder de façon monolithique car la nature de leur tâche est simple. A l'opposé il est possible de développer les gestionnaires au sommet de la hiérarchie selon des modèles plus flexibles et modulaires comme MAPE-K. L'un des avantages de cette approche sur les architectures distribuées est la cohérence accrue du comportement résultant de l'interaction des gestionnaires. C'est le rôle des supérieurs d'apporter une solution aux conflits qui émergent entre les différents sous gestionnaires. Cela s'effectue en amont, en garantissant que le découpage des politiques est cohérent, et en aval en rectifiant le comportement des gestionnaires qui conduisent le système dans une mauvaise direction.

Pour l'instant, la réflexion s'est effectuée dans un contexte centralisé : il n'y a qu'une plateforme. Il existe trois niveaux de gestionnaires (figure 15) :

- **le gestionnaire de la plateforme** dont le rôle est de gérer les ressources de la plateforme et d'arbitrer leur attribution entre les différentes applications. Il doit assurer le bon fonctionnement de la plateforme et des applications essentielles. Il peut ainsi prendre la décision de supprimer une application pour permettre aux autres de fonctionner correctement : une alarme est plus importante que l'affichage d'informations météo par exemple. Il possède une vision globale sur l'ensemble des activités de la plateforme.
- **le gestionnaire d'applications** a pour objectif d'assurer le bon fonctionnement de l'application en veillant sur les services qui la composent. Par exemple, il détecte les erreurs au niveau des services ou choisit les services les plus adaptés. Le gestionnaire d'applications met en œuvre les politiques de l'application ainsi que celles fixées par le gestionnaire de la plateforme. Il doit par exemple être capable de limiter les ressources occupées par l'application si le gestionnaire de la plateforme en fait la demande. Le gestionnaire d'applications possède une vision globale sur l'application, ce qui lui permet de prendre des décisions cohérentes à ce niveau.
- **les gestionnaires de services** gèrent les services associés. Ils sont assez proches de la vision multi-agents présentée précédemment, mais leur capacité de décision est moins grande. Ils ont une perception limitée de leur environnement et doivent appliquer les politiques fixées par le gestionnaire d'application.

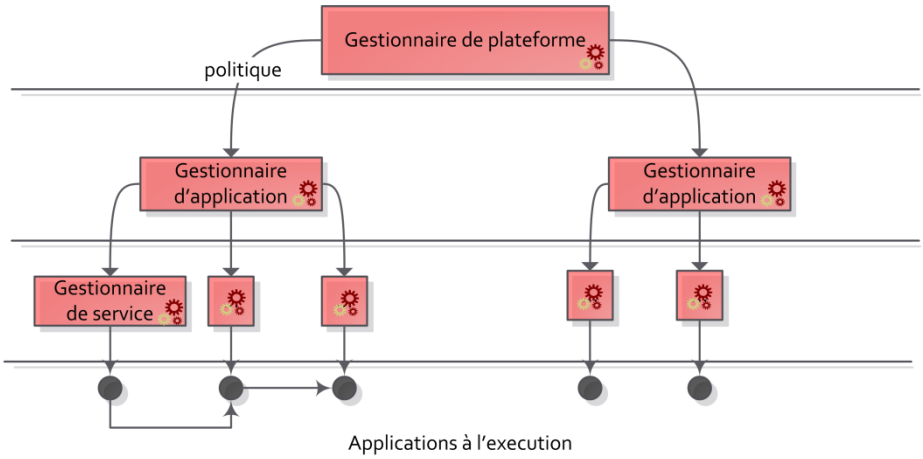

Figure 15 : Exemple d'application hiérarchique

L'objectif n'est donc pas de faire émerger un comportement intelligent par le bas mais d'appliquer un contrôle des décisions par le haut. Chaque gestionnaire tend à essayer d'optimiser

le fonctionnement de sa ressource. Ces objectifs peuvent être opposés, lorsque par exemple deux applications veulent le contrôle des mêmes ressources. La hiérarchisation des gestionnaires permet, en arbitrant, de résoudre ces conflits.

L'un des principaux défauts de cette plateforme est qu'elle spécifie mal la nature des messages échangés entre les gestionnaires. Ces informations sont fréquemment définies de façon ad-hoc. D'autre part, l'architecture des différents gestionnaires n'est pas précisée et apparait comme principalement monolithique, ce qui constitue un handicap important pour la maintenabilité, la réutilisabilité et l'évolutivité des gestionnaires.

6 EVALUATION DES SYSTEMES AUTONOMIQUES

L'adoption large de l'informatique autonomique par les entreprises ne peut se faire de façon brutale. Il n'est pas raisonnable de penser que tous les logiciels et les processus d'administration d'une entreprise vont se voir automatiser du jour au lendemain car les entreprises possèdent des centaines de logiciels sur étagères de provenances et de technologies diverses. La plupart des logiciels utilisés par une entreprise ne sont pas développés par elle et celle-ci est donc tributaire de ses fournisseurs. L'approche la plus raisonnable consiste donc à faire évoluer les systèmes vers plus d'autonomie de manière progressive. Il faut pour cela évaluer le degré d'autonomie des systèmes actuellement en fonctionnement pour pouvoir planifier les améliorations de façon itérative. Conscient de ce besoin d'évaluer le niveau d'autonomie, IBM a dès l'origine établi des modèles d'adoption. L'entreprise estime qu'il existe cinq niveaux permettant de classer le degré de maturité.

	Niveau basique	**Niveau géré**	**Niveau prédictif**	**Niveau adaptatif**	**Niveau autonomique**
Définition	Se base sur de multiples sources de données générées par le système, la documentation des produits pour effectuer manuellement les tâches d'administrations.	Les données sont collectées et agrégées sous une forme plus digeste pour les administrateurs.	Le système collecte les données et établie des corrélations entre les données pour donner des conseils	Le système est capable d'agir de façon autonome lorsque que cela est nécessaire	Les composants du système comprennent les politiques et les processus métiers.
Besoins	Nécessite une équipe d'experts très compétents	Les administrateurs analysent et prennent les décisions	Les administrateurs approuvent les décisions et effectuent les actions	Le personnel gère la performance en terme d'accord de niveau de service (SLA)	Le personnel se concentre sur les besoins métiers.
Bénéfices		Meilleure connaissance du système par les administrateurs. Productivité accrue	Réduit la compétence nécessaire pour administrer le système Augmente la réactivité des administrateurs.	Les applications sont plus flexibles et résilientes. Les interventions humaines sont rares.	Les politiques métiers guident la gestion des systèmes d'informations. Les applications sont agiles et résilientes

Tableau 2 : Degré d'autonomie adapté de [47]

Du niveau basique au niveau autonomique, on passe d'un niveau où les tâches sont complètements manuelles vers un niveau où les composants de l'infrastructure informatique fonctionnent de concert pour adapter dynamiquement le comportement du système. Le **niveau basique** correspond à celui de la plupart des applications traditionnelles selon IBM. Pour maintenir ces applications, une équipe expérimentée d'administrateurs doit analyser les rapports produits par de multiples sources de données. En se basant sur leur expérience, ils prennent toutes les décisions et planifient les actions à entreprendre, y compris les tâches de routines. Le deuxième niveau est le **niveau géré** : à ce niveau les administrateurs disposent d'outils et de script capables de collecter mais surtout d'agréger les données dans des ensembles cohérents. Cet ensemble de données est plus digeste pour les administrateurs et il en résulte une meilleure appréhension du système. Cela ne change toutefois rien au niveau de compétence requis pour

prendre les décisions. Au troisième niveau de maturité, le **niveau prédictif**, les fonctionnalités autonomiques sont capables d'analyser les données pour suggérer des solutions aux administrateurs. Le système est en mesure de détecter des motifs et de prédire des configurations plus adaptées. Les compétences nécessaires pour l'administration sont ainsi moins spécialisées car le système vient en aide sur une partie des compétences spécialisées. Le **niveau adaptatif** est le premier niveau à mettre en œuvre une boucle de contrôle fermée. Le système est ici capable de prendre des initiatives. Il collecte les données, identifie les problèmes, trouve des solutions et met en œuvre les plans d'action. Pour cela, il possède une base de connaissances qu'il complète durant l'exécution. Il respecte des objectifs définis en termes de qualité et de contrat de services. Autrement dit, les compétences nécessaires pour la définition de ces objectifs demeurent élevées. En revanche le système est dorénavant autonome et les interventions humaines restent limitées. Le dernier niveau est le **niveau autonomique** : ici le système est non seulement capable de se gérer de façon autonome mais aussi de comprendre les objectifs et les politiques métiers. Ainsi, les systèmes appartenant à ce dernier niveau permettent à du personnel ayant peu de compétences spécialisées de fixer des objectifs. Le nombre nécessaire d'experts d'un système donné est réduit car les interventions sont limitées.

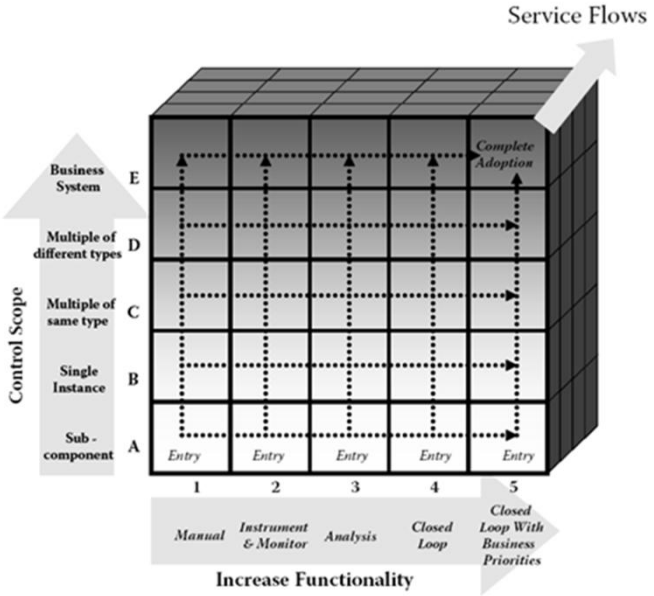

Figure 16 : Modèle d'évaluation de la maturité d'une application par IBM provenant de [47]

En plus du modèle de maturité des solutions autonomiques, un modèle plus complet d'adoption a été proposé (figure 16) par IBM. Il reprend en abscisses les cinq catégories que nous avons détaillées pour le modèle de maturité. L'axe des ordonnées se concentre lui sur la façon dont les fonctionnalités sont réparties au sein de l'entreprise. En bas de l'échelle, seuls les composants sont concernés. Chaque composant peut individuellement être tout à fait autonome sans que le système lui-même le soit. Sans coordination entre ces composants il n'est en effet pas possible d'obtenir un système capable de comprendre des objectifs. Le dernier axe concerne la capacité des systèmes autonomiques à comprendre les processus et les objectifs métiers, et à s'intégrer dans une logique d'entreprise. La construction de systèmes autonomiques ne peut s'effectuer qu'en évoluant sur ces trois tableaux.

En dehors de l'évaluation de leur degré de maturité, il existe d'autres critères permettant d'évaluer une solution autonomique. Les auteurs de [88] proposent en plus les métriques suivantes :

- **la qualité de service** : en quoi la solution autonomique améliore-t-elle le service fourni ? Ce critère est la composition de différents paramètres comme l'augmentation de la disponibilité, le temps de réponse, la pertinence des résultats fournis par les services. C'est un des critères fondamentaux car les solutions autonomiques sont en générales conçues pour augmenter la qualité des services. Ces critères varient naturellement en fonction des applications considérées. Généralement les recherches s'orientent vers l'augmentation des performances ou de la fiabilité ; mais lorsque l'on cherche à améliorer l'expérience des utilisateurs, il peut être acceptable de détériorer le temps de réponse si on améliore la précision des résultats.
- **le coût** : la mise en place d'une solution autonomique permet-elle de réduire les coûts? Pour de nombreuses entreprises, l'adoption des technologies est motivée par la compression des coûts, ce qui comprend les salaires des administrateurs et les coûts de maintenances (y compris les répercutions d'une erreur). L'évaluation est parfois délicate car elle ne peut s'effectuer immédiatement et doit faire l'objet d'une étude sur la durée. Il faut être capable d'estimer la quantité de problèmes que la solution a permis d'éviter, c'est à dire non seulement le coût de l'erreur, mais aussi les répercutions à long terme. Ainsi l'autonomisation d'une infrastructure est un investissement à long terme au coût initial important.
- **la granularité/flexibilité** : c'est l'évaluation de la capacité des systèmes autonomiques à être modifiés. A quel niveau se situe l'intelligence : fortement distribuée dans chaque composant ou centralisée ? Lorsque chaque composant est autonome et mû par des règles qui lui sont spécifiques, on peut s'attendre à un très bon niveau de réactivité. Il est également aisé de modifier le comportement en changeant des composants. Les auteurs font cependant remarquer que la conception de ce type de système n'est pas toujours possible. L'adjonction de capacités autonomiques aux composants les rend plus lourds : le coût du déploiement est plus élevé. Dans les systèmes où les composants apparaissent et disparaissent à une fréquence soutenue, ce coût peut devenir inacceptable.
- **la robustesse** : ce critère évalue la capacité du système à faire face à des pannes imprévisibles. Beaucoup de systèmes sont conçus pour empêcher ou réparer les anomalies du système. Cependant leurs performances peuvent être extrêmement différentes selon que la cause des pannes est extrêmement prévisible (si elle se répète dans le temps) ou totalement aléatoire.
- **l'adaptation** : c'est-à-dire les contraintes qui régissent l'adaptation d'un système et les techniques mises en œuvre pour adapter le système.
- **le temps d'adaptation et le temps de réaction** : le temps que le processus d'adaptation met pour reconfigurer le système lorsqu'un changement de situation le nécessite. Autrement dit, le temps entre l'instant où un besoin de reconfiguration est identifié et celui où les changements sont effectifs. Il dépend des algorithmes qui analysent les données et à la recherche d'une solution.
- **la sensibilité du système aux changements de l'environnement :** un système trop sensible risque d'effectuer régulièrement des opérations inutiles. Trouver la bonne sensibilité est une tâche complexe mais extrêmement importante car elle est nécessaire à la stabilité du système.
- **la stabilisation** : le temps que le système met à connaitre son environnement et stabiliser son fonctionnement. C'est un critère particulièrement utile pour les systèmes ouverts basés sur des techniques d'auto-apprentissage.

7 SYNTHESE

Posséder des solutions autonomiques fiables pour la gestion des systèmes complexes est aujourd'hui un pré-requis pour soutenir le rythme imposé aux entreprises du secteur. Les applications ont fortement évoluées, devenant de plus en plus larges, hétérogènes et dynamiques, ce qui rend leur configuration particulièrement difficile pour les administrateurs. Les erreurs humaines représentent aujourd'hui un coût important pour les entreprises comme nous l'avons souligné dans l'introduction. C'est pourquoi le domaine de l'informatique autonomique connait un essor important avec le soutien de grandes entreprises et sous l'impulsion d'IBM.

L'informatique autonomique propose une solution à la complexité des systèmes informatiques modernes. La promesse est de diminuer l'effort nécessaire à l'administration des systèmes et donc de réduire les couts de maintenance et d'augmenter la disponibilité. Atteindre cet objectif impose de minimiser les interventions humaines en augmentant le niveau d'abstraction des interfaces d'administration. Les tâches administratives répétitives et redondantes seront déléguées au système qui devra en absorber la complexité. Celle-ci ne disparait donc pas, elle se déporte de l'administration à l'application, des administrateurs aux développeurs. Sans surprise, les systèmes autonomiques seront, par essence, plus complexes à construire que les systèmes traditionnels.

De nombreux domaines de recherche influencent l'informatique autonomique, et des travaux de nature très variée sont conduits dans ce domaine. Pour ces raisons, il n'existe pas de définition précise du domaine. A partir de quel stade un système doit-il être considéré comme autonomique ? Quelles sont les propriétés exactes d'un tel système ? A ces questions, la littérature répond par un ensemble de propriétés que nous avons énumérées et des modèles de maturité sans pour autant que le nombre ni la nature de ces propriétés ne fassent consensus. Certaines recherches se focalisent sur l'autoréparation, d'autres sur l'optimisation des performances. Cela souligne une particularité des systèmes autonomiques : ils doivent prendre en compte différents objectifs, parfois divergents, ce qui constitue une des premières difficultés dans leur construction.

Le paysage de recherche de l'informatique autonomique est donc varié. Il est principalement composé :

- **d'applications ad-hoc** qui visent à autonomiser une application particulière. C'est de cette façon que les applications étaient construites avant le développement du domaine. C'est encore notamment le cas des applications pervasives. Elles ne possèdent pas d'architecture particulière.
- **d'intergiciels et de méthodes spécialisées dans la réalisation d'une étape de la boucle autonomique**. Il existe notamment de nombreux framework spécialisés dans la supervision (par exemple [89] ou [90]) sans compter ceux issus d'autres domaines et les applications industrielles (JMX [91] ou TPTP[92]).
- **de plateformes spécialisées dans un domaine spécifique ou dans le traitement d'un problème particulier.** C'est par exemple le cas d'Unity [53] qui se focalise sur la répartition de ressources. Le grand avantage de ces plateformes est qu'elles prennent en charge une grande partie de l'autonomisation car elles ont une excellente connaissance du domaine d'application. Elles ne s'appliquent cependant qu'à un domaine réduit d'applications.
- **de plateformes génériques,** par exemple Rainbow [51] ou Automate [68] qui couvrent un large champ de domaines, d'application.

C'est à ces plateformes que nous nous intéressons en particulier. Pour ces canevas, la séparation entre adaptation et application fait consensus. L'architecture des gestionnaires, lorsqu'elle n'est pas monolithique, s'appuie généralement sur une boucle de contrôle ressemblant peu ou prou au modèle MAPE d'IBM. Il n'existe pas, en revanche, de consensus sur les techniques à utiliser pour la construction de la plateforme. La nature des politiques (règles/code),

l'architecture de gestion sous-jacente (centralisé/décentralisé) et la nature des informations contextuelles (architecture/données) sont très différentes d'une plateforme à l'autre.

L'essentiel de l'effort est généralement davantage consacré à la communication entre les différents gestionnaires ou à la récupération des données qu' à l'architecture du gestionnaire lui-même. Pourtant cette dernière est importante.

Le modèle MAPE-K décrit une excellente architecture logique qui permet une bonne compréhension de l'ensemble des étapes nécessaires à la conception d'un gestionnaire. Cependant, son application littérale produira généralement des gestionnaires mal équilibrés, peu performants, difficiles à maintenir et à étendre. En effet, la granularité des blocs proposés ne s'adapte pas à tous les types de gestionnaires : trop grande pour les gestionnaires simples et trop faible pour les gestionnaires de grande taille ou centralisés. De ce fait, peu de projets implémentent la boucle de contrôle directement en bloc MAPE. Ce dernier est fréquemment implémenté sous forme de règles ou de composants gros grains qui posent des problèmes de maintenance ou de dynamisme.

En particulier, l'architecture MAPE-K fournit peu d'indications permettant la prise en compte de plusieurs objectifs différents au sein d'un même gestionnaire. Nous l'avons dit, les gestionnaires poursuivent plusieurs objectifs (auto-protection, auto-optimisation, auto-réparation) parfois contradictoires. Des projets comme Jade [62] proposent l'implémentation de ces préoccupations sous forme de plusieurs boucles, sans pour autant apporter de solution claire aux problèmes des conflits entre les différents objectifs.

D'autre part, la possibilité de faire évoluer dynamiquement l'architecture interne du gestionnaire est un aspect souvent ignoré par la littérature. Elle nous semble cependant importante pour deux raisons. Tout d'abord les applications sous-jacentes évoluent dynamiquement dans le temps, et les gestionnaires devraient pouvoir s'adapter à ces évolutions sans qu'il soit nécessaire de les arrêter. L'extensibilité est une capacité qui nous paraît essentielle compte tenu de l'évolution actuelle des applications. Ensuite les gestionnaires sont des applications comme les autres qui consomment des ressources et qu'il peut être nécessaire d'adapter en fonction des situations. Les règles permettent le dynamisme comme dans AutoMate[68] mais nous semblent d'une granularité trop faible, ce qui entraine des coûts en maintenance et une faible réutilisabilité.

La réutilisabilité des fonctionnalités communes aux gestionnaires et des algorithmes s'avère généralement faible. Pourtant, compte-tenu de la complexité fonctionnelle et des exigences de fiabilité des applications autonomiques, fournir des solutions génériques et réutilisables apparait comme essentiel pour faciliter leur développement.

Chapitre 3 - **Approches orientées service**

Dans le chapitre précédent, nous avons mis en avant l'importance des plateformes d'exécution dans la réalisation des applications autonomiques. Le choix de la plateforme peut avoir une influence considérable sur le temps de développement, l'instrumentation des éléments administrés, la variété des adaptations possibles et également sur la flexibilité et la maintenabilité du gestionnaire.

Ce chapitre présente les approches à services. Elles se sont développées parallèlement au domaine de l'informatique autonomique et proposent des solutions aux défis de l'informatique moderne que sont l'hétérogénéité et le dynamisme croissant des applications. Ces architectures sont particulièrement prometteuses dans le développement d'applications autonomiques et simplifient grandement le travail des concepteurs.

1 UNE REPONSE A DEUX BESOINS INDUSTRIELS

La programmation orientée service (Service Oriented Computing) est un paradigme de programmation qui a été fortement popularisé par les Web Services mais qui est, comme nous le verrons, loin de se limiter à ces derniers. Elle propose des solutions à deux grands besoins de l'informatique moderne. Elle permet d'une part la réutilisation et l'intégration d'applications hétérogènes dont le patrimoine des entreprises est généralement constitué, et d'autre part la conception d'applications dynamiques donc plus adaptables et évolutives.

Gestion d'applications hétérogènes

L'histoire de l'informatique est jalonnée d'un nombre important de langages (C, ADA, Java, SmallTalk) et de technologies dont chacune répondait au besoin du moment (mainframe, application client/serveur, applications web...). Les entreprises disposent d'un héritage important d'applications qu'il serait couteux de remplacer. Malgré cette importante hétérogénéité, ces applications doivent être capables de communiquer entre elles pour répondre aux besoins des entreprises. Une des préoccupations récurrentes des DSI ([93]et [94]) est donc l'organisation de cette diversité.

Ce problème a donné naissance à la métaphore de la cité [95] qui établit un parallèle entre l'urbanisation et l'organisation des communications entre applications. De nombreuses solutions logicielles ont été proposées pour assister les urbanistes, dont les EAI (Enterprise Application Integration), les moteurs de workflow et Web Service.

Les EAI proposent de prendre en charge la communication entre les différentes applications. Elles reposent sur la notion de wrappers dont le rôle est de cacher l'hétérogénéité des données, d'adapter les protocoles d'interaction et d'établir les correspondances sémantiques. Les EAIs souffrent cependant de défauts importants : les solutions sont propriétaires et leur coût est très important ; de plus il s'agit de solutions centralisées, ce qui implique qu'une défaillance de l'EAI met à mal toute l'organisation de l'entreprise et pose des problèmes de passage à l'échelle.

Les outils de workflow proposent de modéliser et d'exécuter des processus métiers pour piloter les applications. Ces solutions souffrent néanmoins des mêmes défauts que les EAI : les solutions sont propriétaires, généralement *ad hoc* et nécessitent parfois la modification des applications.

En mettant en avant la notion de service et de contrat de services qui sont des concepts indépendants de l'implémentation, les approches à services (en particulier les Web Services) proposent de faire communiquer les applications via des standards (basés sur XML pour les Web Services). Comme pour les EAIs, il s'agit de fournir des wrappers qui offrent des services standards. Cette standardisation apporte une solution efficace au problème de l'hétérogénéité ; il est alors possible de faire de l'orchestration et de la chorégraphie.

Le dynamisme

La qualité des logiciels n'a cessé de s'améliorer au cours de ces dernières années avec l'adoption massive de processus de développement efficaces et d'outils d'évaluation (tel que le CMM-I[4] des entreprises) ; pourtant les produits sont rarement parfaits dès leur sortie. Les temps de mise sur le marché se réduisent et de fait il faut être en mesure de présenter des correctifs après la sortie des logiciels, des mises à jour ou des ajouts de fonctionnalités. L'offre de logiciel s'est progressivement transformée pour passer des logiciels monolithiques rarement mis à jour de type Word, aux plateformes proposant des services à la carte : les Set Top Box des fournisseurs

[4] Le CMM-I(Capability Maturity Model Integration) est un référentiel de critères permettant d'évaluer les fournisseurs de logiciels. Voir http://www.sei.cmu.edu/cmmi/

d'accès français offrent un bon exemple de plateforme. L'objectif est de fidéliser les utilisateurs à une plateforme en proposant des services qui évoluent dans le temps pour rester concurrentiels : téléphonie, puis télévision, puis Vidéo On Demand. L'offre doit être évolutive, les logiciels se doivent d'être dynamiques pour éviter les interruptions de services dues à une mise-à-jour.

Les méthodes de conception logicielle traditionnelles ne supportent pas le dynamisme. On assiste en réponse à l'évolution des contraintes et des technologies, à l'émergence de nouvelles méthodes de conception. Elles se sont développées progressivement avec l'évolution des besoins. La programmation orientée service que nous allons présenter veut prendre en compte le dynamisme. Elle repose sur la notion de services qui, lorsque l'architecture d'exécution le permet, peuvent être remplacés en cours d'exécution. L'implémentation n'est en effet chargée en mémoire qu'au moment de l'utilisation (lazy binding)

2 PRINCIPES

2.1 LA NOTION DE SERVICE

Le concept de service est de mieux en mieux défini dans la littérature. Comme souvent en informatique le terme service est une métaphore qui fait référence ici à la notion de service utilisée communément dans les activités humaines. La définition la plus fréquemment citée est celle de Papazoglou :

- « *Services are self-describing, platform agnostic computational elements* ».

Bien que minimaliste cette définition met en avant l'une des propriétés fondamentales des services : l'indépendance par rapport à l'implémentation. Toutefois, les services sont généralement implémentés au moyen de composants, si bien qu'il existe parfois des confusions entre le concept de service et celui de composant [96]. Par exemple la définition suivante donnée par [97] fait explicitement mention aux composants :

- *« A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract.».*

Un service n'est pas lié à une technologie : ce n'est pas une portion de code. Comme le soulignent les auteurs de [93] :

- *«Les services offrent une vue logique des traitements ou données existant déjà ou à développer. Chaque service encapsule ces traitements et données et masque ainsi l'hétérogénéité du système d'information ».*

C'est une des différences principales avec les composants qui sont quant à eux beaucoup plus liés au code et à l'environnement d'exécution.

L'objectif de l'approche orientée service est de réduire la dépendance entre les entités qui composent l'application : autrement dit de réduire le couplage entre les services. Une seconde définition très couramment retenue est celle du consortium OASIS qui se propose de standardiser les services :

- *« A service is a mechanism to enable access to one or more capabilities, where the access is provided by using a prescribed interface and is exercised consistent with constraints and policies as specified by the service composition. A service is accessed by means of a service interface where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider. A service is opaque in that its implementation is typically hidden from the service consumer except for (1) the information and behavior models*

exposed through the service interface and (2) the information required by service consumers to determine whether a given service is appropriate for their needs. ».

Pour le consortium OASIS, un service fournit un ensemble de fonctionnalités décrites dans une spécification, appelée interface, ainsi qu'un ensemble de contraintes et de politiques d'accès aux fonctionnalités offertes. L'implantation du service n'est pas visible à l'utilisateur. Seules les informations qui peuvent permettre de déterminer si le service correspond aux besoins de l'utilisateur sont disponibles. Nous pouvons noter qu'il est tout de même difficile de discerner une information utile d'une information inutile pour le choix d'un service.

Certaines définitions s'attachent davantage à l'utilité de la description des services et à leurs interactions [98] :

- *« A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should be ideally be governed by declarative policies and thus support a dynamically reconfigurable architectural style. »*

Le modèle d'interaction est ici clairement décrit. Dans un premier temps, les fournisseurs de services décrivent le service fourni. Puis, le client recherche le service correspondant à ses besoins en fonction de la description de service fournie. Enfin, le client se lie au fournisseur et il l'invoque. Ces actions peuvent être réalisées avant ou pendant l'exécution de l'application à services.

Sur la base de ces définitions, nous utiliserons cette définition issue de [99] :

- *« Un service est une entité logicielle qui fournit un ensemble de fonctionnalités définies dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. A partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties. »*

2.2 L'APPROCHE A SERVICES, UN STYLE ARCHITECTURAL

Les termes SOC (Service Oriented Computing) et SOA (Service Oriented Architecture) sont souvent employés comme des synonymes et il est difficile de faire émerger de la littérature des clivages importants dans l'utilisation de ces deux termes. Ce manque de clarté est tout naturel pour des concepts aussi récents, l'essentiel de l'effort se portant sur les aspects technologiques. De plus, de par l'hégémonie des services Web, de nombreux articles font référence à cette technologie, c'est-à-dire à une architecture (SOA) qui met en œuvre les principes de l'approche à services (SOC) ; les auteurs utilisent alors un terme pour l'autre. Ainsi, l'approche à services (Service Oriented Computing) n'est pas une architecture de référence mais un paradigme de programmation, ou un style architectural, au même titre que la programmation orientée objet ou l'approche à composants. Elle se décline comme nous le verrons en plusieurs architectures à services : les SOA[5], dont la plus connue est celle des services Web.

Nous présentons ici d'abord les principes du SOC. L'objectif principal de l'approche à service est de permettre le développement et l'évolution séparée des constituants d'une application en réduisant leur dépendance. Ce faible couplage est obtenu par la mise en œuvre d'un modèle d'interaction dans lequel interviennent trois acteurs (figure 17) :

[5] SOA : acronyme de Service Oriented Architecture.

- **un fournisseur de services** qui s'engage à fournir le service décrit dans une **spécification de service** et qui donne les informations nécessaires à sa sélection et sa bonne utilisation. Ce sont les entités logicielles qui fournissent l'implémentation du service.
- **un consommateur de service** est un utilisateur de services. Il requiert les services répondant à une spécification de services donnée.
- **un courtier de service** qui fournit un ensemble de mécanismes permettant la publication et la découverte de services. Ce courtier met en relation des fournisseurs avec les consommateurs. Il est très souvent désigné par annuaire de services, ou registre de services, ce qui laisse penser qu'une base de données est utilisée mais d'autres mécanismes plus distribués existent.

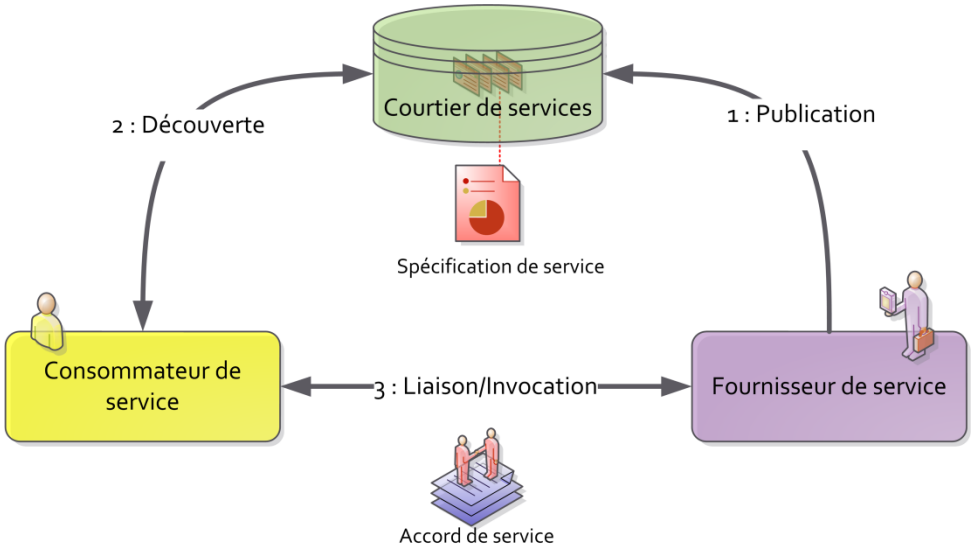

Figure 17 : Interactions entre les acteurs de l'approche à services

Il faut souligner qu'un consommateur de services peut également fournir des services (et réciproquement). Par métonymie, le fournisseur de service est parfois désigné par le service qu'il implémente. L'approche qui organise la mise en relation des consommateurs avec les fournisseurs s'effectue en trois temps, au moyen de primitives de communication :

- la **publication du service** par le fournisseur. Il s'annonce auprès du registre de service en fournissant la description du service fourni. Il lui transmet les informations nécessaires à la découverte et à l'utilisation du service.
- la **recherche de services** par les utilisateurs du service. Les consommateurs de services interrogent le courtier de services pour localiser les fournisseurs de services qui correspondent à ses besoins.
- L'établissement de la **liaison** entre un consommateur et un fournisseur de services. Une fois le fournisseur sélectionné par le consommateur, il peut y avoir une phase de négociation pour déterminer les conditions d'utilisation du service. C'est la réalisation d'un accord de service [100]. Le consommateur peut alors utiliser le service dans les conditions définies.

L'indépendance entre les constituants de l'application est rendue possible par deux aspects. D'abord, l'introduction du rôle de courtier de service permet d'augmenter l'indépendance des constituants de l'application, en particulier à l'exécution puisqu'il n'est plus nécessaire pour le client de connaitre ses fournisseurs avant l'exécution. Ensuite, l'une des caractéristiques fondamentales de l'approche à services est que l'assemblage d'une application à services est réalisé à partir des **spécifications de services**. Une spécification s'exprime dans un langage qui

est indépendant des technologies utilisées pour l'implémentation des fournisseurs et consommateurs. Elle se trouve au cœur des échanges entre les services. Elle contient [101] :

- **des informations syntaxiques** : l'interface du service contient des informations syntaxiques. Elle est décrite au moyen d'un IDL (Interface Description Language) qui correspond parfois au langage utilisé pour l'implémentation du service, comme JAVA par exemple, ce qui peut prêter à confusion.
- **une description de la sémantique** des opérations proposées par l'interface.
- **des propriétés** permettant la sélection du service telles que la configuration actuelle du service (la langue, la position géographique,...) mais aussi des informations relatives aux qualités de service.

2.3 UNE APPROCHE SOC, DES ARCHITECTURES SOA

Comme nous l'avons expliqué plus haut, le SOC définit principalement le modèle d'interaction entre les fournisseurs et les utilisateurs de services. Il existe de nombreuses façons de mettre en œuvre ces principes. Une architecture à services (SOA) est une réalisation particulière de cette approche qui définit un environnement d'intégration et d'exécution des services en proposant un ensemble de technologies cohérentes. Un environnement d'exécution fourni (figure 18) :

- **des mécanismes de base** qui rendent possible la publication, la découverte, la communication, la négociation, la composition, la contractualisation et l'invocation de services.
- **des mécanismes supplémentaires** qui prennent en compte les besoins non-fonctionnels des applications tels que la sécurité, la gestion des transactions ou la gestion des services.

Ces mécanismes sont généralement réalisés par une pile de protocoles comme le représente la figure 18. La représentation de cette pile, en particulier la place des activités transversales, varie d'une organisation et d'un auteur à l'autre [102]. Les **protocoles de transport** spécifient comment les informations sont échangées entre les applications sur un support de communication (autrement dit sur une machine ou sur internet). A ce niveau, il n'est pas fait mention de la nature des données échangées – on peut, en cela, comparer ces protocoles à ceux de la couche transport des modèles OSI. Ces protocoles sont souvent éprouvés et utilisés dans d'autre contexte que celui des architectures à services. Les **protocoles de communication** s'appuient sur la couche transport et permettent l'échange de requêtes. Ils définissent la nature des messages échangés entre les acteurs du SOA et l'organisation des échanges (synchrone, asynchrone). Dès ce niveau, les protocoles sont généralement destinés uniquement aux architectures à services.

La **description des services** est fournie par les pourvoyeurs de services pour décrire les opérations possibles au sein du service et les informations syntaxiques telles que les paramètres d'entrées/sorties et leur typage. La description de la syntaxe s'appuie généralement sur un langage de description d'interface (IDL). Les utilisateurs peuvent ainsi connaitre les capacités fonctionnelles et non fonctionnelles du service que s'engage à fournir le pourvoyeur. La technologie utilisée pour implémenter les **services** de la couche services n'est pas fixée, ce qui est une caractéristiques de l'approche. Toutefois, certains SOA contraignent de fait la technologie utilisée, c'est par exemple le cas de la plateforme à services OSGi sur laquelle les services sont implémentés en JAVA. Enfin au sommet se trouve l'ensemble des technologies permettant d'organiser la **composition des services.** Autrement dit, les mécanismes permettant l'assemblage des services pour former une application.

Le **courtier de service** est ici représenté transversalement car il est parfois, fourni sous la forme d'un service, notamment dans le cas des services Web. Ainsi s'appuie-t-il sur tous les

protocoles et peut-il bénéficier des améliorations proposées dans les mécanismes non-fonctionnels.

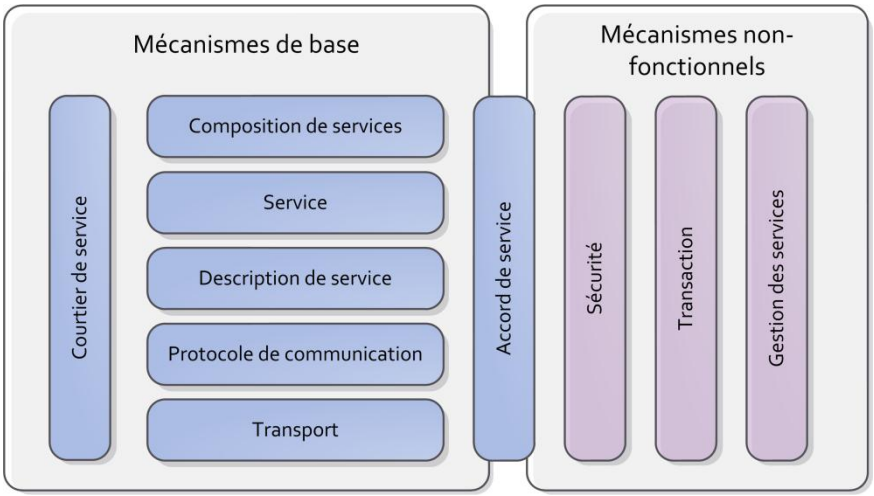

Figure 18 : Environnement d'intégration de services [103]

L'**accord de service** (ou contrat de service) fixe les conditions dans lesquelles le service sera utilisé par le client le requérant. Le contrat lie donc les pourvoyeurs et consommateurs de services. Il spécifie notamment la liste des opérations offertes, les fonctionnalités, ainsi que des caractéristiques non-fonctionnelles comme la qualité de service. C'est pourquoi, dans cette représentation, le contrat appartient aux deux parties. Dans la pratique, la définition de contraintes portant sur les aspects non-fonctionnels est difficile à mettre en œuvre.

Trois mécanismes additionnels sont représentés ici car ils ont fait l'objet de protocoles (en particulier pour les services Web) :

- la Sécurité : la grande ouverture des architectures à services fait qu'il est parfois nécessaire de garantir que les données échangées entre les services ne seront pas interceptées. Si des échanges sécurisés sont possible avec des méthodes traditionnelles par exemple l'utilisation de tunnel SSL, le risque est grand de voir augmenter exponentiellement le nombre de tunnels et de certificats. Ainsi la sécurité fait-elle aujourd'hui l'objet d'étude particulière dans le cas des approches à services avec, par exemple, les protocoles WS-security[104], XACML[105] ou SAML[106]. Voir à ce sujet [99].
- les transactions : la gestion des transactions joue un rôle important dans les applications modernes. C'est un aspect crucial pour toutes les opérations commerciales. Lorsque les services sont utilisés en collaboration, les transactions permettent d'assurer la cohérence des données.
- la gestion des services : les plateformes à services doivent fournir des outils permettant de piloter les services et de produire des vues et des synthèses sur le déroulement de l'exécution des applications.

A partir de cette représentation, il est facile d'identifier l'une des premières difficultés des approches à services. La multiplication des protocoles rend l'apprentissage des technologies parfois laborieux. D'autre part, les SOA se révèlent rarement interopérables, ce qui est paradoxal compte tenu de l'objectif des approches à services. La communication entre les différents SOA est aujourd'hui une thématique fleurissante et certaines plateformes fournissent des ponts entre les technologies [107] [108]. Ainsi certaines technologies qui se prétendent standards sont dans les faits simplement ouvertes, car trop peu utilisées.

Une autre représentation très citée dans la littérature est la pyramide de l'architecture à services étendue, proposée par Papazoglou[109](figure 19). La base de la pyramide est constituée des mêmes mécanismes de base que ceux décrits précédemment : publication, découverte, sélection, liaison avec le cœur la spécification. Leur réalisation est un pré-requis au faible couplage. L'étage central de la pyramide contient les mécanismes de composition mais également les technologies permettant d'assurer les transactions et de vérifier la conformité des compositions de services. On constate qu'en fonction des auteurs la classification des fonctionnalités dans les catégories fonctionnels/non-fonctionnels changent. Au sommet, l'auteur place les outils d'administration qui englobent l'administration des services, de la plateforme et des services. Enfin les propriétés orthogonales, telle que la sécurité, apparaissent transversalement. La plupart des SOA ne réalisent pas la pyramide dans son intégralité et la mise en œuvre reste délicate. Les derniers niveaux de la pyramide, bien qu'indispensables pour donner aux administrateurs une vision globale des applications, sont rarement fournis.

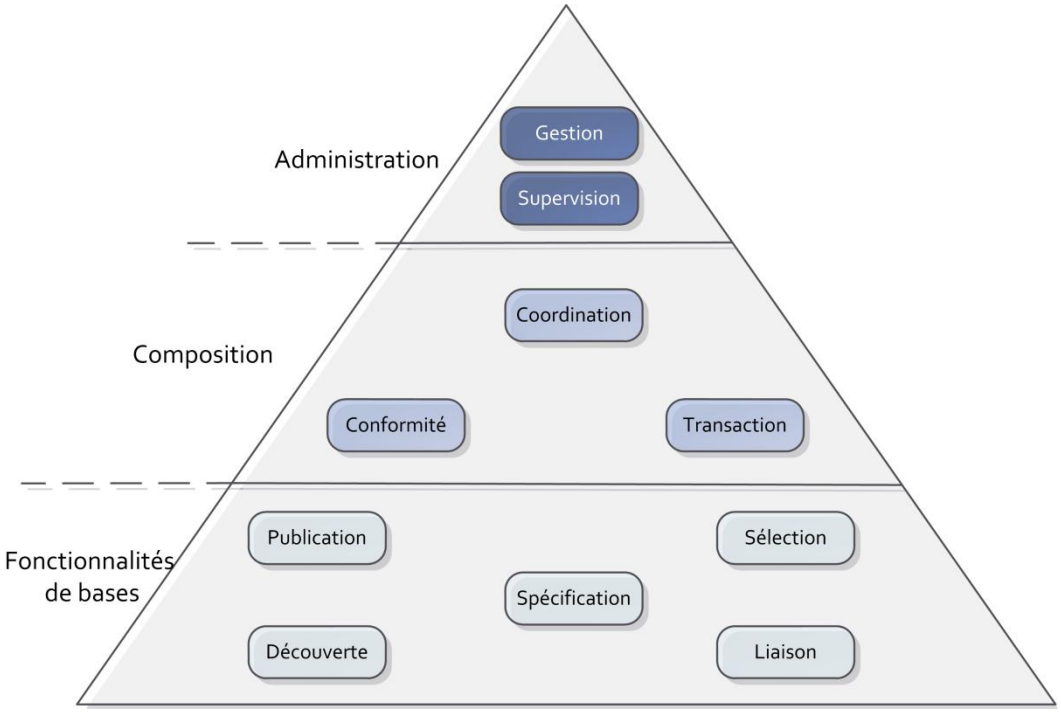

Figure 19 : Fonctionnalités d'un SOA étendue [109]

2.4 NOTION D'APPLICATION A SERVICES

Définition et propriétés

Les applications à services sont issues, nous l'avons dit, de l'interaction des consommateurs et des fournisseurs de services. Définir le périmètre exact d'une application à services, c'est-à-dire quels fournisseurs et consommateurs appartient réellement à l'application, est souvent délicat. Imaginons par exemple qu'un fournisseur de services, un correcteur orthographique, soit utilisé par plusieurs consommateurs de services, des traitements de textes. Doit-on ici considérer qu'il y a trois applications – un correcteur orthographique et deux traitements de textes - ou deux applications, c'est-à-dire deux traitements de textes intégrant un correcteur d'orthographe ? Le même problème se pose lorsque l'application utilise des services fournis par la plateforme d'exécution (par exemple le MOM d'OSGi). Ces services font- ils partie de l'application ? Le problème n'est pas nouveau puisqu'on le rencontre pour les librairies partagées dans un système d'exploitation. Cette difficulté résulte en partie du fait que dans

l'approche à services, les constituants de l'application ne sont pas la propriété de l'application. Avec les applications dynamiques ou adaptatives, la définition est encore plus floue puisque elles évoluent en fonction de l'environnement.

Nous considérons ici que la notion d'application est liée à l'intention du concepteur. En général cet objectif est évident lorsque l'application a été modélisée et décrite, par exemple au moyen d'un langage de définition d'architecture (ADL). Nous prendrons donc une définition générale : une application est un ensemble *cohérent* de services *hétérogènes* et *distribués*, qui peut requérir les services d'autres applications, et qui fournit des services à des utilisateurs physiques ou logiques (par exemple d'autres applications).

La mise en œuvre du modèle d'interaction à services confère aux applications les propriétés suivantes [110] :

- **la distribution** : les constituants d'une application à services sont souvent distribués. Ils peuvent se trouver sur différents nœuds d'un réseau de façon transparente.
- **l'hétérogénéité** : les constituants d'une application à services évoluent dans un environnement hétérogène. Les entreprises et les développeurs des clients utilisant des services ne sont généralement pas ceux qui les fournissent. Les détails de l'implémentation du fournisseur de services ne sont pas connus lors de l'implémentation du client.
- **les liaisons tardives** entre fournisseurs de services et consommateurs. Un consommateur de services ne se lie à son fournisseur que lorsqu'il en a réellement besoin.
- **le masquage de l'hétérogénéité** : les fournisseurs de services et les consommateurs de services n'ont pas à connaitre les détails de l'infrastructure de communication. Les services offrent une vision logique qui peut être très différente de la réalité physique.
- **le dynamisme,** lorsque l'architecture à services le prend en compte. Le faible couplage et la liaison tardive font qu'un grand nombre de décisions (configuration, choix des services par exemple) peuvent être prises à l'exécution et non à la conception.

Les techniques de composition

L'approche à services encourage la conception de nouveaux services par la composition de services existants. La composition décrit comment s'opère l'assemblage des services et leur coordination.

La définition d'une composition s'effectue en plusieurs étapes. Il faut en premier lieu identifier les fonctionnalités abstraites fournies par les différents constituants de l'application. Le concepteur doit ensuite identifier les fournisseurs de services disponibles dans les dépôts avant l'installation ou dans les registres s'ils sont déjà disponibles. Il est alors nécessaire de procéder à la sélection parmi la liste des candidats constituée précédemment. C'est ici que la description des services est utilisée pour déterminer quels services sont les plus intéressants pour l'application, conformément à une stratégie de sélection donnée. La construction de l'application s'achève par la négociation et la configuration des services. Enfin, une fois l'application construite, il faut coordonner l'exécution de ces services. Les propriétés des applications à services et des services, en particulier la réutilisabilité et les performances, sont fortement liées à la technique de composition choisie. On peut distinguer deux grandes catégories de techniques de composition (figure 20) :

- la **composition à base de procédés**, ou comportementale, est de loin la plus utilisée actuellement avec notamment le langage BPEL[111]. Un procédé décrit l'ensemble des activités, et des actions à effectuer pour réaliser un objectif donné.

Un modèle de procédés est souvent décrit sous la forme d'un graphe : les nœuds sont les activités et les arcs modélisent les données échangées entre les activités. Cette approche se focalise donc sur la notion d'activité et non directement sur la notion de spécification de services. Il existe un découplage fort entre les services puisqu'ils n'ont aucune connaissance les uns des autres. Ainsi, un service ne peut pas savoir à qui les données qu'il produit seront transmises. Souvent seul le mécanisme de composition (orchestrateur ou canevas d'exécution) connaît l'ensemble des services qu'il peut utiliser, de fait les registres de services sont peu utilisés.

- la **composition structurelle** est la forme traditionnelle de composition dans les approches à composants. Elle se base sur la notion de langage de description d'architecture (ADL) qui permet de décrire les dépendances entre des unités de code. Avec cette technique, la logique de composition est disséminée dans les constituants de l'application. Ces derniers connaissent les relations qui les unissent aux autres. Dans les modèles à composants orientés service modernes (SCA, iPOJO), les dépendances s'expriment en termes de spécifications de services ce qui augmente le découplage entre les composants.

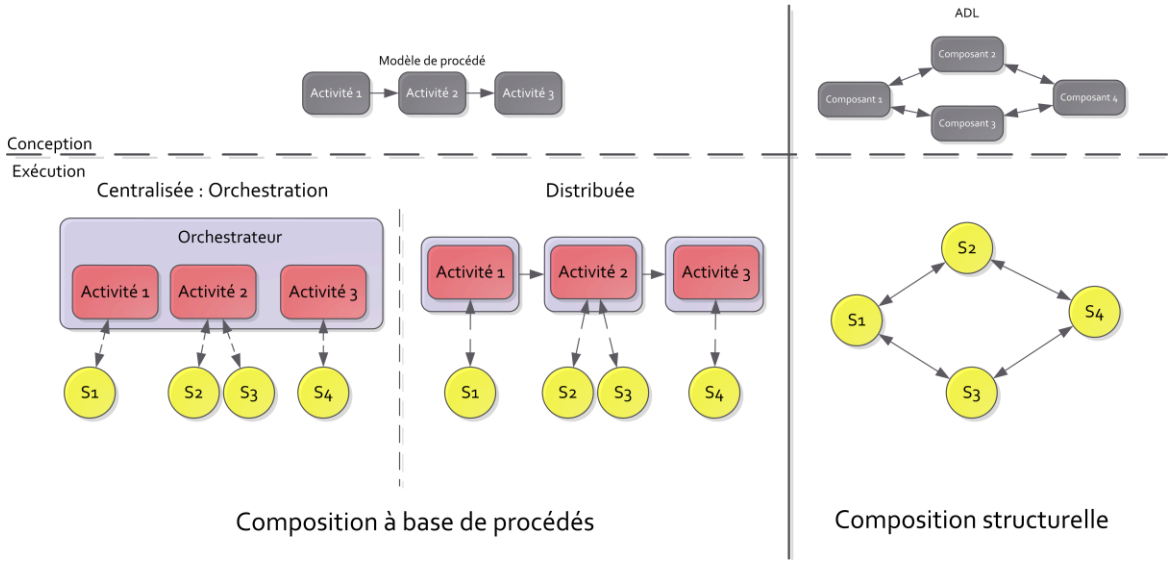

Figure 20 : Différentes techniques de composition

Lorsque l'on effectue une composition à base de procédés, il faut choisir entre deux organisations du contrôle :

- **centralisée** : on parle **d'orchestration** qui, comme son nom l'indique, prend pour image celle du chef d'orchestre et des musiciens. Ici chaque service intervient lorsque le contrôleur central l'ordonne. Les enchainements sont donc gérés dans une seule entité centralisée, l'orchestrateur, ce qui induit une grande cohérence dans les actions prises. Cet orchestrateur se charge des appels et de la collecte des résultats en se basant sur un modèle de procédé.
- **distribuée** : le concepteur tente ici de remédier aux problèmes de performances qui peuvent se poser dans l'orchestration. Une technique consiste à diviser l'orchestrateur en plusieurs unités de code réalisant chacune une activité donnée. Cela permet de limiter les coûts en communication car le mécanisme de coordination se trouve plus près des services qu'il appelle. On parle alors **d'orchestration distribuée** (ici représentée par la figure 20). Enfin des travaux cherchent à faire communiquer les services directement entre eux en respectant le

modèle de procédé. Il s'agit alors de **chorégraphie**, non représenté par la figure 20 car très proche dans sa représentation de la composition structurelle.

Ces modes d'organisation font naturellement écho aux problématiques rencontrées également dans l'informatique autonomique et l'on retrouve ici les mêmes problèmes que ceux évoqués dans le premier chapitre, avec la nécessité de choisir entre deux systèmes sans pouvoir en cumuler les avantages, à savoir : plus de réactivité et passage à l'échelle en distribué, plus grande cohérence en centralisé.

2.5 DYNAMISME DANS L'APPROCHE A SERVICES

Avec le faible couplage, les possibilités offertes pour faire évoluer une application de façon dynamique sont fortes. Le modèle d'interaction proposé par l'approche à services ne détermine pas quand les services se lient entre eux. Cette liaison peut se produire durant les trois instants du cycle de vie :

- **lors du développement** : le consommateur connait la localisation de ses fournisseurs dès le développement. Au mieux, le consommateur possède une liste de services qu'il utilisera pendant l'exécution, au pire la liaison est opérée directement dans le code du consommateur. Dans tous les cas cette technique affecte grandement les capacités d'adaptation de l'application durant l'exécution car il n'est souvent plus possible de modifier ces liaisons. Elle limite également l'intérêt du registre de services.
- **lors du déploiement** : lorsque le consommateur est déployé, les services sont sélectionnés parmi ceux disponibles.
- **lors de l'exécution** : lorsqu'un pourvoyeur de services disparait, les clients recherchent de nouveaux services fournissant un service équivalent.

Deux propriétés s'avèrent essentielles dans les approches à services dynamiques.

La **découverte dynamique** est la possibilité pour un client de découvrir un fournisseur à l'exécution (généralement via un ou plusieurs courtiers de services). Elle ne décrit pas le comportement que le client doit adopter lors de la disparition du service. C'est une propriété fondamentale de l'approche à services [112].

La **disponibilité dynamique** « fait référence au fait que la présence des composants qui constituent une application n'est pas statique, mais peut changer lors de l'exécution d'une application, potentiellement en dehors du contrôle de celle-ci »[101]. Les SOAs qui gèrent la disponibilité dynamique proposent des mécanismes permettant de réagir à la disparition ou à l'apparition d'un service (fournir un service équivalent) : on parle alors de SOA dynamique [113]. Le dynamisme nécessite en conséquence l'ajout de deux nouvelles primitives (illustrées figure 21) :

- le **retrait de service** : lorsqu'un service n'est plus disponible, il se retire du courtier de services.
- la **notification** : le courtier de services notifie les consommateurs lorsqu'une modification intervient dans le registre. A chaque inscription d'un nouveau pourvoyeur de services, le courtier avertit les consommateurs : ils peuvent l'utiliser. Chaque départ de service est également signalé. Les notifications peuvent également concerner d'autres modifications dans l'environnement d'exécution, comme les modifications de propriétés. Pour prévenir les consommateurs, le courtier de services doit diffuser un message sur un bus, ou garder une liste de tous les consommateurs.

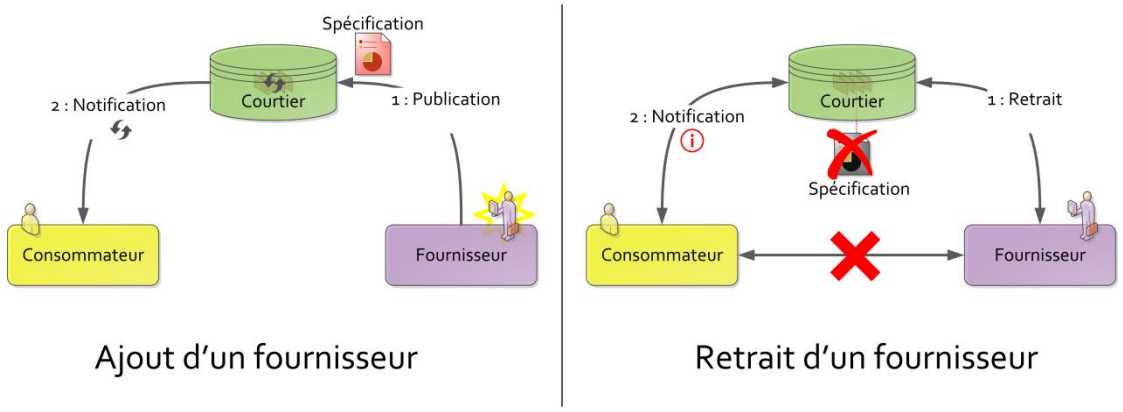

Figure 21 : Gestion du dynamisme dans l'approche à service

Il est possible d'étendre la pyramide proposée par Papazoglou pour prendre en compte le dynamisme [114] en ajoutant ces deux primitives à la base de la pyramide. Nous pouvons aussi compléter les niveaux supérieurs par des fonctionnalités autorisant la conception et la modification d'applications dynamiques, en particulier les mécanismes permettant de superviser, voire modifier, l'architecture de l'application.

3 EXEMPLES D'ARCHITECTURES A SERVICES

Il existe aujourd'hui de nombreuses architectures à services : Jini[115], UPnP[116], DPWS[117], les services Web et OSGi par exemple. Nous ne présenterons ici qu'un sous-ensemble de ces technologies parmi lesquelles deux technologie très différentes : les services web et OSGI. Nous nous focalisons ensuite sur les approches à base de modèles à composants SCA et iPOJO.

3.1 LES SERVICES WEB

On constate ces dernières années un fort engouement pour le SOA de la part des entreprises. Ce dernier s'est accompagné d'une communication marketing (tout ce qui fait référence au web se vend bien) et de simplifications qui ont concouru à faire passer l'idée exagérée que le SOA se réalise uniquement à base de Web Services et l'idée fausse que l'usage de Web Services est synonyme de SOA. Les architectures à base de services Web sont actuellement les plus utilisées dans l'industrie. Elles ont atteint un niveau de maturité élevé.

Du point de vue des industriels, l'apport principal des services Web est la gestion de l'hétérogénéité des technologies. Les services Web ont été conçus pour une utilisation sur les réseaux, en particulier internet, en passant à travers les pare feux des entreprises. Ils permettent la mise en relation de systèmes d'informations appartenant à des entreprises différentes et facilitent ainsi la coordination des entreprises avec leurs fournisseurs. Ce cadre d'utilisation particulier a des conséquences sur l'implémentation de l'architecture à services Web : la gestion de l'hétérogénéité est privilégiée par rapport au dynamisme et aux performances. Les aspects permettant la sécurisation des échanges et la coordination des services sur le réseau se révèlent également très importants.

Les technologies d'exécution des services Web fournissent un cadre d'interopérabilité permettant de masquer les détails d'implémentation des fournisseurs de services [118]. Pour les utilisateurs de services, le fournisseur est une « boite noire » exposant ses fonctionnalités par l'intermédiaire d'un ou plusieurs ports. Il y a deux modes de communication entre les services Web : synchrone et asynchrone. Lors d'un appel synchrone, le client se bloque en attendant la réponse de son fournisseur. C'est le mode le plus simple à mettre en œuvre et le plus utilisé, mais il pose des problèmes lorsque le traitement du fournisseur est long ou que sa charge de travail est importante. Pour y remédier, les services peuvent également s'appeler de façon asynchrone ; le

client peut alors continuer ses traitements en attendant une réponse potentielle de son fournisseur. Les communications asynchrones sont plus complexes car elles nécessitent le partage de plus d'informations entre le fournisseur et son client : la localisation du client et du fournisseur, et un moyen d'identifier la réponse à une requête donnée.

L'architecture à services repose sur un nombre important de standards Web – XML[119], SOAP[120], WSDL[121], UDDI[122] et les WS-* par exemple – dont le but est de permettre des communications entre machines à travers un réseau. Nous représentons un sous-ensemble de ces protocoles dans la figure 22. L'une des forces de l'architecture à services est que la majorité des mécanismes évoqués plus haut sont supportés. En revanche, cette multiplication des standards dont certains se font concurrence entraine, d'un organisme à l'autre, des variations dans la vision de l'architecture à services : chacun met sa technologie en avant.

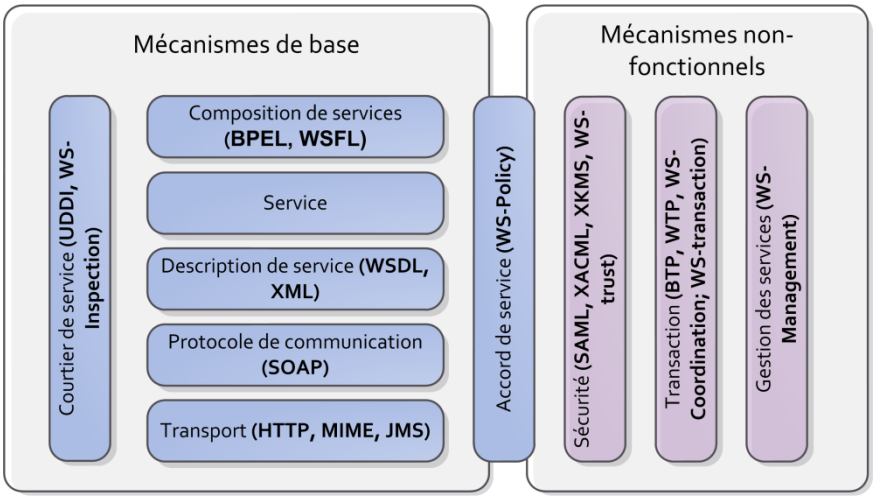

Figure 22 : Les protocoles utilisés pour les services Web

L'objectif n'est pas ici de décrire l'ensemble des protocoles utilisés dans les services Web. Nous présenterons trois spécifications de base, normalisées par le concortiumW3C : WSDL[121], SOAP[120], UDDI[122]. L'ensemble des langages utilisés par les services Web est basé sur XML[119] qui est une recommandation du W3C définissant un langage balisé textuel. La figure 23 illustre le rôle de chacun des protocoles.

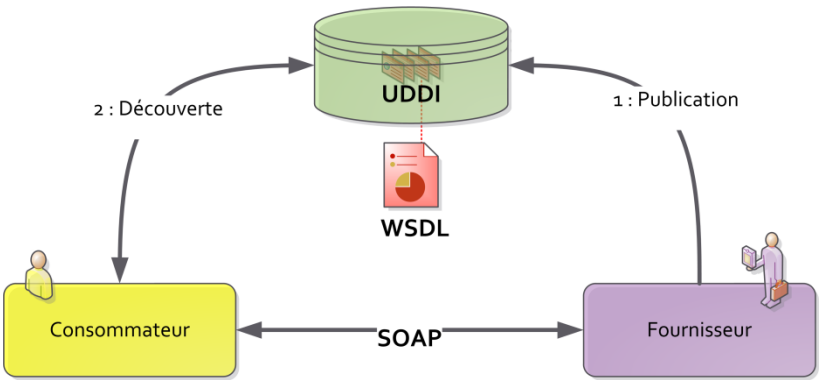

Figure 23 : Architecture des services Web

L'usage du XML n'est pas anodin. Un grand nombre de librairies supportent l'interprétation de ce format, et ce dernier est souvent utilisé dans les systèmes d'information des entreprises. En revanche, l'interprétation des messages au format XML a un cout en performance. Pour maintenir une application à services Web à un niveau de performance raisonnable, les

concepteurs doivent contenir le nombre de services et limiter au maximum leurs interactions. En conséquence, les services Web ont souvent une granularité plus forte que dans les architectures à services à appel direct telles qu'OSGi. Ils sont peu utilisés dans des applications pour lesquelles la performance prime. Les entreprises ont parfois recours à des boîtiers accélérateurs XML permettant de décharger les serveurs du traitement du XML et de gagner en performance.

La communication avec SOAP

La communication entre les services s'appuie sur le protocole SOAP [120], acronyme de Simple Object Access Protocol. SOAP est le successeur d'XML-RPC[123], technologie d'invocation de services à distance. Il gère l'échange des messages entre services en s'appuyant généralement sur le protocole standard http. Un message SOAP est constitué de deux parties :

- **d'un entête** qui contient des informations sur la nature du message : sécurité, chiffrement, identification de l'émetteur. Cette partie est facultative.
- **d'un corps du message** qui contient une requête ou une réponse SOAP, c'est-à-dire l'opération à invoquer et les paramètres d'invocation.

SOAP est un protocole simple et, principal avantage, il autorise la communication à travers les pare-feux et les proxys. Il n'est toutefois pas exempt de défauts : l'emballage, déballage des messages SOAP se révèle couteux, et la taille des messages est plus importante qu'en binaire car XML est un métalangage verbeux. Pour ces raisons il existe aujourd'hui des solutions permettant l'échange de données binaires telles que MTOM [124] du W3C. D'autre part, SOAP ne s'appuie pas toujours sur le protocole de transport HTTP mais sur des protocoles propriétaires de type RMI avec JAVA, beaucoup plus performants. Enfin, une solution alternative à SOAP est l'utilisation des principes architecturaux REST (REpresentational State Transfer[125]) pour organiser les échanges de messages de services. En comparaison des échanges SOAP, ceux de type REST sont beaucoup moins verbeux.

La spécification avec WSDL

La spécification des services est décrite en WSDL[6][121]. Ce langage permet de fournir une description du service indépendante du langage utilisé et de la plateforme sur laquelle il est déployé. La figure 24 donne un exemple de fichier de description de services écrit en WSDL. Conformément au principe du SOC, les descriptions WSLD font abstraction des détails techniques d'implémentation du service et se focalisent uniquement sur les fonctionnalités. Un fichier se compose de deux parties :

- une **partie abstraite**, indépendante de la technologie, qui décrit les objets métiers, la nature des messages et les opérations supportées par le service ;
- une **partie concrète**, spécifique à l'implémentation, qui fournit des informations sur la localisation du service et les protocoles employés lors de son utilisation. Ces aspects sont dépendants de la technologie réseau choisie.

La communication entre services se fait par échange de messages entre des terminaisons, constituées d'un ou plusieurs ports. Ces derniers sont typés. Les informations contenues dans un fichier WSDL sont les suivantes :

- **objets métiers** : c'est la description des types des données qui seront utilisés par le service Web ;
- **message** : définition abstraite de la structure et du format des messages utilisés pour accéder au service.

[6] Acronyme de Web Service Description Language

- **interface du service et opérations** : liste des opérations possibles avec les paramètres nécessaires à leur utilisation ;
- **protocole de liaison** : c'est le protocole que les clients doivent utiliser pour invoquer le service ;
- **points d'entrée du service** : localisation du service.

L'un des reproches adressés à l'encontre du format WSDL est qu'il ne permet pas d'exprimer les propriétés non fonctionnelles d'un service. Des spécifications existent pour combler ce manque, telles que WS-agreement[126], WSMO[127] ou WS-Policy[128].

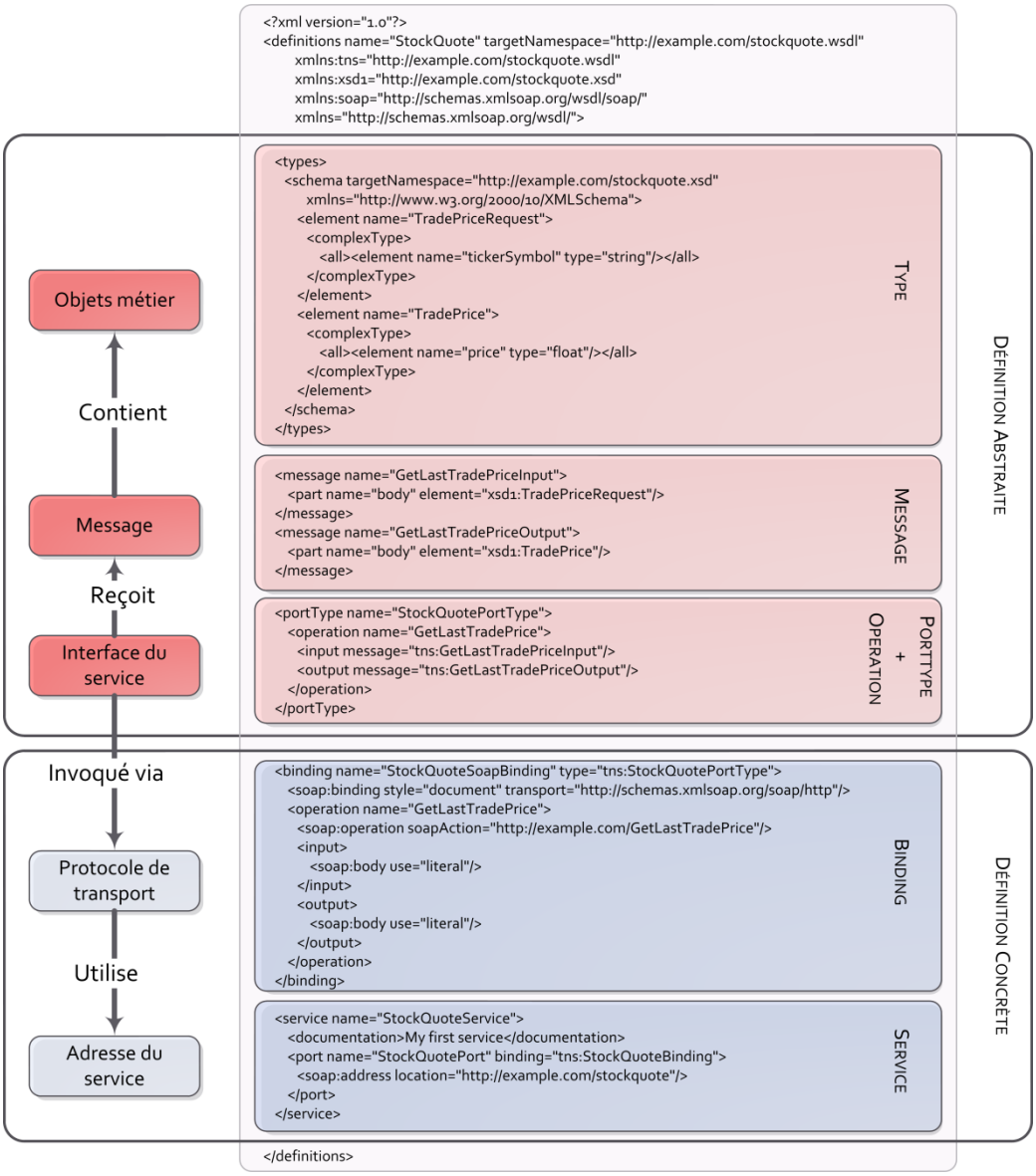

Figure 24 : Exemple de fichier WSDL inspiré de [93][99]

La découverte avec UDDI

La découverte se fait au moyen d'UDDI[7] : un système d'annuaire, à l'origine norme du W3C reprise par l'OASIS[122]. Il est construit sur la base de SOAP et WSDL. L'annuaire est consultable via un système de page : pages vertes, jaunes et blanches qui donnent des informations de nature différente. Les pages vertes décrivent les contrats WSDL. Les pages jaunes contiennent les catégories de services (territoire géographique, type de produit, secteur d'activité, ...). Les pages blanches informent sur les fournisseurs de services (nom de l'entreprise, description, coordonnées, ...).

Dans la pratique UDDI est extrêmement peu utilisé, sans compter que les notifications ne sont généralement pas implémentées. La disponibilité dynamique est de fait rarement gérée.

Synthèse

La force des services Web réside dans la distribution et la standardisation. En revanche, l'utilisation de standard XML a un cout important au niveau des performances (emballage/ déballage) : la communication entre deux services Web est beaucoup plus couteuse [96] qu'entre deux composants. Il faut également souligner qu'en pratique, à l'instar de la plupart des standards Web (HTML, CSS, XHTML), ces standards ne sont pas toujours bien respectés (SOAP en particulier) : les services fournis sont souvent décrits de façon presque standard, si bien que les clients doivent avoir un certain niveau de tolérance pour pouvoir les utiliser. La multiplication des standards est également un frein et nuit en définitive à l'interopérabilité. D'un autre coté, le grand nombre de protocoles offre une grande flexibilité aux entreprises et l'architecture à services Web reste une des rares à offrir une palette aussi large de propriétés fonctionnelles prises en compte – en dépit du fait qu'un certain nombre de ces protocoles n'ait pas atteint un niveau de maturité suffisant.

3.2 OSGI

OSGi Alliance[8] est un consortium indépendant fondé en 1999 qui regroupe plus de 70 compagnies dont IBM, Nokia, BMW et Alcatel. Il s'est donné pour mission de créer des spécifications ouvertes et de favoriser l'adoption massive par l'industrie d'une plate-forme ouverte de livraison et de gestion de services. OSGi est une spécification de plateforme à services qui définit un environnement d'exécution et de déploiement de services ainsi qu'un ensemble de services techniques : par exemple des services de monitoring, d'échange de messages et d'administration.

La spécification OSGi [129] vise, à l'origine, des plateformes aux performances modestes : les plateformes résidentielles et les équipements réseaux. L'objectif est d'autoriser l'administration et la mise à jour des applications de façon transparente durant leur exécution, sans les arrêter. Les passerelles résidentielles se sont démocratisées en France sous la forme de Set Top Box telles que Freebox ou Livebox. Ces passerelles étaient à l'origine limitées en capacité mémoire et puissance de calcul ; le surcoût induit par l'interprétation du XML rendait par exemple l'usage des services Web inapproprié pour ces plateformes. La spécification OSGi a alors mis l'accent sur l'économie des performances et des coûts mémoire. De nos jours, cette spécification est utilisée pour une grande variété de domaines : dans les téléphones portables ou dans les serveurs d'applications par exemple. Les plateformes OSGi peuvent être employées pour développer des applications dynamiques de grande taille comme OW2 Jonas[9].

[7] Acronyme de *Universal Description, Discovery and Integration*

[8] http://www.osgi.org/

[9] http://wiki.jonas.ow2.org/xwiki/bin/view/Main/WebHome

Il existe aujourd'hui de nombreuses implémentations de cette spécification, par exemple Felix[10] et knopflerfish[11]. OSGi est en particulier le support d'exécution d'un environnement d'exécution très populaire, Eclipse, avec une implémentation nommée Equinox[12]. En théorie les services OSGi sont interopérables et peuvent donc être utilisés d'une implémentation à l'autre. En pratique il existe encore des différences qui rendent une adaptation encore souvent nécessaire – ces différences s'atténuent à mesure que la spécification évolue et se précise. Ces spécifications s'appuient sur des machines virtuelles permettant de supporter le dynamisme à l'exécution des applications, ce qui a orienté le choix du consortium sur JAVA. La figure 25 représente une plateforme OSGi classique.

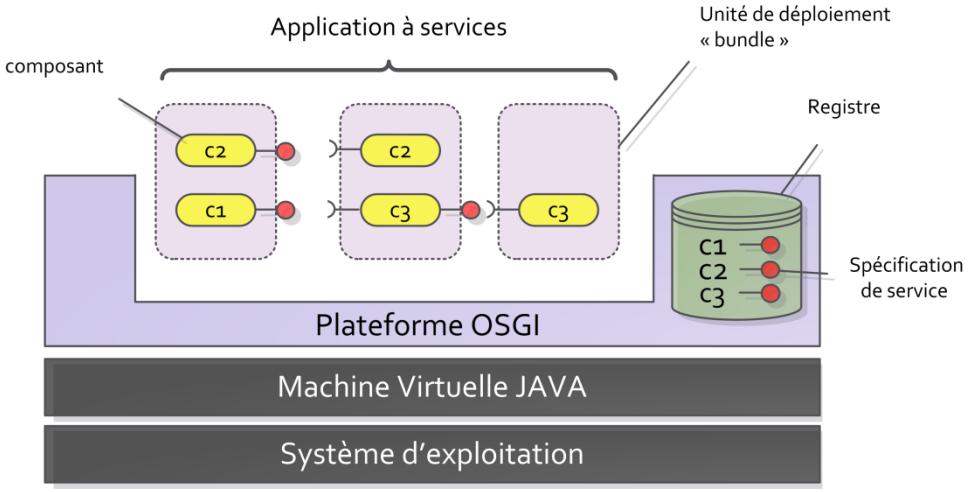

Figure 25 : Plateforme à service OSGi

L'une des forces d'OSGI est de prendre en charge le cycle de vie des applications à services. Les applications sont conçues de façon très modulaire par l'interconnexion d'unité de déploiement : les **bundles** (figure 26).

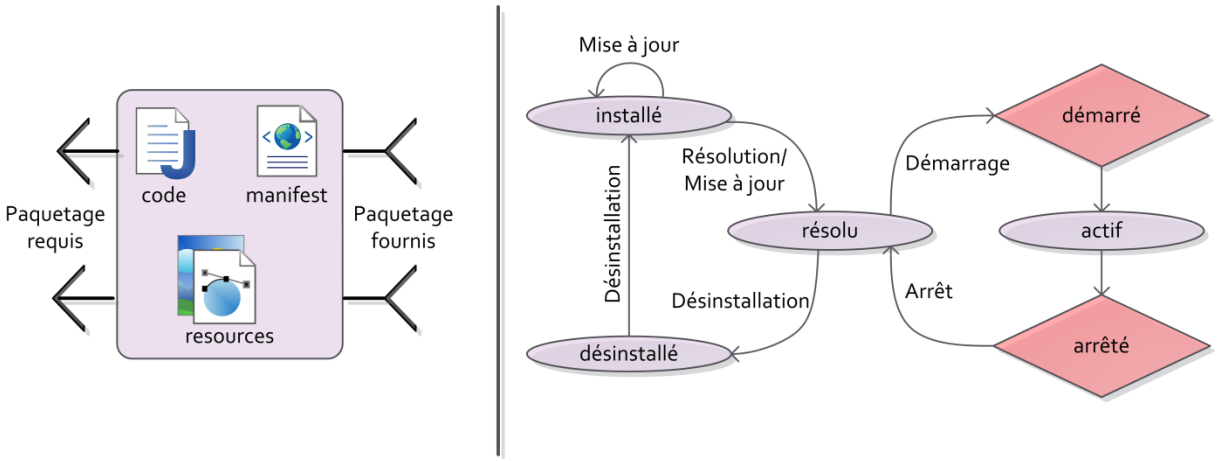

Figure 26 : un bundle et son cycle de vie

[10] Projet apache : http://felix.apache.org/site/index.html
[11] http://www.knopflerfish.org/
[12] http://www.eclipse.org/equinox/

Un bundle est implémenté sous la forme d'un fichier JAR contenant du code compilé, des ressources (images, fichiers de configuration) et un ensemble de métadonnées contenues dans un manifest. Les bundles expriment des dépendances de code en déclarant une liste de paquetages java nécessaires à leur fonctionnement. Ces paquetages sont fournis par la plateforme ou par d'autres bundles. Un bundle peut être déployé, installé, supprimé et administré durant l'exécution des applications sans affecter l'exécution des bundles qui ne dépendent pas de lui et sans arrêter la plateforme. Avant de démarrer un bundle, la plateforme s'assure que toutes les dépendances de code sont résolues. Les plateformes OSGi proposent même des mécanismes permettant de prendre en compte les différentes versions d'un même paquetage. La figure 26 illustre le cycle de vie d'un bundle.

Description de service

La spécification OSGi ne prend en charge que la base de la pyramide de Papazoglou. Elle définit une architecture à services dynamique très fortement liée avec le langage de programmation JAVA comme illustré par la figure 27.

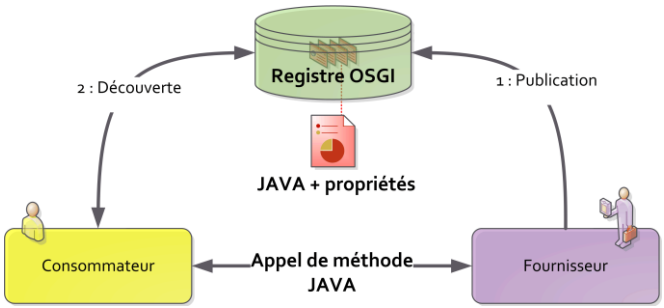

Figure 27 : Architecture à Service OSGI

La plateforme OSGi fournit un annuaire aux services qu'elle héberge. Ce registre est centralisé et permet aux services hébergés par la plateforme de se découvrir. La spécification ne fournit en revanche pas de mécanismes permettant la communication entre les plateformes. En pratique, les applications OSGi sont donc plus difficiles à distribuer et c'est pourquoi il existe des ponts pour mettre en œuvre des services Web. Une plateforme OSGi permet la communication entre les trois acteurs du SOC :

- **le fournisseur de services enregistre ses services auprès du registre** fourni par la plateforme OSGi. Le service est décrit par une interface JAVA.
- **le consommateur de services découvre les services dans le registre** et peut s'abonner à des notifications le prévenant de l'arrivée de nouveaux services. Le consommateur obtient une référence vers l'instance de service qu'il souhaite utiliser. Lors de la recherche de services dans ce registre, il est possible de filtrer les résultats au moyen d'un filtre LDAP sur les propriétés de service, un ensemble de services correspondants est alors proposé au client.
- **l'invocation est directe**. Le consommateur se sert de sa référence sur l'objet de service pour l'utiliser. Il n'y a donc pas de coût de communication contrairement aux services Web.

La relation forte avec JAVA est le prix à payer pour obtenir les mécanismes nécessaires aux dynamismes. Contrairement aux architectures à base de Web services, l'architecture OSGi privilégie donc le dynamisme par rapport à la gestion de l'hétérogénéité des applications. D'autre part l'invocation directe permet d'obtenir des performances très supérieures à celle des architectures Web.

Les services sont proposés par le code contenu dans les bundles. Les dépendances de services ne sont pas explicitement exprimées dans le bundle : elles peuvent se trouver

directement dans le code. Dans les approches à composants que nous présentons plus loin, un bundle peut héberger plusieurs composants fournissant et requérant des services. Ainsi les aspects de gestion de services et de déploiement du code sont relativement indépendants. La plateforme à services s'appuie sur ces mécanismes de déploiement : ce ne sont pas les services que l'on déploie mais du code implémentant des services.

Si WSDL est considéré par certains comme pauvre, la description des services sous OSGi l'est encore plus. Une spécification de service OSGi est composée :

- **d'une interface écrite en JAVA** qui fixe la syntaxe du service. Elle est standardisée entre les différents fournisseurs du même service.
- **d'un ensemble de propriétés du service**. Elles définissent les caractéristiques fonctionnelles et non fonctionnelles du service comme sa configuration ou la qualité de services.

Dynamisme

Bien qu'OSGi permette la disponibilité dynamique, jusqu'à la troisième version de la spécification (R3) la plateforme ne proposait pas de mécanisme pour gérer cette disponibilité de façon simple. Les clients avaient à charge d'écouter les notifications et de trouver un service remplaçant si nécessaire. Cette gestion devient rapidement complexe lorsque le nombre de services requis par un composant s'accroît : c'est source d'un nombre important d'erreurs. La gestion des appels directs nécessite le support par le langage de programmation d'opérations complexes permettant l'ajout et le retrait de portions de code à l'exécution. Les développeurs de services doivent prendre en compte ces spécificités car il peut se poser des problèmes de déchargement de classes si des références sont tenues. Ainsi il est parfois difficile d'implémenter correctement les services et la gestion des dépendances. L'automatisation de cette gestion a fait l'objet de nombreux travaux ([101],[130],[131]) et la dernière spécification (quatrième R4[129]) offre un mécanisme : Service Component Runtime (SCR) qui fournit un modèle à composants orienté service. Le modèle à composants iPOJO, que nous présentons après, propose une solution à ce problème.

Synthèse

OSGi est donc une architecture à service très différente de l'architecture à services Web car elle ne vise pas les mêmes objectifs. Les plateformes OSGi mettent l'accent sur le dynamisme et les performances, mais la relation forte avec le langage JAVA et la centralisation sont un frein à la gestion de l'hétérogénéité. OSGi n'arrive donc pas à faire totalement abstraction des aspects techniques. D'autre part, l'architecture à services OSGI ne propose pas par défaut de solutions simples pour gérer les aspects non fonctionnels. La conception d'applications à services directement au-dessus d'une plateforme d'OSGi peut s'avérer ardue car elle demande une bonne connaissance des mécanismes d'OSGi et de JAVA. C'est pourquoi elle sert de base à des architectures à services plus complètes comme iPOJO.

4 LES APPROCHES A COMPOSANTS ORIENTES SERVICES

Les deux approches exposées présentent des défauts. Un des manques importants de l'architecture à services Web est de ne pas proposer de mécanismes permettant la gestion du cycle de vie des applications. Cette architecture souffre également de problèmes de performances.

A contrario, si OSGi propose ces mécanismes et offre de bonnes performances, les services OSGi se distribuent mal. La programmation OSGi est extrêmement difficile à maitriser car il faut gérer le dynamisme et le cycle de vie des services. Le code fonctionnel du service et la gestion du dynamisme sont indifférenciés lors de l'implémentation, ce qui rend le code difficile à maintenir. La gestion du dynamisme rend périlleuse la conception d'applications comprenant de nombreux services. En effet, pour chaque liaison entre services, l'utilisateur doit être capable de réagir aux apparitions, modifications et disparitions de services. Chaque fournisseur de service est tenu de s'enregistrer auprès de l'annuaire, déclarer ses modifications et notifier de son départ. L'ensemble de ces actions sont répétitives et représentent rapidement un volume de code important pour peu qu'un composant propose de multiples services et en dépende. La gestion de la mémoire est un aspect important dans OSGi. Lorsqu'un composant disparait, l'objet reste en mémoire tant que les autres composants détiennent une référence vers cet objet ; c'est un choix technique des concepteurs de JAVA. Souvent les utilisateurs de services oublient de relâcher ses références car ne pas le faire n'a pas de conséquence directement observable. C'est une source d'erreurs difficiles à identifier et qui peuvent dégrader considérablement les performances de l'application et de la plateforme. C'est une des raisons de l'apparition d'approches alternatives telles que les modèles à composants orientés service.

4.1 COMPOSANT ET MODELE A COMPOSANTS

La programmation à base de composants est apparue avant l'approche à service. Elle préconise la construction des applications par la composition de blocs de code réutilisables appelés composants. L'approche à composants est dans la lignée des approches modulaires et des approches à objets. Elle propose de découper les applications en unités fonctionnelles cohérentes de taille raisonnable pour faciliter le développement. L'application résulte de l'assemblage de ces unités de code.

Il n'y a pas actuellement de consensus sur la notion de composant [132]. La définition généralement retenue est celle de Szyperski :

- *« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties ».*

De cette définition et [101][133], on retient les caractéristiques suivantes :

- un composant propose des fonctionnalités accessibles par l'intermédiaire **d'interfaces.** Un composant s'utilise conformément à un usage officiel qui permet aux développeurs de logiciels clients de l'employer.
- **un composant n'est pas lié à un ensemble prédéterminé de clients**.
- **un composant peut être utilisé par des tiers** : l'approche à composants distingue bien la phase de développement de la phase d'assemblage. Le développement et l'assemblage peuvent être effectués par des personnes différentes.
- un composant est une unité de code **réutilisable**.
- un composant a des **dépendances explicites** : lorsqu'un composant dépend des fonctionnalités d'autres composants, il le déclare explicitement.
- **un type de composant peut être instancié** : l'approche à composants fait la distinction entre les types et les instances de composants. C'est la même

distinction qu'entre classe et instance d'objet. Dans la majorité des modèles à composants, les composants sont des types qui peuvent exister sous la forme de plusieurs instances de configuration différentes.

- **en règle générale, un composant est une unité de déploiement**. Toutefois avec certains modèles à composants reposant sur OSGi comme iPOJO, l'unité de déploiement contient plusieurs composants.

Une bonne décomposition aboutit généralement à un couplage faible entre les composants ; la granularité plus forte des composants, si on les compare à l'objet, rend la réutilisation plus crédible au sein des entreprises et même entre les entreprises. Le recours à des approches à composants permet donc de faciliter le développement, d'augmenter la réutilisabilité et la maintenabilité des applications, et de faciliter leur évolution.

Pour organiser la réalisation d'applications à base de composants, les développeurs ont recours à un modèle à composants. Une définition communément admise de modèle à composant est la suivante :

- *« A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model. »[134].*

Ces modèles proposent généralement un langage de description d'architecture (voir [135] pour des exemples) et parfois un langage de description d'interface ou IDL[13]. Il existe aujourd'hui de nombreux modèles à composants [136] tels que Fractal[137], EJB[138] ou par exemple COM[139].

Un aspect particulièrement intéressant de certains modèles à composant est qu'ils fournissent des mécanismes permettant la séparation de la logique métier des aspects orthogonaux. Ils introduisent la notion de conteneur (illustré figure 28).

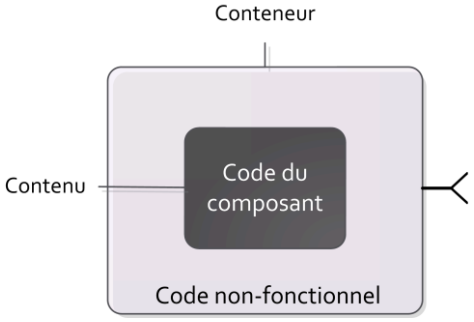

Figure 28 : Composant et son conteneur

Un conteneur est une coquille qui entoure le contenu - le code métier – de l'instance, il prend en charge les préoccupations non fonctionnelles telles que l'administration du composant ou la gestion des interactions avec les autres instances. L'objectif n'est pas ici de présenter tous les concepts des modèles à composants, une comparaison de nombreux modèles à composants peut se trouver dans [101] et [132].

[13] Acronyme de *Interface Description Language (voir aussi la définition de l'OMG sur http://www.omg.org/gettingstarted/omg_idl.htm)*

4.2 ORIENTATION SERVICE DES MODELES A COMPOSANTS

Les modèles à composants traditionnels souffrent de limitations. En général les applications sont assemblées durant la phase de conception et les liaisons entre composants s'effectuent tôt dans le cycle de vie de l'application. La composition se fait donc sur la base d'un ensemble de types prédéfinis et connus pendant la phase de développement. De fait, cette définition précoce constitue un frein à l'évolution de l'application pendant l'exécution. Lorsqu'il est possible de modifier les relations entre les composants, il n'est plus possible d'ajouter de nouveaux types de composants s'ils n'ont pas été prévus à la conception. Ainsi, les approches à composant ne prennent souvent pas en compte le dynamisme lié à l'ajout ou au retrait de types de composants.

La proposition des modèles à composants orientés service consiste à utiliser les concepts du SOC dans les modèles à composants. Il existe aujourd'hui plusieurs implémentations de ces modèles comme iPOJO [140], Spring[14] ou SCA. Pour les auteurs de [101] et [130], un modèle à composants orienté service possède les caractéristiques suivantes :

- **un service est une fonctionnalité fournie**. Un service fait référence à une ou plusieurs opérations réutilisables.
- **un service est caractérisé par une spécification de services** (contrat) qui contient des informations syntaxiques et parfois des informations sémantiques et comportementales. Il décrit également les dépendances sur les autres spécifications de services.
- **les spécifications sont implémentées par les composants.** Un composant peut implémenter plusieurs spécifications de services, il devient alors pourvoyeur de plusieurs services. De la même façon, une spécification peut être implémentée par plusieurs composants. Il existe alors plusieurs fournisseurs du même type de service. Le composant respecte le contrat de service défini dans la spécification qu'il implémente et décrit les contraintes liées à l'implémentation du service. Il peut dépendre d'autres spécifications de services.
- **le modèle d'interaction de l'approche à services est utilisé pour résoudre les dépendances de services.** Les services sont enregistrés par les instances auprès d'un mécanisme de découverte. Lorsqu'une instance de composant dépend d'une spécification, elle peut découvrir l'ensemble des instances fournissant cette spécification dans l'annuaire.
- **les compositions de services sont décrites en termes de spécification de services**. Une composition est constituée d'un ensemble de services. Elle ne fait pas directement référence aux aspects technologiques. Un mécanisme de sélection se charge de rendre cette composition concrète en choisissant les composants respectant les spécifications de services dont elle dépend.
- **la substituabilité repose sur les spécifications de services**. Lors d'une composition, un composant respectant une spécification de service peut être remplacé par une implémentation alternative la respectant également.

Lors de la réalisation d'applications basées sur ces modèles, les développeurs cherchent donc à minimiser les dépendances de code au profit des dépendances de services. Les ADL fournis par les modèles à composants orientés services permettent d'exprimer les dépendances, non plus seulement en termes de code, mais en termes de spécifications de services, ce qui augmente la substituabilité des composants. Les compositions ne sont plus exprimées en termes d'implémentation.

[14] Spring : http://www.springsource.com/

Les modèles à composants à services organisent donc la séparation entre les besoins liés à l'implémentation du service (les dépendances de code) et ceux liés à l'assemblage de ces composants (dépendance fonctionnelle). Les applications sont ainsi construites à un niveau d'abstraction plus élevé, sur la base des spécifications de services. Les aspects techniques et les dépendances d'implémentations sont relayés à la phase de développement des implémentations de services.

4.3 SERVICE COMPONENT ARCHITECTURE (SCA)

Service Component Architecture (SCA) est un ensemble de spécifications définies par le consortium OSOA[15] composé de grands groupes industriels dont IBM, Oracle ou SAP. SCA introduit un modèle à composants pour faciliter la conception des services. Il existe aujourd'hui plusieurs implémentations de SCA telles que Tuscani[141] ou Aqualogic[142].

SCA se veut très indépendant des technologies utilisées pour l'implémentation des services et des protocoles assurant leur communication (SOAP, RMI). Il est beaucoup utilisé avec les services Web mais il est capable de marier des services écrits en JAVA, .NET, PHP ou COBOL par exemple. L'un des grands avantages de ce modèle est de fournir une représentation des services d'un niveau d'abstraction élevé ; l'assembleur n'a pas à se soucier des contingences techniques. SCA apporte beaucoup de flexibilité en permettant de construire des applications constituées de services de technologies très hétérogènes. Cet aspect est très important pour les entreprises lorsqu'elles chercheront à intégrer les services patrimoniaux avec des services codés dans une nouvelle technologie. SCA fournit un cadre pour la construction de services et un modèle d'assemblage qui permet la création d'applications en liant ces services.

Le modèle SCA permet la conception des applications via une composition structurelle des services ; contrairement à BPEL, qui repose sur un moteur d'orchestration, les invocations se font point à point entre les services. L'élément de composition de base de SCA est le composant. Dans SCA les compositions sont exprimées en termes de types de composant. Un type de composant peut avoir plusieurs implémentations, pourvu que celles-ci respectent sa description. Un composant SCA (figure 29) est décrit dans un ensemble de fichiers XML standardisés. La description définit :

- les **services** proposés par le composant. Le composant expose les fonctionnalités qu'il propose sous forme de service. La description du service peut être réalisée avec des technologies très différentes. Un service JAVA par exemple peut exposer une interface JAVA, tandis qu'un service Web utilisera WSDL.
- les **références** aux services utilisés. Ce sont les services dont dépend le composant pour fonctionner.
- les **propriétés** permettant la configuration du composant. Ces propriétés influencent le comportement du composant.

SCA définit un langage de composition SCDL qui permet d'assurer l'assemblage des services pour former l'application. Ce langage permet la définition de composant composite, c'est-à-dire issu de la composition d'autres composants comme le montre la figure 29. Les composites sont des unités d'encapsulation qui permettent de définir les visibilités et de masquer les informations superflues à l'assembleur. Dans le modèle SCA une application est donc un composite particulier.

Un composite SCA peut être issu de la composition de services hétérogènes. Comme tout autre composant, il peut requérir et exposer des services et possède des propriétés de configuration. Les composants d'un composite n'ont pas accès aux composites par des composants extérieurs, et réciproquement, sans que cela soit explicitement prévu. Pour cela, il existe une

[15] Open Service Oriented Architecture

correspondance entre les services, références et propriétés des services contenus et ceux du composite – c'est le concept de promotion. La force de ces compositions hiérarchiques structurelles est de rendre possible la conception d'applications composées d'un grand nombre de services. Elles permettent à l'assembleur de se focaliser uniquement sur des concepts de haut-niveau.

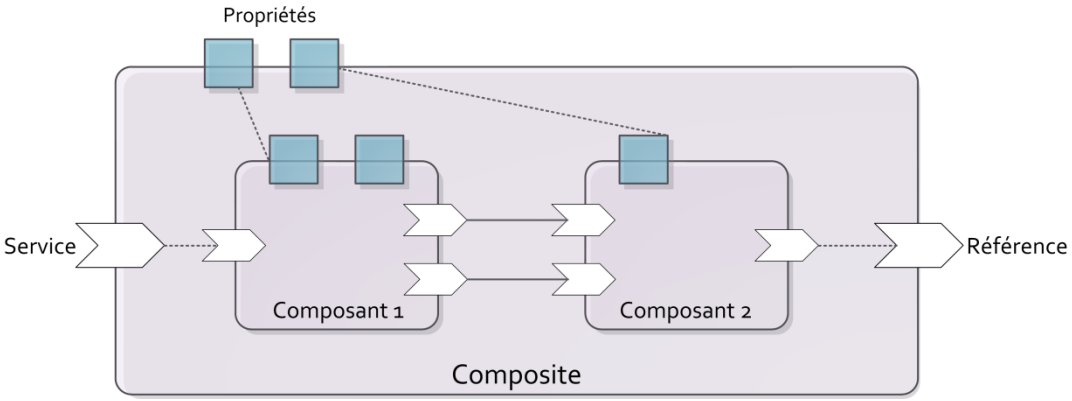

Figure 29 : Représentation d'un composite SCA

En conclusion, SCA est un modèle qui apporte beaucoup de souplesse lors de la création d'applications à base de services. Il facilite la réutilisation de composants en permettant l'assemblage d'applications sans avoir à se soucier des aspects technologiques. Il s'adapte particulièrement bien aux environnements à services hétérogènes. SCA est donc essentiellement focalisé sur l'interopérabilité. L'un des reproches dirigé à son encontre est qu'il ne spécifie rien sur le dynamisme des applications. En particulier, pour bon nombre d'implémentations de SCA, une fois effectuées les liaisons entre composants ne peuvent plus être modifiées, ce qui empêche la substitution d'un fournisseur par un autre.

4.4 IPOJO

iPOJO ([114] [140]) est un autre modèle à composants implémenté sur OSGi. Il est le successeur d'autres modèles à composants tels que SCR (Service Component Runtime), qui se focalise essentiellement sur la résolution automatique des dépendances. Le modèle iPOJO est intégré au projet Apache Felix, une implémentation de la spécification OSGi.

iPOJO combine l'approche à services et la programmation par composants afin d'introduire des caractéristiques de dynamisme au sein des modèles à composants. Il s'appuie sur les concepts de spécification de service, propriété, type de composant, instance, conteneur, et propose un langage de composition permettant la création d'applications nativement dynamiques. Comme SCA, iPOJO propose deux mécanismes : l'un permettant la construction des composants à services et l'autre permettant de décrire la composition des services pour former une application. L'une d'une des particularités d'IPOJO est d'offrir en outre des mécanismes d'introspection permettant la gestion des aspects orthogonaux des services.

iPOJO applique le principe de séparation des préoccupations pour dissocier les aspects orthogonaux du code applicatif métier. Un composant iPOJO, déployable comme tout bundle classique sur la plateforme OSGi, est décrit par deux parties (figure 30) :

- le code applicatif qui est l'implémentation des services fournis par le composant. C'est un ensemble appelé POJO[16] de classes et interfaces java.
- une description (XML ou annotation) qui décrit les aspects non fonctionnels tels que les dépendances de services, les services fournis, les propriétés de services. C'est la description du conteneur.

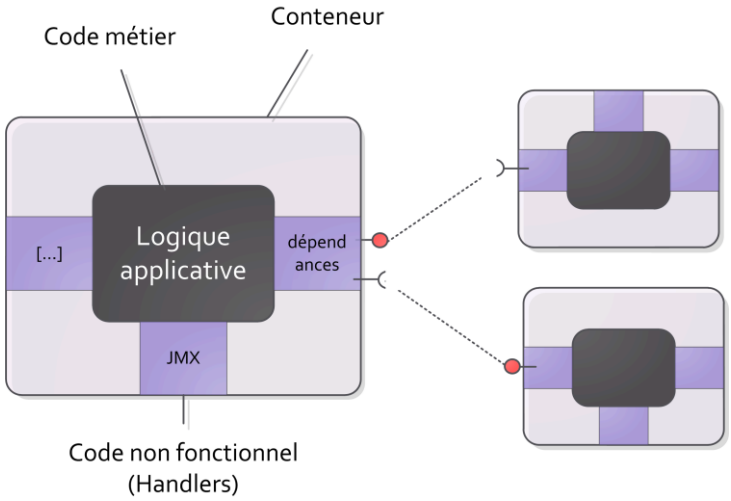

Figure 30 : Conteneur iPOJO avec ses handlers

Les aspects orthogonaux sont gérés par un conteneur. Celui-ci est composé d'un ensemble d'unités spécialisées dans le traitement d'un aspect particulier : les handlers. Par défaut, iPOJO est accompagné d'handlers permettant entre autre la gestion des dépendances, du cycle de vie, la reconfiguration des propriétés du composant. Ces handlers exposent ou requièrent des services comme le montre la figure 30. Il est possible de développer de nouveaux handlers qui pourront alors être utilisés avec tout composant iPOJO.

Le couple OSGi/iPOJO possède les mêmes avantages que ceux décrit pour OSGi. Il rajoute un support efficace de la disponibilité dynamique et une séparation plus nette entre les aspects métiers et les aspects orthogonaux.

Le modèle iPOJO simplifie le développement et la maintenance des composants de l'application avec la notion de conteneur. Il fournit de nombreuses briques réutilisables pour la réalisation de la membrane. Mais l'avantage principal d'iPOJO est la prise en charge du dynamisme des applications. Il permet l'observation du comportement de l'application ainsi que la modification de sa configuration et de sa topologie en fournissant les primitives permettant la reconfiguration de la membrane et des dépendances. Cette caractéristique est particulièrement intéressante dans le cadre de l'informatique autonomique car elle offre de nombreuses possibilités d'adaptation des applications.

[16] POJO pour Plain Old Java Object qui signifie l'utilisation de simples objets (ie. non EJB < version 3 par exemple).

5 SYNTHESE

Nous avons présenté dans ce chapitre l'approche à services. Notre objectif était de donner une vision générale de l'approche orientée service. Le service est au cœur de ce nouveau paradigme de programmation. C'est une entité logicielle d'un niveau d'abstraction élevé qui donne la possibilité au concepteur de se focaliser sur l'essentiel, la fonctionnalité fournie, et de faire abstraction des contingences technologiques. Cette fonctionnalité est décrite par une spécification qui permet la mise en relation des pourvoyeurs avec les consommateurs.

La principale caractéristique de l'approche à services est le très faible couplage entre les entités composant de l'application. Ce couplage faible provient de la présence d'un mécanisme de découverte, souvent un registre. Les pourvoyeurs de services publient leur service dans cet annuaire, donnant ainsi la possibilité aux consommateurs de les découvrir. Les détails techniques de l'implantation du service tels que la technologie d'implantation et la plate-forme d'exécution sont masqués à l'utilisateur du service. Le fournisseur du service ne connaît pas a priori ses utilisateur, ni son contexte d'utilisation.

La gestion de l'hétérogénéité et le dynamisme résultent du faible couplage. Il faut cependant souligner que c'est le premier aspect qui a fait notamment la popularité des Web Services et du SOA. Au demeurant, c'est cet aspect qui est fréquemment considéré dans le domaine de l'informatique autonomique, comme par exemple [60].

Les implémentations de plateformes à services possèdent des propriétés très différentes. Nous avons donné l'exemple de plusieurs technologies représentatives. Les services Web excellent dans la gestion de l'hétérogénéité mais offrent souvent des performances modestes qui résultent des coûts en communication. Le dynamisme et la découverte sont rarement mis en avant et utilisés en dépit de normes existantes comme UDDI. Les plateformes OSGi très dynamiques et performantes sont encore trop contraintes par la technologie d'implémentation (Java), ce qui complique la distribution. Cependant on assiste aujourd'hui à une convergence avec le développement d'outils permettant la communication entre les différents types de plateformes [107].

Jusqu'à présent, l'essentiel de l'attention s'est portée sur l'utilisation des concepts de l'informatique autonomique aux applications à services (voir par exemple [143]). Nous proposons de faire la démarche complémentaire : profiter des propriétés des approches à services pour concevoir des gestionnaires autonomiques. Nous nous concentrons donc sur les aspects dynamiques. Le dynamisme nous apparaît en effet comme l'une des propriétés les plus intéressantes des approches à services : il rend possible la mise à jour des applications sans les interrompre, la proposition de services à la demande et ainsi l'amélioration notable du confort d'utilisation.

Les modèles à composants tels qu'iPOJO offrent des outils performants permettant de surveiller et de modifier la topologie des applications. iPOJO a déjà été utilisé avec succès pour l'implémentation des sondes et des effecteurs [11]. Nous verrons que nous l'utilisons pour l'implémentation de notre canevas mais cette fois pour la conception du gestionnaire lui-même, bénéficiant ainsi du dynamisme apporté par l'approche à services pour faire évoluer la configuration et la topologie du gestionnaire au cours du temps.

Chapitre 4 - **Proposition**

Dans la première partie de cette thèse, nous avons dressé un état de l'art sur deux approches informatiques récentes et qui nous semblent complémentaires : l'informatique autonomique et les approches à services. Lors de la présentation de l'informatique autonomique nous nous sommes attachés à montrer comment les boucles de contrôles sont aujourd'hui construites. Scientifiques et ingénieurs s'accordent sur une construction séparée du gestionnaire autonomique. Nous avons également montré les limites des approches actuelles qui pêchent souvent par manque de modularité, d'homogénéité et de flexibilité.

La seconde partie de notre document traite de notre proposition qui introduit un cadriciel, ou framework, original pour supporter la construction de boucles autonomiques modulaires et flexibles utilisant les approches à services.

Dans le premier chapitre de cette seconde partie, nous rappelons le contexte de notre travail et présentons une vision globale de notre approche. Celle-ci s'articule autour du concept de tâche, brique élémentaire fortement spécialisée et cohérente qui, dans notre vision, sert de base à la construction de la partie décisionnelle des systèmes autonomiques. Nous expliquons ensuite comment les boucles de contrôle peuvent être réalisées avec ce nouveau concept et l'architecture du cadriciel que nous proposons.

1 PROBLEMATIQUE

Nous avons présenté, dans la première partie de ce document, la notion d'informatique autonomique. Nous avons également développé un état de l'art qui nous a permis d'établir des constats de différents ordres. En particulier, nous nous sommes rendu compte de l'importance prise aujourd'hui par le domaine de l'informatique autonomique et du défi majeur posé par la conception de systèmes autonomiques qui doivent posséder des propriétés difficiles à garantir.

Plus précisément, nous avons fait les constatations suivantes :

- **les solutions autonomiques représentent un enjeu majeur pour l'informatique moderne.** En effet, la faisabilité même des logiciels dans de nombreux domaines repose sur l'existence de capacités autonomiques pour faire face à la complexité et au dynamisme des applications. Dans ces différents domaines, tels que l'informatique autonomique, il n'est simplement plus possible de laisser la charge entière de l'administration de bas niveau aux ingénieurs de maintenance.
- **les systèmes autonomiques sont, par nature, plus complexes que les systèmes traditionnels.** Cela découle de l'objectif principal des systèmes autonomiques, à savoir absorber la complexité des tâches administratives de bas niveau en fournissant des interfaces plus abstraites, simplifiées et intuitives à l'administrateur humain. Mais, de fait, la complexité ne disparait pas, elle se déplace de l'administration à l'application, des administrateurs aux développeurs. Sans surprise, la construction de systèmes autogérés s'avère donc une entreprise difficile et coûteuse. Il est nécessaire aujourd'hui de fournir des approches et des technologies permettant de définir aisément des comportements autonomiques.
- **pour autant, les systèmes autonomiques sont soumis aux mêmes contingences que les autres systèmes informatiques.** Ils doivent à la fois atteindre des qualités logiciels telles que la performance et la disponibilité, et subir les contraintes des développements logicielles actuels, à savoir les courts délais de mise sur le marché, l'utilisation obligatoire de composants hétérogènes produits par des fournisseurs divers, l'utilisation massive des réseaux, etc.
- **les solutions autonomiques doivent être capables de s'adapter à des configurations variées.** Cela signifie que les fonctions administratives, elles-mêmes, doivent pouvoir détecter des changements dans l'environnement et s'adapter à ces évolutions. Différents types d'analyse ou de réaction peuvent ainsi être mis en œuvre au cours du temps et des évolutions. Cela peut conduire à un nombre conséquent de contextes possibles nécessitant la mise en œuvre de comportements différents de la part du gestionnaire. Etablir une description exhaustive de l'ensemble des comportements d'un gestionnaire autonomique est une entreprise ardue. En fait, on est amené à faire de l'auto-gestion de la fonction autonomique.
- **enfin, l'informatique autonomique est un domaine à part entière, même si elle est à la croisée de nombreux domaines scientifiques.** Elle doit développer ses techniques et ses méthodes propres pour répondre aux défis qui lui sont posés. C'est le cas en particulier au niveau des architectures : des concepts propres et des organisations appropriées doivent être définis.

Tout au long de l'état de l'art, nous avons porté une attention toute particulière aux aspects architecturaux. Nous avons ainsi présenté en détail les propositions architecturales d'IBM et étudié la structuration des systèmes existants les plus marquants. A ce niveau, nous avons effectué les constatations suivantes :

- **l'architecture définie par IBM est largement acceptée par la communauté autonomique.** Cette architecture, connue sous l'acronyme MAPE-K, définit les fonctions essentielles permettant de mettre en œuvre un système autonomique, à savoir les fonctions de collecte, d'analyse, de planification, et d'exécution. Elle structure également ces fonctions en une boucle de rétroaction. Il faut cependant bien comprendre que MAPE-K est une architecture logique, de haut niveau d'abstraction. Ce n'est pas une architecture d'implantation.
- **l'architecture de nombreux systèmes existants se calque complètement sur MAPE-K.** Malheureusement, l'application trop littérale des principes de cette architecture pour implanter un gestionnaire autonomique produit souvent des gestionnaires mal équilibrés, peu performants, difficiles à maintenir et à étendre. En effet, dans certains cas, des blocs fonctionnels sont simplistes et ne devraient pas jouer un rôle architectural de premier plan. Par exemple, une analyse peut être constituée d'une seule règle ECA. Dans d'autres cas, ils sont trop compliqués et mélangent des préoccupations. Par exemple, plusieurs analyses peuvent se trouver mélangées au sein d'un même composant ou d'un même ensemble de règles.
- **les différentes préoccupations, les priorités et les objectifs des gestionnaires (les propriétés auto-*) sont mêlés les uns aux autres sans séparation claire.** Peu de projets offrent aujourd'hui la possibilité d'implémenter ces préoccupations indépendamment, et lorsque c'est le cas ils ne proposent pas de solution pour gérer les conflits éventuels entre ces différents objectifs. Cela nuit à la réutilisabilité des algorithmes, à la compréhension du gestionnaire, à l'extensibilité et la maintenabilité.
- **les gestionnaires autonomiques actuels sont difficiles à étendre.** En effet, les gestionnaires sont rarement développés à partir de techniques de génie logiciel. Il en résulte souvent des fonctions codées en dur, difficiles à étendre. En particulier, les parties « raisonnement » des gestionnaires sont souvent implantées à l'aide de règles ECA. Or, dans bien des cas, celles-ci présentent une granularité trop faible et rendent les modifications délicates. L'usage de règles a toujours été un problème pour le passage à l'échelle.
- **les architectures et les méthodes utilisées sont généralement peu réutilisables.** Un grand nombre de fonctionnalités étudiées et implémentées dans les solutions actuelles pourraient être réutilisées dans différentes solutions. Actuellement, ces fonctionnalités communes sont en pratique implémentées individuellement et répétitivement pour les différents systèmes. C'est non seulement une perte de temps mais également un risque d'introduction d'erreurs important. Compte-tenu de la complexité fonctionnelle et des exigences de fiabilité des applications autonomiques, fournir des solutions génériques et réutilisables apparait comme essentiel pour faciliter leur développement.

2 OBJECTIFS

Au regard des limitations précédemment mentionnées, l'objectif de notre travail est de définir et d'implanter un cadriciel, ou framework, facilitant le développement de gestionnaires autonomiques dans des situations et domaines variés. Dans ce but, nous visons à définir un modèle architectural permettant le développement de gestionnaires autonomiques :

- **modulaires**. Il convient de définir les différentes fonctions autonomiques d'un gestionnaire de façon modulaire, suivant certains principes bien éprouvés du génie logiciel. Un framework doit ainsi offrir des mécanismes pour la définition séparée de ces fonctions et pour leur contrôle.
- **homogènes**. Il est important de disposer d'un cadre architectural homogène et de ne pas multiplier inutilement les concepts. Cela permet une utilisation naturelle et efficace d'un framework.
- **souples**. Il est important de laisser une certaine liberté à l'utilisateur du framework en ce qui concerne le découpage architectural du gestionnaire. Il est en effet maladroit, à notre avis, de s'enfermer dans un cadre trop strict. Cela ne permet pas, en effet, d'aborder l'ensemble des domaines et situations souhaités. Pour autant, le framework doit permettre l'implantation naturelle des architectures de référence, comme celle d'IBM.
- **évolutifs**. Comme toute application informatique, un gestionnaire autonomique est amené à évoluer. Il faut dès lors fournir des mécanismes permettant la modification des fonctions autonomiques et de leur contrôle. Il s'agit, dans ce dernier cas, de faire évoluer les politiques d'ajustement du logiciel contrôlé.
- **dynamiques**. Certaines évolutions doivent pouvoir se faire de façon dynamique, c'est-à-dire en maintenant un certain niveau de fonctionnalité. Ainsi, le logiciel contrôlé est modifié sans être arrêté. Mieux, certaines évolutions peuvent être gérées par le framework lui-même de façon autonomique durant son exécution.
- **administrables**. On doit pouvoir clairement visualiser la configuration du framework et agir dessus en maîtrisant les effets. Un gestionnaire autonomique est lui aussi confronté à des problèmes importants d'administration. Ceci est bien entendu une condition préalable à l'évolutivité du framework.

Comme nous l'avons déjà indiqué, ces propriétés correspondent à celles que l'on attend de logiciels modernes. Les propriétés d'adaptabilité et de dynamisme sont en particulier des atouts pour faire face à l'hétérogénéité et au dynamisme des applications modernes, en particulier lorsque l'ensemble des contextes d'utilisation sont mal définis ou imprévisibles. Pour atteindre ces objectifs, nous comptons nous appuyer sur les principes et outils définis par le génie logiciel. Plus précisément, et comme le suggère l'état de l'art développé précédemment, nous avons choisi d'utiliser l'approche à composants orientée service. Celle-ci fournit, en effet, de façon native les principes d'abstraction, de modularité, de séparation des préoccupations et de dynamisme.

Un objectif important de ce travail est de clairement définir et de séparer les concepts appartenant de façon générique à un gestionnaire autonomique et les aspects métier, développés au cas par cas. Le but est de permettre aux experts domaine de se concentrer sur l'écriture des fonctions autonomiques et de ne pas gérer entièrement l'enchaînement et le contrôle de ces fonctions.

3 NOTRE APPROCHE

3.1 PRINCIPES

Pour atteindre nos objectifs, nous avons tout d'abord défini la notion de **tâche d'administration**. Une tâche d'administration est une entité indépendante et spécialisée qui réalise une ou plusieurs fonctions d'administration. Il peut s'agir par exemple d'une fonction d'acquisition de données, d'une fonction d'analyse ou bien d'une fonction regroupant analyse et planification. Dans notre approche, un gestionnaire autonomique est constitué d'un ensemble de tâches d'administration et une boucle de contrôle correspond à un chemin parmi l'ensemble de tâches disponibles (figure 31). En fonction des ressources, un gestionnaire comportera un nombre plus ou moins important de telles tâches. Notre approche est à la fois plus souple et plus précise que le modèle MAPE-K d'IBM. Plus souple car une tâche d'administration peut inclure différents types de fonctions d'administration. Plus précise car, comme nous le verrons, les interfaces des tâches sont très précisément définies.

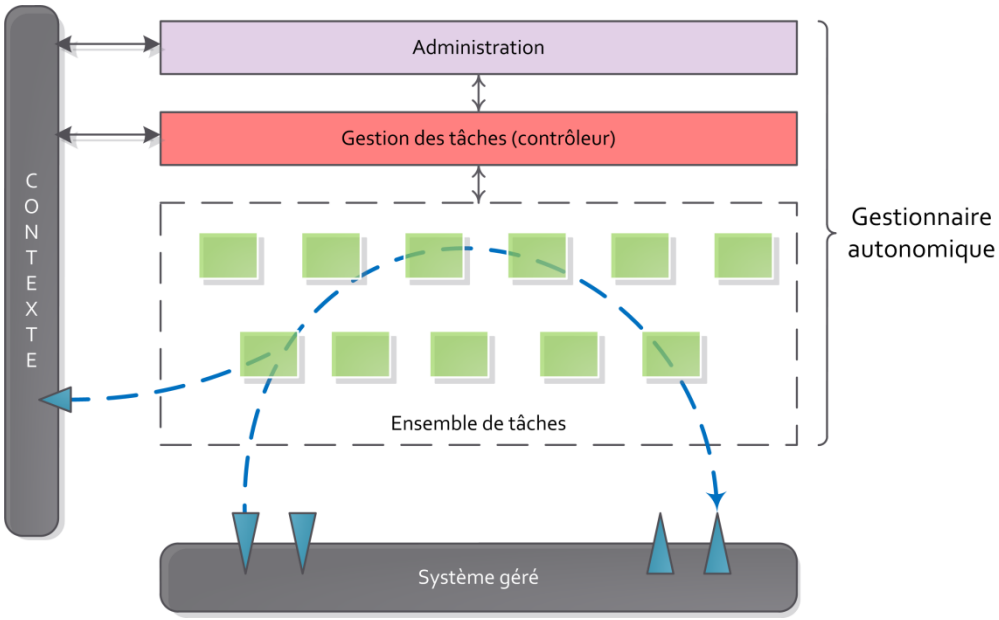

Figure 31 : Vision d'ensemble de nôtre approche

Nous avons ensuite défini une architecture de gestion de ces tâches basée sur deux fondamentaux:

- l'**utilisation d'événements** pour la communication entre les tâches. Cela permet une liaison asynchrone entre les différentes tâches qui ne se connaissent pas. Les résultats des tâches sont transmis sous forme de messages aux tâches concernées, qui les consomment à leur rythme.
- la définition d'un **contrôleur autonomique** chargé de la gestion des tâches d'administration. Il s'appuie sur le contexte d'exécution courant et sur des politiques d'administration pour assurer la gestion des tâches. Cela inclut l'activation opportuniste des tâches, ainsi que l'ajout ou la suppression dynamique de tâches. Par ailleurs, le contrôleur est doté de mécanismes d'observation et d'analyse qui lui permettent de modifier dynamiquement et automatiquement les politiques d'agencement et la configuration des tâches.

Cette architecture a été implantée sous la forme d'un framework fondé sur la technologie des composants orientés service. Cette dernière apporte de façon native les propriétés de

modularité et de dynamisme. Le framework que nous avons développé fournit un cadre pour l'intégration dynamique de fonctions autonomiques et pour leur gestion en fonction du contexte et de politiques d'administration, elles-mêmes évolutives. La gestion comprend l'activation opportuniste des tâches.

Notre proposition répond aux exigences que nous nous étions fixées. Tout d'abord, la **modularité** est apportée par l'introduction de la notion de tâche d'administration. La tâche est une entité logicielle complexe qui encapsule une ou plusieurs fonctions d'administration. Cette notion facilite le développement des gestionnaires autonomiques en permettant aux développeurs de se concentrer sur les fonctions d'administration sans se soucier des contingences informatiques liées à la communication, à la persistance, etc. Elle permet également la réutilisation puisque la même fonction administrative, une analyse par exemple, peut être utilisée dans différents contextes. De fait, la forte spécialisation des tâches encourage leur réutilisabilité. Ainsi, cette approche permet de concevoir des gestionnaires en intégrant des fonctionnalités existantes. Remarquons enfin qu'une bonne modularité permet de mieux appréhender l'utilité de chaque bloc architectural et donc d'analyser et de modifier plus efficacement le fonctionnement d'un gestionnaire autonomique. En définitive, la mise en œuvre de cette vision devrait simplifier la réalisation des gestionnaires.

Par ailleurs, notre approche propose un modèle **homogène** pour l'intégration des fonctionnalités autonomiques. Elle repose sur la composition opportuniste et dynamique de modules spécialisés faiblement couplés, les tâches. Elles sont toutes construites sur le même modèle et obéissent aux mêmes API. Le même concept est ainsi utilisé à chaque étape de la boucle de contrôle autonomique.

La **flexibilité** de notre proposition intervient à deux niveaux. A la conception d'une part, en apportant de la souplesse dans l'implémentation des fonctionnalités autonomiques, de leur réutilisation et de leur assemblage. A l'exécution ensuite, en permettant la modification des politiques d'agencement et la reconfiguration de ces fonctionnalités. Cette opération est soit laissée à l'initiative d'un administrateur, soit effectuée automatiquement. Le comportement du gestionnaire est dynamique : non seulement configurable mais également extensible. En outre, pour renforcer le dynamisme et la flexibilité, l'expression de la combinatoire n'est plus statique. Il est possible d'adapter les relations entre les tâches administratives pour former des solutions initialement non prévues. Cette proposition permet ainsi d'implémenter les différents aspects du comportement du gestionnaire, focalisé sur la gestion de problèmes différents, en isolation pour favoriser leur réutilisation.

Enfin, le **dynamisme** est une caractéristique importante de notre proposition. En effet les relations entre tâches évoluent en fonction de l'environnement et des objectifs. Au cours de l'exécution, le gestionnaire crée les conditions d'une collaboration opportuniste entre ces tâches en fonction de la situation actuelle, les objectifs et les informations disponibles. Les tâches sont découvertes et assemblées dynamiquement pour la détection et la résolution de problèmes complexes, potentiellement inattendus. De cette manière, des stratégies de gestion complexes peuvent être développées grâce à l'intégration de composants simples. La composition est opportuniste dans le sens où elle s'effectue durant l'exécution, en fonction du contexte. Il n'est alors plus nécessaire de décrire statiquement de façon exhaustive l'ensemble des situations de collecte de données, d'analyse, de planification ou d'exécution car le système peut adapter durant l'exécution.

3.2 TACHES D'ADMINISTRATION

Les tâches réalisent une fonctionnalité administrative très spécialisée et simple. Elles possèdent une expertise sur une partie très limitée de la boucle de gestion comme, par exemple, collecter la valeur d'une variable donnée, détecter un problème particulier tel un dépassement de seuil, proposer une solution particulière à un problème ou modifier une variable d'environnement. Il est ainsi possible d'utiliser différentes tâches d'analyse employant chacune une technique différente de résolution de problème. Les tâches sont des entités fonctionnellement indépendantes les unes des autres. Ainsi, il n'est pas nécessaire pour une tâche de connaître le fonctionnement ni même la présence des autres tâches pour fonctionner.

La granularité d'une tâche n'est pas fixe : si elle se destine naturellement à assumer une fonctionnalité de la boucle MAPE, elle peut très bien réaliser plusieurs de ces fonctionnalités. La granularité plus faible des tâches en comparaison des blocs architecturaux généralement proposés apporte de la flexibilité. Une tâche est plus simple à implémenter et plus facilement réutilisable. Pour autant, les tâches offrent une granularité suffisamment élevée pour permettre d'appréhender et de modifier l'action du gestionnaire.

Les tâches s'engagent à fournir la fonctionnalité décrite dans leur type. Celui-ci décrit la fonctionnalité fournie par la tâche, les propriétés de configuration, les propriétés partagées et les informations utilisées et produites. Un type de tâche peut avoir plusieurs implémentations, utilisables et déployables. Une implémentation peut être instanciée plusieurs fois et peut donc s'exécuter plusieurs fois dans le système avec une configuration différente. Ceci autorise la création de tâches génériques – par exemple de détection de seuil – dont l'utilisation sera possible dans des contextes différents.

Occasionnellement, certaines tâches peuvent implémenter la totalité d'une boucle de contrôle : elles ont alors la capacité de collecter des informations pertinentes et d'agir sur le système géré. Toutefois, la majorité des tâches a besoin d'informations générées par d'autres tâches et l'activation des tâches se base sur des événements. En fonction de ces derniers la tâche détermine si elle doit se déclencher. Une tâche se déclenche lorsqu'elle possède l'ensemble des informations nécessaires à son fonctionnement ; elle peut alors à son tour générer des données. De ce point de vue, l'organisation des tâches est comparable à l'organisation des activités dans un workflow, à cette différence que dans les workflows les relations entre activités sont beaucoup plus statiques et déterministes. Chaque tâche décrit les types de données qu'elle génère et les types de données qu'elle fournit. Il est ainsi possible d'inférer l'ensemble des chemins possibles dans un ensemble de tâches.

Une tâche est constituée de deux parties (figure 32) :

- une **condition de déclenchement** qui évalue la pertinence des informations reçues et choisit quand déclencher la tâche en fonction de ces informations et d'informations contextuelles.
- une **fonction d'administration**. Cette dernière peut être réalisée de différentes façons, par exemple en Java ou à l'aide d'un moteur de règles. Du point de vue de l'utilisateur, la tâche est une boîte noire qui dissimule les détails de son implémentation.

Les tâches échangent des messages de deux natures :

- **des données** qui peuvent être de natures différentes : valeur d'un paramètre, détection d'un seuil, solution partielle. Chacune de ces données est typée ce qui permet d'assurer qu'elle sera correctement interprétée par les autres tâches. La nature des informations échangées au sein du système n'est pas figée et évolue au cours du temps et des besoins. Il n'y a pas de contrainte sur la façon dont les données sont représentées.

- **des messages d'administration** qui sont transmis aux mécanismes de gestion des tâches et servant à évaluer la configuration et la politique d'agencement des tâches. Par exemple, une tâche qui estime recevoir trop peu d'information peut demander que la fréquence de production soit augmentée.

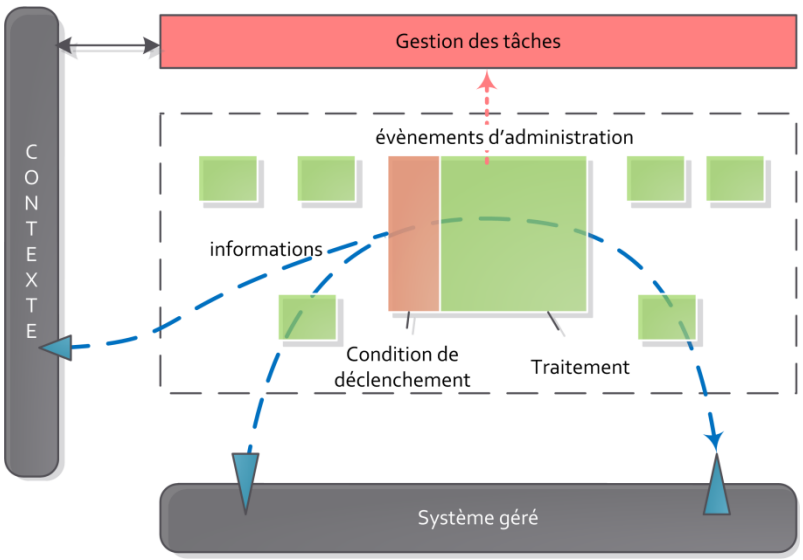

Figure 32 : Fonctionnement d'une tâche.

Bien que les tâches soient toutes construites sur les mêmes modèles, nous pouvons distinguer différents rôles : la collecte d'information, la transformation d'information, l'analyse des données, l'action sur la ressource. En particulier, la transformation d'informations est utile lorsque les tâches manipulent des contenus sémantiquement équivalents mais syntaxiquement différents. Dans ce cas, la tâche s'apparente à un médiateur.

Les tâches sont composites ou atomiques. Le passage à l'échelle est la raison essentielle des composites. Lorsque le nombre de tâches devient important ou qu'elles traitent des problèmes variés et complexes, le nombre de contributions risque d'engorger rapidement les moyens de communication. De même, la gestion des conflits entre les différentes tâches devient rapidement complexe. Le composite est une unité d'encapsulation permettant de former des groupes de tâches cohérents, soit parce qu'elles jouent le même rôle, soit parce qu'elles sont en conflit, soit parce qu'elles s'occupent d'un problème particulier. Chaque composite possède des moyens de communication et des mécanismes de sélection propres, ce qui répartit la charge de travail et facilite le passage à l'échelle.

3.3 ARCHITECTURE DE GESTION DES TACHES

Le framework fournit quatre mécanismes principaux pour permettre la configuration et l'organisation des tâches (figure 33), à savoir la gestion du cycle de vie des tâches, la gestion de la communication entre tâches, la gestion de conflits entre les tâches, et la gestion de la persistance.

Les mécanismes de gestion du cycle de vie fournissent les interfaces permettant d'ajouter, modifier, supprimer des tâches administratives. Il maintient un modèle représentant le système à l'exécution qui décrit la topographie, la configuration des tâches et fournit un ensemble de statistiques sur l'exécution des tâches. En particulier sont calculés le temps moyen d'exécution, le nombre et le type d'informations traités. Ces informations doivent permettre l'évaluation des tâches. Ce mécanisme est capable d'interpréter une description de politique d'agencement fournie par le gestionnaire de politique. Il s'appuie sur un dépôt pour déployer et installer de nouvelles tâches.

Le support de communication propose deux manières de réaliser la communication : soit via un intermédiaire qui distribue les messages (bus à messages), soit via un mécanisme de découverte qui met en relation directe fournisseur et consommateur d'informations.

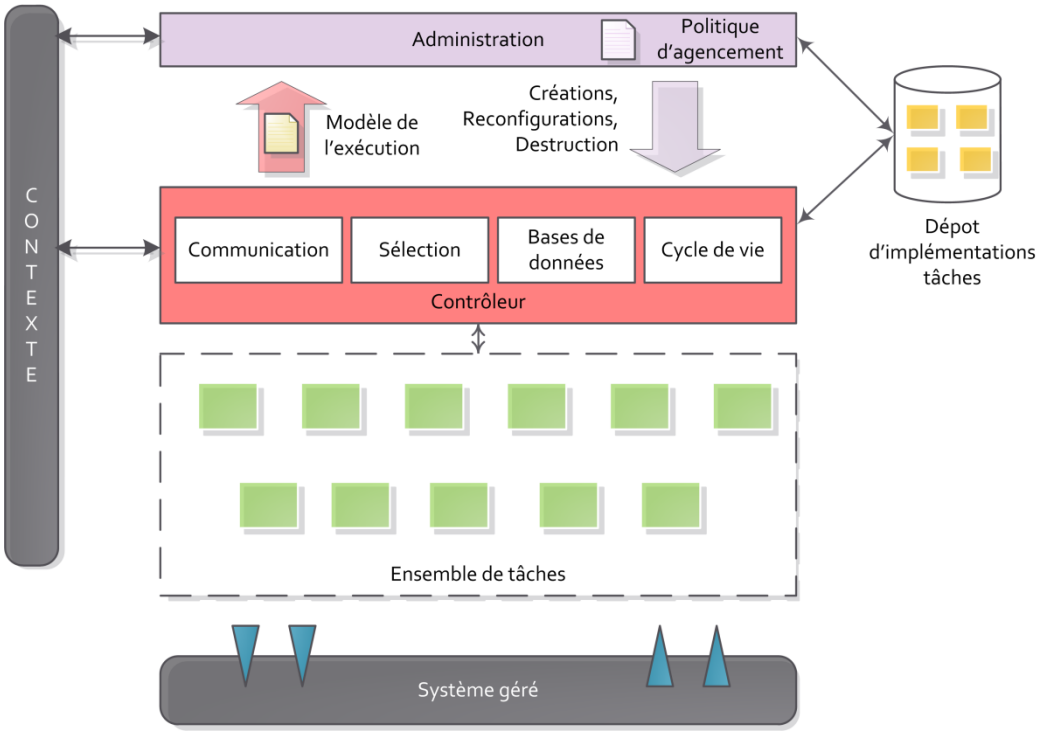

Figure 33 : les mécanismes de gestion des tâches

La sélection intervient lorsque deux tâches incompatibles veulent traiter les mêmes données. Ce mécanisme n'intervient que lorsque nécessaire pour arbitrer Chaque tâche formule une requête portant sur l'ensemble des informations qu'elle souhaite traiter. Ces requêtes peuvent également s'accompagner d'informations supplémentaires afin d'assister le mécanisme de sélection dans ses choix. Elles peuvent par exemple noter l'importance des données, ainsi que fournir une évaluation des ressources qui seront nécessaires à traiter ces données. Dans notre approche, le mécanisme de sélection est interchangeable.

Enfin, la base de données permet aux tâches de stocker et partager des informations. Ce mécanisme est optionnel ; il a pour vocation principale de stocker les informations ponctuelles (déclenchement d'une alarme) qui sinon nécessiteraient l'envoi permanent de messages.

L'administration offre les outils nécessaires à la modification du comportement global du gestionnaire en permettant la découverte de nouvelles tâches. La modification de l'architecture et des politiques d'agencements s'effectue de deux façons (figure 34) :

- **manuellement** : l'expert des systèmes autonomiques peut reconfigurer les politiques d'agencements des tâches en s'appuyant sur les interfaces d'administrations que fournit le framework. L'un des outils de base est une IHM qui synthétise les informations collectés au niveau du gestionnaire de tâche.
- **automatiquement** : un gestionnaire de politiques observe et analyse le fonctionnement du gestionnaire grâce au modèle de l'exécution fourni par le gestionnaire de tâches.

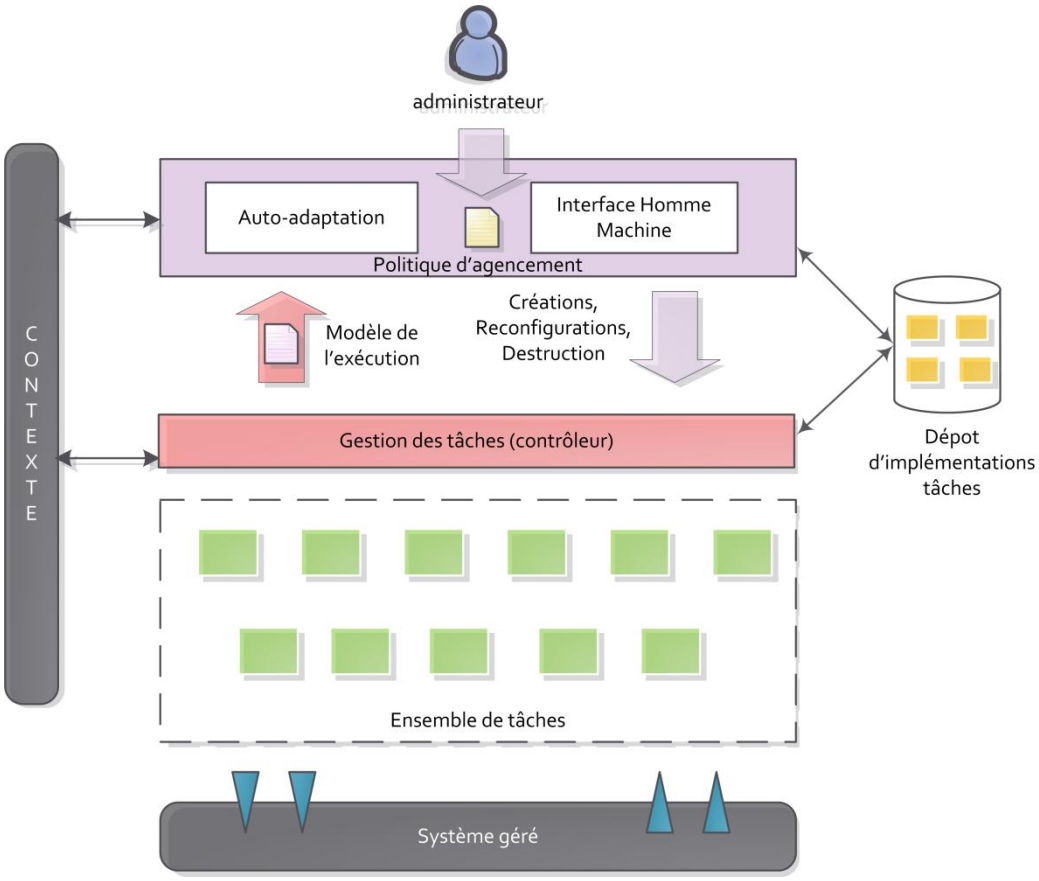

Figure 34 : mécanismes de gestions des politiques

Dans notre approche, une politique de gestion est l'émanation de l'ensemble des tâches déployées, de leur configuration, de la configuration des mécanismes de sélection et de l'agencement de ces tâches. Elle s'écrit dans une description de l'architecture du gestionnaire souhaité, le framework fournit pour ce faire un langage de description d'architecture (ADL).

Lorsque la configuration est automatique, ce niveau est principalement constitué du gestionnaire de politique. Il observe le fonctionnement de la boucle de contrôle et effectue les ajustements nécessaires en ajoutant ou supprimant des tâches administratives. Pour ce faire, il agit sur le modèle de l'architecture souhaité. Sur la base des informations fournies par le modèle à l'exécution, le gestionnaire de politique peut choisir de modifier dynamiquement les processus de composition de tâches, afin d'obtenir un comportement global qui correspond plus étroitement à ses objectifs de haut niveau. Ce niveau connaît les politiques d'administration fixées par

l'administrateur ; et il est responsable du comportement global du gestionnaire. Le mécanisme d'auto-adaptation a la possibilité d'ajouter, supprimer ou reconfigurer les tâches, ainsi que de reconfigurer les paramètres des mécanismes de sélection. Par exemple, un composant de gestion identifié comme défectueux ou inefficace peut être dynamiquement remplacé par une alternative en reconfigurant les tâches qui travaillent de concert avec lui.

4 SYNTHESE

Les applications modernes sont de plus en plus dynamiques et hétérogènes. L'architecture des systèmes modernes n'est plus figée et prévisible. Il en va de même pour les besoins des utilisateurs, les capacités des ordinateurs et des réseaux, et les technologies utilisées. Il nous parait essentiel que les gestionnaires autonomiques soient dynamiquement adaptables et extensibles pour prendre en compte ces changements et faciliter la maintenance.

Les travaux portant sur l'architecture interne des gestionnaires ne nous paraissent pas assez focalisés sur le dynamisme. L'architecture de référence MAPE-K et les architectures similaires ne proposent qu'un découpage logique qui ne correspond pas nécessairement à un découpage adéquat lors de l'implémentation. En particulier, l'intégration de différents objectifs et leur développement indépendant n'est pas toujours évident. Actuellement de nombreux gestionnaires sont construits soit de façon monolithique soit en utilisant des règles ou des composants gros grain, ce qui constitue un handicap pour la maintenance ou le dynamisme et ne favorise pas la réutilisation.

Pour toutes ces raisons nous introduisons un niveau d'abstraction intermédiaire et complémentaire : les tâches d'administration. Une tâche d'administration est une unité fortement cohérente et faiblement couplée qui réalise une opération spécifique de la boucle de contrôle. Ces tâches permettent notamment l'implémentation des architectures MAPE, par exemple une tâche de supervision du processeur ou une tâche de résolution d'un problème particulier. La granularité reste donc potentiellement plus faible qu'un bloc MAPE. Il faut cependant souligner qu'avec la notion de tâche composite, la granularité des tâches d'administration est variable et qu'une seule tâche peut réaliser une boucle de contrôle complète. Les tâches offrent ainsi beaucoup de flexibilité.

Le gestionnaire autonomique résulte de la combinaison opportuniste de ces tâches. A chaque instant, l'ensemble de tâches utilisées peut être modifié en fonction du contexte et des informations remontées par la plateforme. Des mécanismes de sélection permettent de gérer les conflits éventuels et permettent d'assurer une cohérence du comportement du gestionnaire.

Nous proposons un framework permettant la mise en œuvre de cette vision. Ses principales caractéristiques sont les suivantes :

- proposer un modèle homogène et dynamique d'intégration des fonctionnalités autonomiques ;
- permettre et favoriser la réutilisation de fonctionnalités d'administration entre plusieurs solutions autonomiques pour diminuer le temps de développement et augmenter la fiabilité ;
- autoriser l'implémentation des objectifs d'administration en isolation pour simplifier le travail des développeurs ;
- fournir des mécanismes permettant la sélection et la combinaison des tâches d'administration afin d'obtenir des comportements complexes : c'est l'expression de la combinatoire ;
- faciliter l'évolution et l'extension des stratégies de gestion globale en supportant l'ajout, la mise-à-jour et la suppression des tâches ainsi que la modification de leur logique de collaboration ;

- fournir les informations nécessaires à l'analyse du comportement du gestionnaire et à l'évaluation des tâches.

Nous fournissons également une interface permettant la visualisation des tâches en cours de fonctionnement et la modification dynamique de l'architecture du gestionnaire.

Dans la suite, nous présentons une architecture très générique de ce framework en faisant abstraction de l'implémentation. En particulier, nous présentons l'architecture des tâches et nous discutons des différents choix possibles pour permettre leur collaboration. Puis nous présentons une implémentation particulière sous la forme d'un modèle à composants, de sorte que le travail des développeurs est facilité et que la réutilisation est favorisée. Enfin nous donnons un exemple d'application développée au dessus de ce framework.

Chapitre 5 - **Framework proposé**

Dans le chapitre précédent nous avons présenté une vision globale de notre approche pour le développement de gestionnaires autonomiques. Nous avons proposé de construire les gestionnaires autonomiques par la composition opportuniste de tâches d'administration simples et homogènes.

Ce chapitre décrit l'architecture générale de notre framework et discute des techniques permettant l'administration et l'activation opportunistes des tâches, ainsi que la gestion des conflits potentiels. Il se divise en quatre parties.

Dans un premier temps nous nous focalisons sur les tâches atomiques et présentons ainsi les fonctionnalités minimales du framework. Ces tâches atomiques sont constituées de plusieurs modules qui se focalisent chacun sur une préoccupation particulière de l'activité d'une tâche, notamment la communication, son déclenchement, la gestion de conflits.

Dans un second temps, nous décrivons l'infrastructure de contrôle sur laquelle s'appuient les tâches. Cette architecture joue un rôle clef dans la gestion du cycle de vie, la communication et l'organisation des tâches.

Dans un troisième temps, nous abordons la question de l'administration du gestionnaire et notamment sa construction et son adaptation. La couche d'administration est celle qui fournit les outils nécessaires à l'adaptation manuelle ou automatique du gestionnaire autonomique.

A ce stade, cette première version du framework est pleinement fonctionnelle. Cependant, lorsque la taille du gestionnaire est importante, des problèmes de passage à l'échelle peuvent survenir. C'est pourquoi, dans un dernier temps, nous introduisons la notion de tâche composite et détaillons les particularités de ces blocs et les modifications qui doivent être effectuées sur le framework.

1 TACHES D'ADMINISTRATIONS

1.1 ARCHITECTURE INTERNE GLOBALE

Nous nous focalisons dans un premier temps sur l'architecture des tâches atomiques qui sont au cœur de notre approche.

Pourquoi les tâches ? Nous l'avons vu dans l'état de l'art, les gestionnaires autonomiques sont amenés à jouer un rôle de plus en plus important dans l'industrie. La délégation de responsabilités à des logiciels n'est pas sans risque et une panne ou une mauvaise conception peuvent mettre à mal la stabilité des systèmes, voire les détruire. Plus qu'ailleurs, l'attention doit se porter sur la qualité et la maintenance de ces gestionnaires mais aussi leur évolutivité. Or il est peu probable qu'ils soient parfaits dès leur conception, tout d'abord parce que les entreprises n'auront pas les moyens de consacrer le temps nécessaire à une mise en œuvre de qualité. Ensuite, l'évolution des systèmes est de plus en plus rapide et le gestionnaire sera nécessairement utilisé dans des conditions inattendues ou imprévisibles qui, pour peu que le gestionnaire soit complexe, nécessiteront une adaptation. La problématique est donc double : favoriser la création de gestionnaires autonomiques fiables et permettre leur évolution durant leur exploitation.

Pour adapter le gestionnaire, il faut pouvoir changer sa configuration mais aussi sa logique de fonctionnement. Se contenter de jouer sur des propriétés de configuration exposées par celui-ci offre peu de flexibilité. Si l'on construit un gestionnaire statique d'un seul tenant, il faut qu'il soit suffisamment généraliste pour être capable de s'adapter à un grand nombre de situations et par conséquent proposer un grand nombre de paramètres de configuration. Concevoir et configurer de tels gestionnaires est difficile. En particulier, s'il n'est pas modulaire le gestionnaire doit posséder plusieurs versions des algorithmes qu'il utilise pour être en mesure de s'adapter, ce qui aboutit en définitive à une architecture monolithique ou à blocs de forte granularité impossible à maintenir.

A notre sens, pour accélérer le développement des gestionnaires tout en maintenant un niveau de qualité suffisant, il faut fournir un modèle simple de développement qui favorise la réutilisation. Cela passe par l'introduction d'une unité d'encapsulation homogène des fonctionnalités de la boucle de contrôle. C'est la nature des tâches : fortement cohérentes, elles accomplissent une activité spécifique et simple permettant d'avancer dans le déroulement de la boucle de contrôle. Elles fournissent un modèle homogène d'intégration de ces fonctionnalités en possédant la même architecture et en répondant aux mêmes API quelle que soit l'étape réalisée dans la boucle de contrôle. L'homogénéité du modèle simplifie le développement en limitant le nombre de technologies différentes à maitriser à chaque étape du développement. L'encapsulation de ces fonctionnalités en unités fonctionnellement indépendantes et cohérentes permet leur réutilisation et masque l'hétérogénéité des implémentations et des méthodes de collecte, d'inférence ou d'action utilisées.

L'autre principe mis en œuvre avec les tâches est celui de la séparation des préoccupations. Comme les tâches partagent la même architecture, les fonctionnalités développées pour une tâche peuvent être réutilisées pour l'implémentation d'une autre. En particulier, nous verrons que les mécanismes de communication, sélection, déclenchement sont très largement réutilisés entre les tâches. Ainsi le modèle que nous proposons établit une séparation entre :

- **la fonctionnalité :** les tâches sont décrites par un type qui ne fait référence qu'à la fonctionnalité. Le type n'induit pas une implémentation ou une technique d'inférence, de collecte ou d'action quelconque. Cette complexité est masquée à l'expert lors de l'assemblage des tâches.
- **la réalisation de cette fonctionnalité** : lors de l'implémentation d'une tâche, le développeur se focalise uniquement sur la réalisation de la fonctionnalité. Il peut

se concentrer sur le développement de l'algorithme spécialisé car il n'a pas à se soucier de la façon dont les informations seront transmises ni des conditions dans lesquelles la tâche sera utilisée. Cela encourage le développement d'algorithmes plus génériques.

- **les conditions dans lesquelles une tâche est activée** : les conditions nécessaires à l'activation ou la désactivation d'une tâche sont décrites séparément de l'algorithme de traitement. Comme le modèle est homogène, du code développé et utilisé pour une tâche de collecte – par exemple du code permettant d'exprimer des conditions sur l'heure d'activation – est réutilisable au sein d'une tâche de prise de décision.
- **les relations des tâches** : la logique d'assemblage des tâches est exprimée séparément de la fonctionnalité, si bien qu'elle peut être modifiée sans changer l'algorithme codant la fonctionnalité. Ainsi l'expert peut étendre ou modifier les fonctionnalités des gestionnaires en modifiant les relations entre les tâches du système.

Pour réaliser cette séparation, l'architecture d'une tâche est nécessairement modulaire. Une tâche est un module de granularité intermédiaire divisé en plusieurs sous modules, chacun rattaché à l'une des préoccupations évoquées précédemment. Ainsi une tâche est-elle constituée de cinq types d'éléments (figure 35) :

- un **port d'entrée** et un **port de sortie** qui permettent l'échange des données avec les autres tâches. Ils s'appuient sur le mécanisme de communication de l'architecture de contrôle, présenté dans la section suivante, pour souscrire et publier les informations collectées et produites par la tâche. La configuration de ces ports permet de déterminer avec qui la tâche est en **relation**.
- un **mécanisme déclenchement** (le *scheduler*) qui planifie le traitement des données reçues. Pour cela, il emmagasine les données dans un tampon jusqu'à ce qu'une condition de déclenchement soit remplie. Celle-ci peut porter sur les données collectées mais également sur des événements extérieurs. Ce mécanisme précise donc **quand** la tâche est activée.
- un **mécanisme de contrôle** (le coordonateur) qui permet d'assurer plusieurs tâches ne traiteront pas en même temps des informations conflictuelles. Il s'appuie pour ce faire sur le mécanisme de sélection proposé par l'infrastructure de contrôle que nous présentons plus loin. Ce mécanisme permet de décider qui traite les informations.
- le **processeur** de la tâche qui implémente le traitement proprement dit. C'est l'algorithme développé pour accomplir une tâche de la boucle de contrôle. Ce traitement peut être très spécifique pour la résolution d'un problème particulier ou plus générique, tel le calcul d'une moyenne sur un ensemble de données. L'écriture de cette fonction détermine **comment la tâche opère**.
- la **gestion de statistiques** qui calcule des statistiques sur l'utilisation de la tâche permettant à l'administrateur et aux algorithmes de sélection d'évaluer l'activité de la tâche et du gestionnaire.

Certains mécanismes peuvent être optionnels : par exemple, par défaut le *scheduler* et le coordonateur laissent passer toutes les informations sans filtre et le calcul des statistiques n'est pas obligatoire. Ainsi avec une configuration minimale et adéquate, il est possible de construire un gestionnaire avec un minimum d'appels à l'architecture de contrôle. Nous verrons notamment que cela est possible lorsque l'implémentation des ports établit une communication directe entre les tâches.

L'objectif du framework est de fournir diverses implémentations de ces mécanismes pour apporter de la flexibilité dans le développement des gestionnaires. L'ensemble de ces mécanismes

peut être adapté ou réécrit par le développeur pour correspondre aux besoins particuliers d'une application. Il se constitue alors une bibliothèque de fonctionnalités réutilisables.

A cela s'ajoute ce qui caractérise une tâche, à savoir :

- **son type** : chaque tâche s'engage à respecter un contrat défini dans un type qui décrit **la fonctionnalité**, la nature des informations échangées et ses propriétés de configuration ;
- **son état** : il indique l'état d'activité de la tâche et fournit des informations qui permettent aux mécanismes de sélection (ou de gestion de conflit) de l'architecture de contrôle de prendre des décisions ;
- **ses propriétés** qui regroupent les propriétés de configuration mais également les propriétés partagées avec les autres tâches qui sont stockées dans une base commune ;
- **l'identification de la tâche :** une tâche possède un identifiant unique permettant de la discriminer des autres tâches : un nom unique.

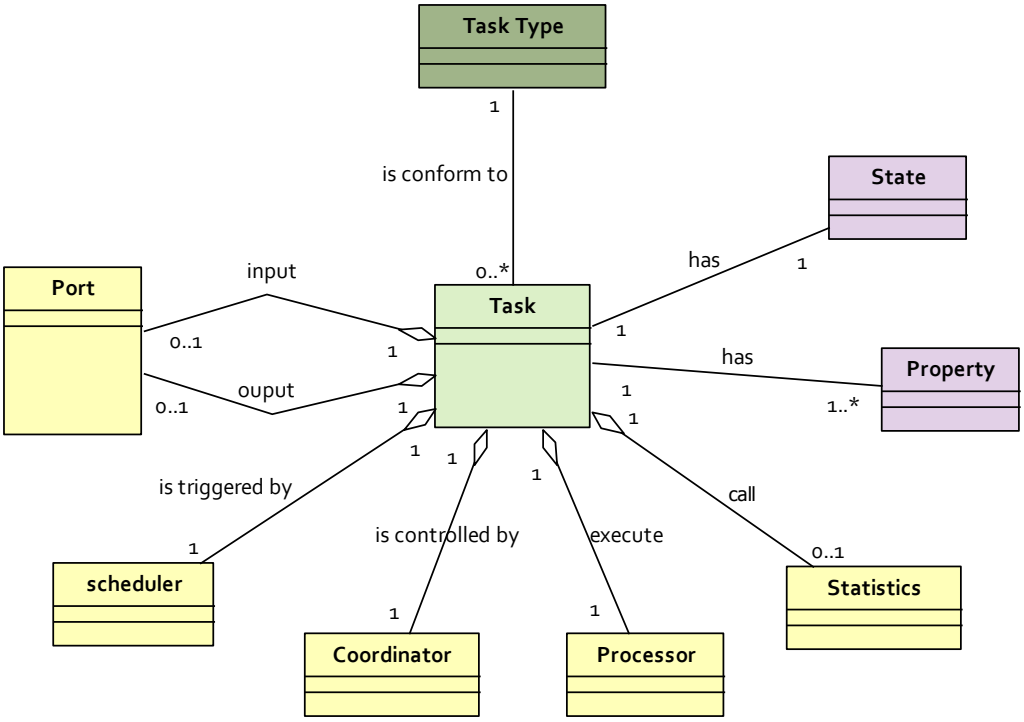

Figure 35 : Modélisation d'une tâche

Les modèles que nous présentons ici sont conceptuels et font abstraction des contraintes liées à la plate-forme d'exécution. Nous considérons ici que la tâche est une instance déployée et fonctionnelle ; nous ne traitons pas de la manière de découvrir, déployer, créer ces tâches. L'infrastructure d'exécution sous-jacente doit répondre à des exigences pour permettre la mise en œuvre de notre approche dans de bonnes conditions. En particulier, cet environnement d'exécution doit satisfaire les exigences suivantes :

- supporter le déploiement et le retrait des tâches à l'exécution afin de pouvoir modifier le comportement du gestionnaire par l'ajout ou le retrait de fonctionnalités ;

- permettre la création, le démarrage, l'arrêt et la destruction des tâches lors de l'exécution ;
- modifier des liaisons dynamiquement pour faire évoluer les politiques de liaison entre les tâches durant l'exécution.

L'ensemble de ces caractéristiques participe à l'évolution du gestionnaire et à la création des conditions pour un comportement opportuniste par l'ajout, le retrait de tâches et la modification de leurs relations en fonction du contexte. Il paraît également important que l'infrastructure :

- supporte la découverte dynamique des modules et donc des tâches permettant ainsi au système de découvrir automatiquement de nouvelles fonctionnalités ;
- permette la supervision des instances de tâches ;
- offre des mécanismes d'injection/interception de valeurs et d'interception de méthodes. Ils faciliteraient l'observation de la tâche, la génération de statistiques et simplifieraient le développement et la configuration de la tâche.

Nous verrons lorsque nous présenterons le modèle d'exécution que les architectures à services sont tout à fait adaptées à ces problématiques. Toutefois, d'autres technologies plus traditionnelles peuvent être utilisées en faisant un compromis sur le dynamisme ou l'évolution du gestionnaire.

Les modules d'une tâche s'appellent séquentiellement dans cet ordre (figure 36) :

1. le port d'entrée, qui collecte via le mécanisme de communication les données nécessaires à la tâche ;
2. le scheduler, qui stocke les informations dans son tampon. Lorsque les conditions nécessaires au déclenchement de la tâche sont remplies, le scheduler transmet les informations au coordonateur.
3. le coordinateur, qui utilise le mécanisme de sélection fourni par la couche de contrôle pour déterminer quelles données la tâche peut traiter.
4. la fonction, qui utilise les données et produit –si nécessaire – de nouvelles informations. En parallèle l'algorithme de statistique calcule des statistiques d'utilisation.
5. le port de sortie, qui envoie les informations produites vers la tâche suivante.

L'échec d'une étape conduit à l'arrêt du traitement et à la suppression des informations liées. La présence de port d'entrée ou de sortie n'est pas obligatoire. En particulier les tâches de collecte d'informations et celle qui agissent sur la ressource n'en ont pas nécessairement.

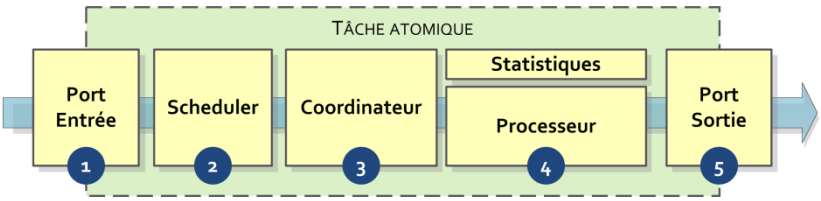

Figure 36 : Ordre d'appel des sous-modules d'une tâche

Dans la suite de cette première partie nous décrivons les sous-modules d'une tâche en suivant cet ordre. Nous décrivons ensuite les propriétés de la tâche.

1.2 LES PORTS ET LES DONNEES MANIPULEES PAR LA TACHE

Les tâches traitent et réagissent à des informations et événements en provenance de leurs ports de communications.

La façon dont la tâche communique avec les autres tâches est manifestement une préoccupation indépendante de la façon dont elle traite les informations. La séparation est nécessaire pour permettre une évolution facile du mécanisme de communication, en gérant par exemple les problématiques de distribution, et pour ne pas réécrire systématiquement le code lié à cette communication. Nous verrons plus loin, lors de la présentation de l'architecture de contrôle, que la communication peut s'effectuer soit directement entre tâches, soit via un bus à messages. Chacune de ces méthodes présente des avantages et c'est pourquoi la possibilité de changer facilement le mode de communication selon le gestionnaire envisagé est importante.

Les tâches communiquent par l'échange de messages qui contiennent les informations nécessaire à l'algorithme de traitement. Nous introduisons deux sous-modules (figure 37) :

- le **port d'entrée** qui collecte les informations pertinentes en provenance des autres tâches. Lorsqu'il ne trouve pas les données nécessaires au fonctionnement de la tâche, il avertit l'architecture de contrôle. La tâche est alors dans l'état invalide.
- le **port de sortie** envoie les données aux ports d'entrée des autres tâches. Il s'assure que les données produites par la tâche sont conformes à ce que la tâche déclare produire. Si une tâche produit une donnée qu'elle n'a pas déclarée, le port de sortie déclenche une erreur.

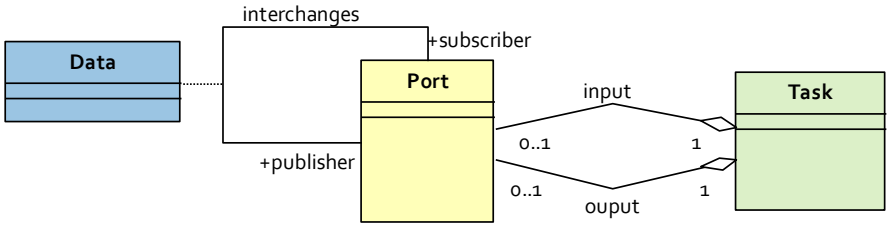

Figure 37 : Port d'entrée et de sortie d'une tâche

Le mécanisme de communication se distribue entre les ports d'entrée et de sortie des tâches et l'architecture de contrôle qui fournit le support de communication. Ces deux parties sont donc intimement liées. L'architecture de contrôle n'apparait pas explicitement dans la figure 37 mais implicitement c'est elle qui permet de réaliser le lien entre les ports.

Le point essentiel ici est que quelle que soit la nature des ports d'entrée et de sortie, la communication entre tâches se fait par publication/souscription d'informations. Les tâches ne se connaissent pas a priori mais connaissent les informations qu'elles produisent et utilisent. Il n'y a pas de dépendance fonctionnelle entre tâches mais dépendance de données. Cette caractéristique est fondamentale pour permettre l'ajout ou le retrait de fonctionnalités au gestionnaire. Les liaisons entre les tâches sont dynamiques et se modifient en fonction de ce que les tâches produisent/utilisent. C'est particulièrement vrai pour les tâches qui modifient dynamiquement la nature de leur production et utilisation, comme nous le verrons plus loin pour les tâches composites.

Cette notion de port est très commune et se trouve dans de nombreux frameworks tels que les workflows ou les intergiciels de médiation. Il y a une différence notable entre ce modèle et celui de ces frameworks. Nous ne faisons pas apparaître ici la notion de liaison sous forme d'une classe car dans notre système ces liaisons sont éphémères et n'ont pas de sens compte tenu du dynamisme.

Pour la communication, nous avons effectué deux choix : limiter le nombre de ports et rendre obligatoire la déclaration des données produites.

Nous partons du principe qu'une tâche n'a que deux ports au maximum. C'est avant tout pour limiter la complexité du framework et de l'implémentation. Il est envisageable d'avoir plusieurs ports d'entrée ou de sortie. Le développeur des tâches a alors le choix entre plusieurs mécanismes de communication pour échanger des données, par exemple : un moyen pour la communication avec des tâches locales, un autre pour la communication avec des tâches distantes. Une autre raison peut être de vouloir avoir un canal de données compressées ou sécurisées et un canal normal de communication. Cependant, devoir gérer en parallèle plusieurs mécanismes de communication complexifie le gestionnaire, d'où notre réticence. Ce détail est avant tout technique. L'étude de ces mécanismes, en particulier la distribution des tâches, dépasse le cadre de cette thèse.

La cardinalité du diagramme souligne que ces ports sont optionnels. Les tâches de collecte de données n'ont pas besoin de port d'entrée ; les tâches d'action ne nécessitent pas de port de sortie. C'est ce qu'illustre la figure 38 avec une tâche de collecte suivie d'une tâche d'action. Il est possible d'envisager que ces deux catégories de tâches bénéficient de ports pour la collecte d'événements, mais alors ces ports seront relativement spécifiques à l'activité de la tâche. Dans l'architecture, nous avons choisi de concentrer ces traitements spécifiques dans le processeur.

Un autre choix réside dans l'obligation pour chaque tâche de déclarer les données qu'elle produit (figure 37). Dans l'absolu, cette déclaration n'est pas nécessaire pour permettre la communication : les tâches ne dépendant que des données, elles n'ont pas besoin de savoir qui les produit. Toutefois cette contrainte nous semble importante pour plusieurs raisons. D'abord contraindre le développeur à déclarer les données qu'il produit l'oblige à stabiliser le comportement d'une tâche. Ensuite, la connaissance des producteurs de données permet au port d'entrée de déterminer à chaque instant si une tâche dispose des sources d'information qui lui sont nécessaires. Si ce n'est pas le cas, il est possible de notifier la couche supérieure ou l'administrateur pour trouver une solution afin de combler le manque – en rajoutant une tâche par exemple. Cette caractéristique nous parait essentielle pour garantir le bon fonctionnement du gestionnaire. Enfin, il est possible à un instant donné de connaître l'ensemble des combinaisons possibles des tâches, ce qui fournit des informations sur les comportements du gestionnaire.

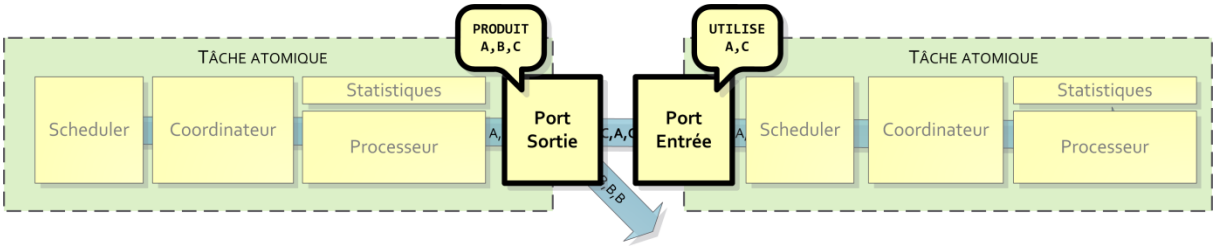

Figure 38 : Les tâches déclarent les types de données qu'elles produisent et consomment

Attachons-nous maintenant aux données proprement dites.

Quelle est la nature des informations échangées ? Nous ne pouvons pas ici détailler le contenu des données, car le système n'impose pas de contrainte sur la nature et la représentation des informations. Leur contenu, leur nature dépend de l'activité de la tâche. Ainsi, les tâches peuvent utiliser des représentations très variées d'une même information : un entier, une chaine de caractères, un arbre ou tout objet du système. Durant l'exécution, la nature des données traitées fluctue en fonction de l'ajout de nouvelles tâches et du retrait de tâches obsolètes. Cette flexibilité permet au développeur de choisir la représentation la plus appropriée pour l'identification d'un problème particulier.

Des informations communes sont toutefois nécessaires. Même s'il est important de permettre une représentation flexible de l'information, il faut en contrepartie s'assurer que les

tâches disposent d'un minimum de connaissances sur la nature des données échangées : un langage commun. Elles doivent être capables d'interpréter les informations produites par les autres tâches. C'est pourquoi la notion de type de donnée est importante pour assurer cette cohérence.

Un type de donnée (figure 39) possède un nom unique et une description. Il définit un ensemble d'attributs qui ont un nom unique et une valeur par défaut. La description, tant pour le type que pour les attributs, correspond à la documentation. Elle décrit la façon de l'utiliser et le contenu. La sémantique n'est donc pas ici décrite formellement.

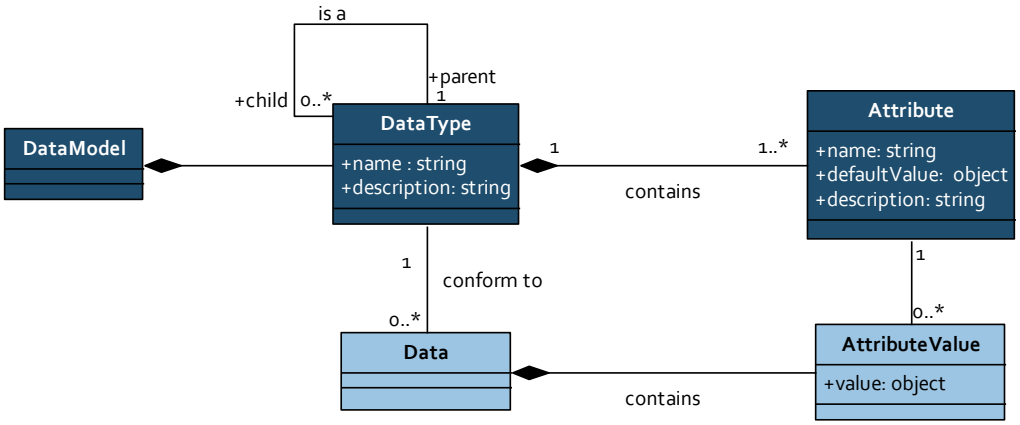

Figure 39 : Modèle de donnée

La description d'un mécanisme de typage complet dépasserait le cadre de cette thèse, notre objectif est d'en montrer l'intérêt et de fournir une amorce à la réflexion. Le typage introduit ici est minimal ; les vérifications sont essentiellement syntaxiques. L'objectif est d'assister le développeur en attirant son attention sur le besoin des tâches d'avoir un langage commun. Si des contrôles supplémentaires sont nécessaires, on peut ajouter à moindre coût un type « simple » aux attributs en s'inspirant par exemple de la description des types dans les services Web ou en s'appuyant sur le typage fourni par la plateforme d'exécution. Aucun mécanisme de typage n'est parfait et un mécanisme trop complexe aura un coût en performance.

On définit une hiérarchie entre les types et on exprime deux contraintes simples de conformité :

- Un type B est un sous-type de A si B possède l'ensemble des attributs de A.
- Une donnée est conforme à son type si elle donne une valeur à chacun des attributs définit par le type.

De cette manière nous introduisons une dose de polymorphisme. Les sous-types contiennent autant d'informations que leur parent et peuvent être manipulés comme leur parent. Prenons par exemple le cas d'une tâche qui utilise un type « *Alarme* » qui contient un attribut « *ON* » de type booléen qui vaut « *vrai* » lorsqu'une alarme est déclenchée. Cette tâche peut tout à fait utiliser un sous-type *d'Alarme* de façon transparente. Si par exemple une tâche produit une donnée de type *AlarmeLocalisée* qui en plus de l'attribut « *ON* » contient un attribut « *localisation* », cette donnée peut être vue comme une donnée de type *Alarme* sans conséquence.

Les données sont encapsulées dans des messages qui contiennent deux parties : un entête et la donnée proprement dite. Cet entête contient les informations suivantes :

- **le type et l'identifiant unique de la donnée** qui permet de faire référence au message ;

- **l'historique de traitement** qui, lors de l'activation du traçage des messages, contient le nom du créateur et la liste des messages à l'origine de celui-ci ;
- **des méta-informations** telles que le temps de création qui, lorsqu'elles sont collectées, permettent d'évaluer la qualité d'une réponse du gestionnaire.

Les entêtes des messages sont les informations échangées entre les tâches et le mécanisme de gestion de conflit pour permettre la répartition des traitements des données lorsque cela est nécessaire.

L'historique du traitement est une information importante car elle permet d'identifier les chemins empruntés par les messages au sein du gestionnaire autonomique. La couche supérieure ou l'administrateur ont alors des informations permettant l'identification des cycles ou des famines. Un temps de vie à l'image du TTL des paquets IP peut ainsi être ajouté si nécessaire.

Le calcul de l'historique de traitement et des méta-informations ne sera qu'approximatif, en associant les données en entrée de la fonction avec les données produites par celle-ci lors d'un appel. Cette méthode donne des résultats erronés lorsque la fonction retient des données d'un appel sur l'autre et ne prend pas en compte les données en provenance de la base de données.

Ces informations sont masquées au développeur de la fonction. Il ne reçoit que les données. Il ne peut pas construire d'algorithme dépendant d'une autre tâche, ce qui serait contraire à la séparation établie et préconisée plus haut.

1.3 LE SCHEDULER : DECLENCHEMENT DE LA TACHE

Après la collecte des données vient en second lieu la planification de leur traitement. Lorsqu'il a collecté des données, le port d'entrée les transmet au mécanisme qui gère le déclenchement de la tâche : le scheduler (figure 39 et figure 40)

L'activation d'une tâche est événementielle. La collecte d'informations n'implique pas nécessairement leur traitement. Chaque tâche attend l'opportunité de contribuer au processus d'adaptation en observant l'environnement. La capacité des tâches à évaluer la pertinence des données collectées et à planifier leur traitement est au cœur de notre approche. C'est ce qui crée les conditions d'un comportement opportuniste du gestionnaire en lui permettant de réagir à des événements contextuels.

Le système a un caractère opportuniste parce que l'activation des tâches n'est pas connue ni planifiée à l'avance, mais déterminée à chaque étape du raisonnement du gestionnaire en fonction de la situation courante. L'événement ou la combinaison d'événements déclencheurs d'une tâche résulte de changements significatifs de l'état du gestionnaire, par exemple la nature et la quantité des informations reçues, le nombre d'activations ou l'heure.

Exprimer séparément les conditions d'activation de la tâche et l'algorithme de traitement est intéressant pour plusieurs raisons.

D'abord, ce découplage favorise la réutilisation, par exemple si le développeur développe un algorithme qui compte et attend qu'un nombre déterminé de données soit collectées, il peut l'utiliser avec n'importe quelle tâche indépendamment de l'opération réalisée par la tâche. Il est également plus facile d'attribuer la même condition de déclenchement sur plusieurs tâches.

Ensuite, permettre la modification des facteurs de déclenchement d'une tâche permet de modifier globalement le comportement du gestionnaire.

Enfin, cette séparation permet à l'administrateur et à la couche supérieure d'identifier et comprendre rapidement les conditions nécessaires au déclenchement d'une tâche, ce qui est difficile lorsque les codes de fonctionnalité et d'activation sont mélangés.

L'ensemble des données collectées passe par un module appelé scheduler (ou planificateur) qui décide selon sa configuration et sa stratégie interne du moment opportun pour traiter les données. Il est composé de deux parties (figure 40) :

- **un tampon** qui conserve une copie des données tant qu'elles n'ont pas été traitées et qu'elles ne sont pas obsolètes ;
- **des conditions de déclenchement** qui consultent régulièrement les données collectées et évaluent leur pertinence en fonction du contexte. Elles peuvent également observer des informations et s'abonner à des événements contextuels (heure, objectif de l'administrateur) ou calculer des informations (nombre de sollicitations)

Toutes les données collectées par les tâches transitent par un espace de stockage. Ce **tampon** est un module générique facilement réutilisable d'un scheduler à l'autre ; il n'y a donc pas de raison pour les développeurs d'une tâche de le modifier. Il fournit une garantie très importante : celle que les données stockées ne sont pas devenues obsolètes. En effet, conserver des données sur une trop grande durée risquerait de provoquer l'exécution des tâches sur la base d'informations qui ne font plus sens. Pour éviter cela, chaque donnée stockée est datée et un mécanisme d'expiration consulte régulièrement le tampon pour purger les données périmées. Nous avons choisi de faire dépendre la durée de vie d'une donnée de son type. Ainsi, lors de la création d'un type de donnée, le développeur indique une durée maximale de conservation. Bien sûr, le tampon peut être configuré pour utiliser d'autres délais pour certains types de tâche, voire pour les ignorer.

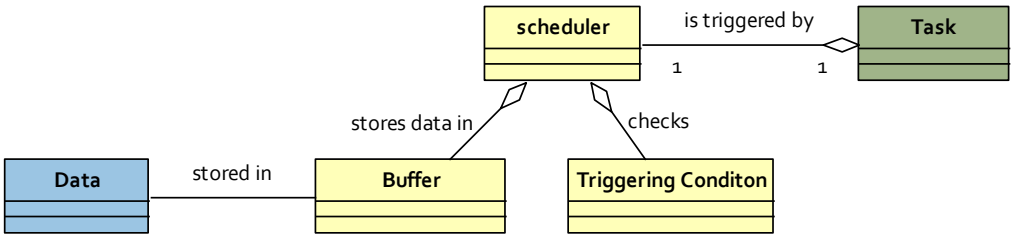

Figure 40 : Les sous-modules d'un scheduler.

L'utilisation d'un tampon pour chaque tâche peut laisser craindre une consommation élevée de mémoire du fait de la duplication des données. Cependant, dans notre modèle les données produites ne peuvent être modifiées (lecture seule) et par convention chaque tâche produit de nouvelles données. Nous verrons dans l'implémentation qu'avec JAVA, lorsqu'elles ne sont pas distribuées, les tâches ne stockent qu'une référence (un pointeur) vers la donnée, ce qui limite l'utilisation mémoire – car les données ne sont pas dupliquées. Le stockage par chaque tâche peut éviter un trop grand nombre de communications avec une base centralisée et apparait donc comme inévitable. En outre, le mécanisme d'expiration est un facteur contribuant à limiter la consommation mémoire.

Régulièrement, de leur propre initiative ou lorsqu'une nouvelle donnée s'ajoute au tampon ou encore lors de la modification du contexte, les **conditions de déclenchement** sont vérifiées. Leur nature est très variable, le plus souvent elles concernent :

- **le type des données collectées** : une tâche peut exprimer le besoin que deux types d'événements ou de données soient présents. Une condition de type « A ET B OU C» n'activera la tâche que si, et seulement si, le tampon contient (A,B) ou (A,B,C) ou (C). Pour illustrer, les tâches de notre application test ne réagissent qu'en cas de dépassement d'un seuil d'utilisation de mémoire et de dépassement d'un seuil processeur.

- **la quantité de données collectées** : lorsqu'une tâche a besoin de faire des calculs sur un ensemble suffisamment grand de données (moyenne, écart-type,...) ;
- **la valeur des données collectées,** ce qui permet de filtrer et d'écarter des données. Il est ainsi possible d'éliminer les données manifestement aberrantes : valeur négative ou nulle lorsque cela est impossible.
- **un intervalle de temps ou une heure précise**, pour envoyer des informations à intervalle de temps régulier lors de la collecte de données.
- **la date de dernière activation de la tâche** : il est ainsi possible de faire attendre la tâche entre deux activations, de sorte que les effets du processus d'adaptation soient visibles et que la tâche ne traite pas deux fois le même problème.
- **les informations partagées avec les autres tâches**. Le scheduler de chaque tâche a accès à l'espace de stockage partagé entre les tâches et peut ainsi réagir à des changements de valeur.

Les événements utilisés par le scheduler appartiennent majoritairement au flot d'exécution – réception d'une nouvelle donnée, seuil de donnée dépassé – mais ils font également référence au contexte. La tâche réagit à des événements internes et externes. Les événements internes concernent le flot d'exécution, l'état de la tâche ou encore l'historique d'exécution. La collecte des événements externes relève de la responsabilité des conditions de déclenchement ; ils se rapportent à l'environnement de la tâche, par exemple l'heure ou les ressources disponibles dans le système. Le facteur déclenchant l'activité d'une tâche peut donc être totalement indépendant des données reçues par la tâche.

Bien entendu, une condition peut être issue de la combinaison de plusieurs autres conditions plus simples. Une expression du type (ALARM ET (12h30, 16h00) OU ALARM.on = «true») combine des événements internes et externes et fait référence aussi bien au type qu'à la valeur des données.

Le framework fournit un ensemble de schedulers réutilisables qui peuvent être employés tels quels ou servir de squelettes à des schedulers plus complexes.

Il est très important de souligner que les conditions de déclenchement ne se contentent pas de déterminer le moment d'activation de la tâche mais également ce qu'elle doit traiter. Le scheduler a donc une influence sur l'ordre et la nature des données. Sur l'ordre d'abord car il peut accorder une priorité plus importante à certains types de données et demander leur traitement avant les autres. Sur la nature ensuite car il peut sciemment écarter des données jugées non significatives, soit parce qu'elles sont manifestement erronées, soit parce que le contexte ne s'y prête pas. Pour illustrer nous pouvons prendre l'exemple d'une tâche qui ne considère pas les données de la même façon selon qu'il fait nuit ou jour ou qu'une alarme est déclenchée.

Les conditions de déclenchement sont génériques et réutilisables mais ne permettent que la réalisation d'un filtrage grossier laissant nécessairement traiter des données qui ne sont pas pertinentes. Lorsqu'un filtrage plus précis et spécifique est nécessaire, il est possible de s'appuyer sur le bloc processeur.

Tout d'abord en utilisant les informations qu'il produit et que le scheduler observe ; ellesportent sur le déroulement des exécutions précédentes. Le scheduler peut alors déclencher la tâche sur la modification d'une information. Par exemple le processeur peut souhaiter ajuster dynamiquement en fonction des exécutions précédentes la quantité de données nécessaires.

Ensuite, en appelant à un algorithme spécifique implémenté par le processeur. Le processeur implémente alors la planification et la fonctionnalité. La séparation systématique des aspects que l'on sait pourtant fortement spécifiques et non réutilisables risquerait autrement de conduire à une trop grande fragmentation du code et d'influencer négativement l'efficacité de la tâche.

La figure 41 récapitule les échanges entre les schedulers et les autres modules :

1. le tampon reçoit les données brutes en provenance du port de communication, en notant leur date d'arrivé. Les données expirées sont purgées régulièrement. Le tampon notifie les conditions de déclenchement.
2. ces dernières observent l'environnement et le tampon jusqu'à ce que les conditions soient satisfaisantes.
3. à l'issue du processus d'évaluation des conditions, le scheduler produit une requête décrivant l'ensemble des données que la tâche se propose de traiter (figure 41). A ce stade du traitement des données, chaque tâche agit indépendamment du fonctionnement des autres tâches et la requête ne tient pas compte des conflits éventuels entre tâches. Dans la majorité des cas, lorsqu'il n'y a pas de conflit, c'est cette requête qui sera traitée, dans les autres cas seul un sous-ensemble des données identifiées sera traité.

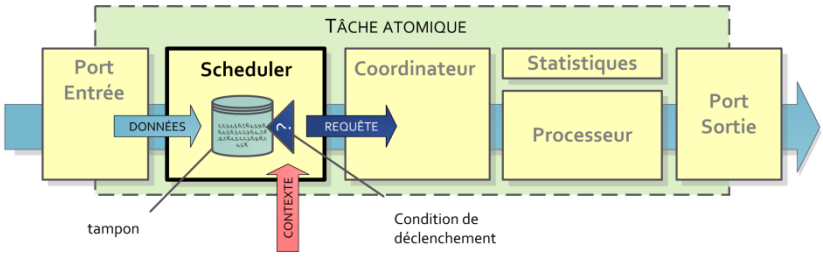

Figure 41 : Planification de l'exécution des données

Pour donner une latitude au processus de sélection chargé de résoudre les conflits, les requêtes sont décrites sous la forme d'expressions qui déterminent en intension l'ensemble des données à traiter. Cette expression porte sur les entêtes de données à traiter et utilise des combinaisons de commutateurs ET et OU (inclusif), ainsi que de parenthèses en cas d'ambigüité. L'objectif de la tâche est de traiter le plus de données comprises dans l'expression.

Prenons un exemple. Par convention nous notons le type de donnée par une lettre et l'identifiant unique par un chiffre. Soit A,B,C,... des types, et A1,A2,B... des données. La requête (A1 ET B2) OU C3 signifie que la tâche pourra traiter soit les données (A1,B2,C3) soit les données (A1,B2) soit la donnée (C3) mais qu'en aucun cas les données A1 et B2 ne peuvent se traiter de façon séparée. Le Tableau 3 donne des exemples de conditions et les requêtes associées.

Condition	Tampon	Contexte	Requête scheduler
(A ET B) OU (D ET E) OU C	A1, C3	Ø	C3
(A ET B) OU (D ET E) OU C	A1,B2,C3	Ø	(A1 ET B2) OU C3
(A ET B) OU (D ET E) OU C	A1,A5,B2,C3,D4	Ø	((A1 OU A5) ET B2) OU C3
A ET (12h30-16h00)	A1	11h00	Ø
A ET (12h30-16h00)	A1,A2,E3	13h00	A1 OU A2

Tableau 3 : Exemple de conditions et de requêtes

Le scheduler joue un rôle particulier pour la synchronisation. En effet, dans notre modèle, nous avons fait le choix qu'un seul traitement simultané était possible. Ainsi une même tâche ne peut pas être utilisée simultanément par deux processus simultanés. A chaque tâche correspond donc un et un seul processus, ce qui simplifie l'identification des blocages, le calcul des statistiques, la résolution des conflits et plus généralement la compréhension du comportement du gestionnaire. Lorsqu'il souhaite un traitement parallèle, le développeur peut instancier la même tâche à plusieurs reprises. C'est le scheduler qui assure qu'aucune requête ne sera produite tant que celle en cours n'est pas traitée ou refusée.

1.4 LE COORDINATEUR : AUTORISATION DE TRAITEMENT DES DONNEES

A ce stade, la tâche a collecté les données nécessaires à son fonctionnement, vérifié leur pertinence et décidé le traitement d'une partie de ces informations en émettant une requête. Avant de pouvoir utiliser des données, la tâche doit vérifier qu'elle en a l'autorisation. C'est le rôle du troisième module de la tâche : le coordinateur (figure 42).

L'indépendance de chaque tâche est la force du système car elle permet de couvrir un large spectre de situations en créant les conditions d'un comportement opportuniste des tâches. Toutefois elle possède un inconvénient important : **celui des conflits**. A un instant donné, de nombreuses tâches peuvent chercher à s'exécuter indépendamment – apportant ainsi une contribution pertinente au processus d'adaptation – mais aussi parfois de façon redondante voire conflictuelle.

Les conflits sont inhérents aux systèmes événementiels. Dans le cadre de notre architecture, la granularité des blocs constitue un facteur limitant le nombre de conflits potentiels, surtout en comparaison de l'usage de règles. Ainsi un découpage adéquat devrait garantir la résolution d'une grande partie des conflits en interne au sein d'une tâche (lorsqu'elle-même utilise des règles). Cependant, dans des environnements dynamiques et imprévisibles comportant un nombre important de tâches, la concurrence est inévitable.

Il existe des cas dans un système réactif où cette concurrence est souhaitable. Un système répond de nombreuses façons à des stimuli donnés pour adapter l'application. Ainsi, il est possible de créer volontairement un système dans lequel plusieurs tâches entrent en concurrence pour traiter les mêmes informations. A un instant donné, le choix de l'une plutôt que d'une autre dépend alors fortement du contexte et des objectifs fixés par l'administrateur. Par exemple, si les ressources manquent on aura tendance à privilégier les tâches les plus économes au détriment sans doute de l'efficacité. Dans d'autres cas, lorsque l'urgence de la situation l'exige, l'utilisation de stratégies certes plus grossières mais fournissant un résultat plus rapide est souhaitable. Ou encore, l'utilisation de plusieurs stratégies successives, l'une prenant le pas sur la précédente lorsqu'elle défaille, participe à la construction de systèmes très flexibles.

Nous décrirons plus loin, lors de la présentation de l'architecture de contrôle, quelles sont les méthodes possibles permettant la mise en concurrence des tâches. Nous verrons notamment que l'élection des tâches actives résulte soit d'une négociation directe entre les tâches, soit de la décision d'un contrôleur centralisé. Dans tous les cas, il est nécessaire d'associer à chaque tâche un module chargé d'obtenir les autorisations nécessaires et de construire la liste des données qui seront effectivement traitées. C'est le rôle du module que nous appelons coordinateur.

Pour garantir une bonne réactivité du gestionnaire et éviter des coûts superflus en communication, il est impératif que la sélection ne s'opère que lorsque cela est nécessaire. Pour ce faire l'expert configure les coordinateurs en leur fournissant la liste des types de données conflictuels (ou selon le coordinateur un autre discriminant). Ainsi, le coordinateur peut établir la liste des rapports conflictuels d'une requête et construire une requête qu'il négocie. La requête négociée ne porte que sur les données conflictuelles, si bien qu'aucune requête n'est soumise lorsqu'il n'y pas de conflit ; l'exécution suit alors son cours sans perte de temps.

La figure 42 résume la séquence permettant la détermination des rapports à traiter :

1. Le coordinateur utilise la requête du scheduler pour générer la requête négociée. Elle est obtenue très simplement en supprimant les références aux rapports non interdits. Dans notre architecture, ce calcul est générique. Comme la requête du scheduler, la requête négociée est en fait l'expression de l'ensemble des sous-requêtes possibles de par l'utilisation des commutateurs.
2. Le mécanisme de sélection utilise la requête négociée. Nous détaillons plus loin les différents mécanismes qui à partir de cette requête négociée calculent la liste des

données qui pourront être utilisées. Ceux que nous avons utilisés raisonnent essentiellement sur les types de données, la priorité et l'état des tâches. Remarquons déjà que plusieurs stratégies sont possibles : ils peuvent notamment chercher à maximiser le nombre de tâches actives ou maximiser le nombre de rapports traités par les tâches de priorité maximale. Dans ce processus, le coordinateur joue un rôle passif (tel qu'il est décrit ici) ou tient un rôle plus actif en participant à la sélection proprement dite. Avec un coordinateur actif, la sélection devient négociation.

3. A l'issue de la sélection, chaque tâche reçoit la liste des autorisations et un nouvel état (ou le calcule). A partir de cette liste, le coordinateur calcule l'ensemble des rapports à traiter qu'il transmet au processeur.
4. Les données traitées et celles interdites sont supprimées du tampon du scheduler. Celles qui n'ont pas pu être traitées, du fait d'une conjonction avec une donnée interdite, pourront éventuellement l'être dans une prochaine requête si elles n'ont pas expiré.

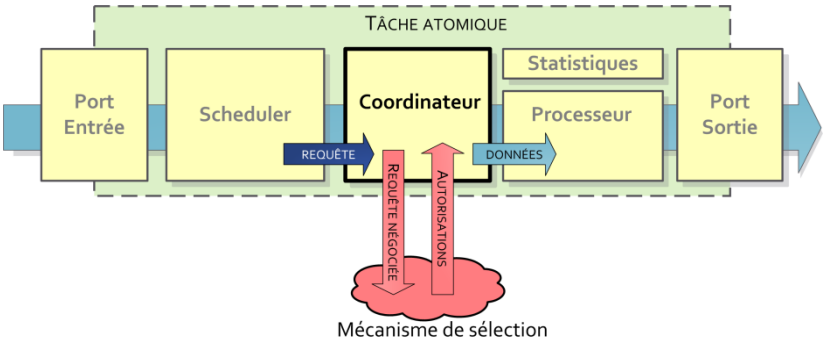

Figure 42 : Le coordinateur détermine les données qui seront finalement traitées

Pour éviter les communications superflues, la requête négociée ne contient pas les données proprement dites mais leur entête (identifiant unique et type essentiellement). En revanche la requête négociée contient des informations sur la tâche (son nom et son état) et peut contenir des méta-informations qui permettent de l'évaluer plus efficacement, par exemple une priorité ou une estimation du temps de traitement. Lorsqu'elles sont utilisées, ces informations sont fournies par le processeur (appel d'une méthode spécifique) puisque fortement dépendant du traitement effectué.

Le Tableau 4 montre des exemples de requêtes négociées, d'autorisations et la liste des données finalement traitées à l'issue de la sélection. Rappelons que par convention nous notons le type de donnée par une lettre et l'identifiant unique par un chiffre. Soit A,B,C,… des types, et A1,A2,B… des données.

Requête (scheduler)	Types conflictuels	Demande négociation	Autorisation reçue	Traitement final
(A1 ET B2) OU C3	A	A1	Ø	C3
(A1 ET B2) OU C3	A	A1	A1	A1, B2, C3
(A1 ET B2) OU C3	A, C	A1 OU C3	C3	C3
(A1 ET B2) OU C3	A, B	A1 ET B2	A1,B2	A1,B2, C3
(A1 ET B2) OU C3	A, B,C	(A1 ET B2) OU C3	A1	Ø

Tableau 4 : Exemple d'obtention d'autorisations et de construction de la liste des données à traiter

Comme nous l'avons expliqué, l'intérêt d'exprimer les requêtes sous la forme d'une expression est de donner une marge de manœuvre au mécanisme de sélection. Il peut de cette manière ne satisfaire que partiellement une requête, là où dans le cas d'un mécanisme plus simple « tout ou rien » il aurait simplement interdit l'activation de la tâche. Les cas de famine où une tâche se trouve systématiquement privée d'exécution sont de ce fait plus rares. Ce mécanisme donne une flexibilité au développeur qui peut dans les cas simples fonctionner en « tout ou rien » – fournissant une liste de rapports qui sont interprétés comme tous obligatoires – et dans les cas plus complexes donner plus de latitude à la négociation.

1.5 LE PROCESSEUR : FONCTIONNALITE ET ROLE DE LA TACHE

Dans les précédentes étapes nous avons vu comment les données sont collectées, filtrées et sélectionnées. A présent, le mécanisme de sélection a déterminé que la tâche pouvait les utiliser. Au cœur de la tâche, c'est le processeur qui réalise les traitements.

Jusqu'alors, l'essentiel des modules que nous avons définis sont indépendants du traitement effectif de la tâche. Le développeur réutilise à loisir des ports, schedulers ou coordinateurs qui correspondent à ses besoins pour construire ses tâches. Dans notre architecture, le processeur est la partie la plus spécifique. Il est responsable du traitement des données et assiste parfois, comme nous l'avons signalé plus haut, le scheduler et le coordinateur en affinant le filtrage ou en fournissant des estimations sur le temps de traitement.

Le processeur est avant tout spécialisé dans la production et la manipulation des données. Il ne se charge pas de leur routage et n'a aucune connaissance sur la provenance ou la destination des messages. Il reçoit un ensemble de données qu'il traite et il produit à son tour un ensemble de résultats. Le processeur n'indique ni les sources de données, ni les destinations ; il se consacre entièrement au traitement. La séparation forte des préoccupations permet ici au développeur de se focaliser sur le développement de l'algorithme proprement dit sans se préoccuper des conditions de son utilisation.

L'action du processeur est bien supérieure à une simple règle qui, prise séparément, ne fait pas sens. Alors que les systèmes experts fonctionnent en déclenchant des règles en réponse à des stimuli, notre système fonctionne en déclenchant des modules complets qui peuvent eux-mêmes être composés de règles, mais également des routines de logique floues, un réseau de neurones ou de simples procédures Le rôle du processeur est par exemple de récupérer l'utilisation mémoire dans le système, de détecter une panne sur un serveur, de proposer une solution à cette panne ou d'installer un nouveau serveur. Le regroupement des fonctions, l'opération réalisée par la tâche, doit avoir un sens pour l'administrateur et sur le plan métier.

Si l'on se réfère à ce que nous avons présenté dans l'état de l'art, et en particulier à l'architecture de référence proposée par IBM, plusieurs rôles se dessinent pour les tâches. L'essentiel des tâches assument les quatre rôles de la boucle MAPE (figure 43) :

- les **tâches de collecte** qui remplissent le rôle M de la boucle MAPE. Elles n'ont a priori pas de port d'entrée : le développeur implémente les capacités de récupération des données en s'appuyant sur les sondes de l'environnement et de l'application. Elles peuvent en revanche s'appuyer sur le scheduler pour être réveillées à intervalle de temps régulier.
- les **tâches de prise de décision** qui correspondent aux rôles des blocs « analyse » et « planification » de la boucle MAPE. Ces tâches se basent sur les données qu'elles reçoivent et produisent des informations nouvelles qui font avancer le processus d'adaptation.
- les **tâches d'exécution** qui s'appuient sur les effecteurs de l'application et de l'environnement pour adapter l'application. Elles ne possèdent pas a priori de port de sortie.

En plus de ces grandes catégories, d'autres tâches jouent un rôle de facilitation et de coordination de l'action. Ce sont :

- les **tâches de médiation** qui transforment les données produites sans en modifier le sens. Elles permettent la communication entre des tâches qui utilisent une représentation des données différentes et facilitent ainsi leur réutilisation.
- les **tâches de synthèse** qui sont utilisées lorsque des tâches fonctionnent en concurrence et qu'il faut trouver un compromis. C'est un moyen simple d'assurer un comportement du gestionnaire globalement cohérent. Elles sont indispensables lorsque les conflits ne peuvent être résolus qu'après l'activation d'une tâche ou pour comparer deux solutions.

Parfois, lorsque le développeur le souhaite et que l'activité de la tâche est clairement définie, cette dernière réalise une boucle complète d'adaptation ou de large portion de cette boucle comme le montre la figure 43. Il y a un compromis à trouver entre la grande flexibilité – résultant du découpage – et les difficultés d'assemblage et les coûts en performances liés à une fragmentation trop forte. Le découpage doit se justifier par le souhait du développeur de réutiliser la tâche dans une autre situation ou de la mettre en concurrence avec d'autres implémentations ou solutions.

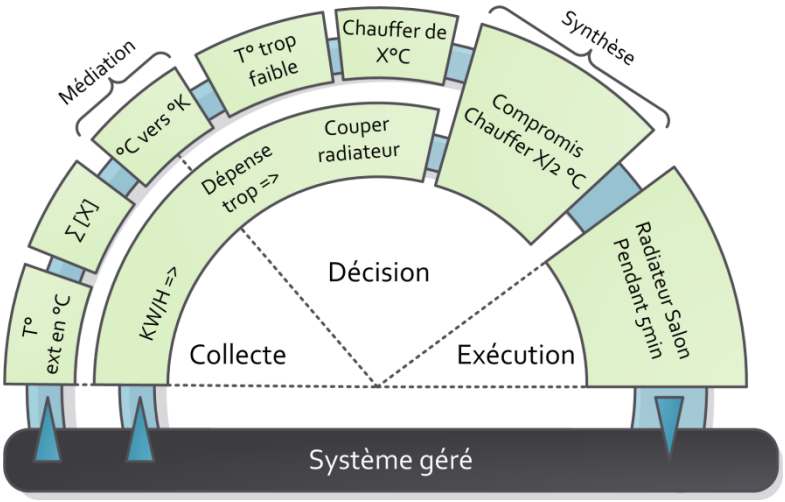

Figure 43 : Exemple de tâches

L'algorithme implémenté par le processeur est relativement indépendant des autres tâches. Il dépend bien évidement des données mais il n'a pas de dépendance fonctionnelle vis-à-vis des autres tâches. En particulier, il n'a aucune connaissance sur le nombre, l'implémentation et même l'objectif des tâches qui lui ont fourni les données. L'assembleur du gestionnaire et les autres tâches n'ont pas à se soucier de la façon dont l'opération est réalisée et encore moins de la technologie utilisée. Pour autant, comme pour toute implémentation, des dépendances fonctionnelles peuvent exister, comme des dépendances vers des librairies ou des services déployés sur la plateforme d'exécution ou fournis par elle.

Le processeur n'est pas nécessairement implémenté d'un seul tenant. Il est même souhaitable que l'algorithme soit construit sur la base de briques et de logiciels existants et éprouvés. Pour ce faire, soit la tâche embarque, encapsule et masque ces outils, soit elle est simplement une passerelle vers cette solution existante. Dans tous les cas, la complexité est dissimulée à l'expert qui assemble et choisit les tâches du gestionnaire.

Plusieurs tâches sont susceptibles d'implémenter très différemment la même fonctionnalité. Techniquement d'abord en utilisant du code, des règles ou en s'appuyant sur un

logiciel existant, comme un planificateur. Considérons, pour illustrer, une tâche de planification chargée de trouver une solution à un problème. Il existe à disposition du développeur un grand nombre d'algorithmes de planification tels que STRIPS [63], en partant du but, ou en partant de l'état initial. Certains de ces algorithmes finissent en un temps connu ou un nombre d'étapes limité, d'autres non. L'utilisation de tel ou tel algorithme a des conséquences sur le temps d'analyse, la qualité du résultat et les ressources consommées, si bien qu'il est pertinent de vouloir en utiliser plusieurs dans différentes tâches pour pouvoir en changer en fonction du contexte et des besoins.

La figure 44 représente un processeur en activité. Le processeur s'appuie sur trois flux de données.

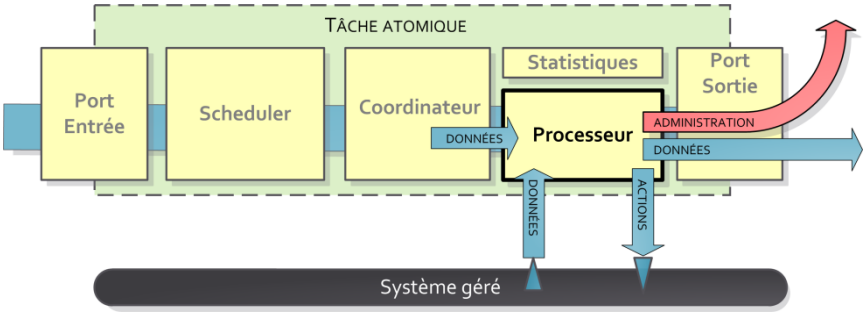

Figure 44 : Le processeur produit des données et des messages d'administrations

Le premier, celui sur lequel nous sommes attardés pour l'instant, est constitué des données fournies par le scheduler et produites par le processeur. Comme nous l'avons vu, une partie des données que le processeur utilise lui ont été transmisses par les autres tâches parce qu'il y est abonné. Ces données sont standardisées et leur représentation est décrite dans un type de donnée. Elles sont utilisables par d'autres tâches, qui les consultent en parallèle s'il n'y a pas de conflit, ou par les couches supérieures du framework pour évaluer l'activité du gestionnaire. Des mécanismes que fournit le framework assurent la diffusion des données et le processeur contribue à ce flux en n'en transmettant à son port de sortie.

Il existe un second flux d'exécution entre les sondes ou effecteurs du système administré et le processeur. C'est sur lui que se basent les moniteurs et exécuteurs pour collecter les informations et agir sur le système. Les échanges entre ces touchpoints et le processeur ne sont pas standards ; ils restent propres à l'algorithme et à l'implémentation de ce dernier. Les moyens de communication utilisés sont variés : socket, JMX [91], web services, appel direct de services. Cette partie spécifique doit être prise en charge par le processeur.

Le dernier flux, non représenté ici, est composé de l'ensemble des données écrites et lues par l'ensemble des processeurs dans une base de données commune. Celle-ci correspond au bloc K de l'architecture MAPE-K et permet aux tâches d'échanger des informations persistantes (activation d'une alarme) : elle évite ainsi un trop grand nombre de communications. Les données déposées et collectées dans la base sont typées.

A l'issue du traitement, les processeurs produisent deux types d'informations (figure 44) :

- **les données** qui sont destinées à être utilisées par d'autres tâches. Elles contiennent des informations pertinentes pour d'autres tâches qui vont permettre de faire avancer le processus d'adaptation. Comme nous l'avons précisé lors de la présentation des ports, ces données sont typées et de représentations extrêmement variées.
- **des messages d'administration** à destination des couches supérieures du framework et qui fournissent des informations sur le fonctionnement de la tâche et ses besoins par rapport à son environnement. Par exemple une tâche qui estime

recevoir trop peu d'informations peut demander que la fréquence de production soit augmentée. Ces messages contribuent à l'évaluation de la configuration et la politique d'agencement des tâches. Ces messages d'administration sont ainsi utilisés par les couches supérieures et par l'administrateur pour reconfigurer les tâches de façon à optimiser le fonctionnement du gestionnaire.

L'exécution de l'algorithme implémenté par le processeur est soumise à deux conditions qui doivent être garanties par l'implémentation du framework. Il faut d'une part assurer qu'au cours de l'exécution, la tâche sera capable de recevoir et stocker des données en provenance des autres tâches. L'exécution d'une tâche n'entrave pas la communication et ne lui fait pas perdre de données pertinentes. A l'issue du traitement, les conditions de déclenchement du scheduler évalueront ces données – si elles n'ont pas expiré – pour lancer un nouveau traitement éventuel. D'autre part, le framework doit prévenir les couches supérieures en cas de blocage manifeste de la tâche de sorte que cette dernière puisse prendre une décision : attendre, stopper la tâche ou la détruire. Un processeur est considéré comme bloqué lorsqu'un délai d'exécution jugé maximal a expiré. Celui-ci est estimé par un expert et fixé lors de la configuration de la tâche. Naturellement ce délai peut être infini lorsque l'expert estime que la durée d'exécution ne peut se borner.

A chaque appel au processeur, un module de statistique (figure 44) s'exécute. Il calcule et fournit des informations sur l'activité de la tâche. Elles permettent d'assister les couches supérieures et les mécanismes de sélection dans l'évaluation de l'efficacité des tâches en cours d'exécution. L'architecture de contrôle ou l'administrateur peut alors décider de remplacer des tâches ou de modifier les mécanismes de sélection – en jouant par exemple sur la priorité d'une tâche – pour privilégier l'utilisation de certaines tâches plutôt que d'autres. Les informations collectées par l'algorithme de statistique sont variées et le développeur peut à loisir réécrire l'algorithme de statistique de base fourni par le framework. Par défaut, ce dernier calcule :

- le nombre d'exécutions de la tâche ;
- la quantité et le type de données traitées ;
- la quantité et le type de données produites ;
- le temps moyen d'exécution de la tâche ;
- le nombre de fois où la tâche a été considérée comme bloquée.

Pour pouvoir calculer ces statistiques, il existe deux méthodes. Soit l'ensemble des données transitent d'abord par le module de statistiques qui se charge alors de faire appel à la méthode de traitement du processeur. Il est ainsi très facile de calculer les informations. Soit le module utilise la programmation par aspect et détecte l'appel de la méthode, ce qui favorise un découplage plus fort. Ces informations ont l'avantage d'être très peu couteuses à calculer. Toutefois, pour gagner en performance, ce module de statistique est débrayable.

Enfin, comme nous l'avons signalé lors de la présentation des données, un historique est calculé. C'est le module chargé d'appeler le processeur qui ajoute à chaque donnée produite les informations permettant de reconstruire l'historique. Ces informations permettent à l'administrateur et à l'architecture de contrôle d'évaluer le processus d'adaptation sur des cycles complets. Ainsi, à partir d'une donnée, il est possible de déterminer le cheminement qui a permis de l'obtenir. Cependant cette information ne peut être qu'approximative et coute potentiellement de l'espace mémoire. C'est pourquoi, tout comme les statistiques, l'historique n'est calculé que lorsque l'expert le décide.

1.6 INTERFACES D'ADMINISTRATION, CYCLE DE VIE ET CONFIGURATION DE LA TACHE

Nous venons donc de décrire les cinq étapes permettant à une tâche de sélectionner et de traiter des données pour prendre part à une boucle de contrôle. Durant cette présentation, nous avons toujours considéré que la tâche était active. Or une tâche connaît de nombreux états. Nous nous intéressons ici plus en détail au cycle de vie de la tâche et aux informations nécessaires à sa configuration.

On adjoint à chaque tâche un module d'administration (figure 45) qui se charge de stocker son état, sa configuration et offre une interface d'administration unifiée à l'architecture de contrôle. Il ne se consacre qu'à la tâche à laquelle il appartient et ne doit pas être confondu avec le module d'administration des tâches de l'infrastructure de contrôle présenté plus loin. C'est le premier module créé lors de l'instanciation d'une tâche. Il se charge de la création et de la configuration des autres sous-modules.

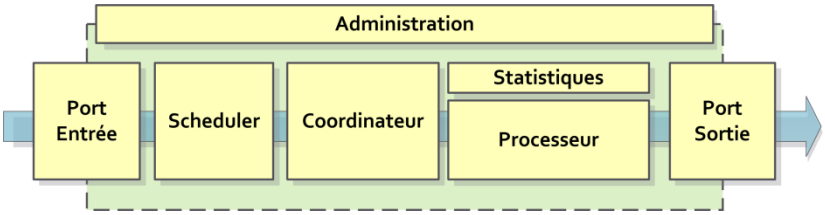

Figure 45 : Administration de la tâche

Au démarrage du gestionnaire, les couches supérieures se chargent de trouver les implémentations des tâches nécessaires au gestionnaire. Elles instancient alors les tâches. Une fois créée, une instance de tâche passe par quatre états principaux (figure 46) :

- **configurée** : l'ensemble des sous-modules de la tâche ont été créés et la tâche a reçu sa configuration initiale. La tâche n'a pas de relation avec l'extérieur et en particulier ne communique aucune information sur la nature des données qu'elle produit et utilise.
- **invalide** : la tâche est démarrée et les ports d'entrée utilisent le mécanisme de communication pour découvrir les types de données disponibles. Dans cet état, au moins une dépendance de données n'est pas satisfaite et la tâche n'informe pas les autres tâches sur la nature des données qu'elle produit.
- **valide** : l'ensemble des dépendances de données sont satisfaites. La tâche fait connaître au mécanisme de communication les données qu'elle produit et qu'elle utilise. Dans cet état, elle collecte et produit des données.
- **détruite** : une tâche est détruite par l'administrateur ou la couche supérieure lorsqu'elle ne répond plus au besoin du gestionnaire.

Nous avons entrevu lors de la présentation du processeur qu'une tâche dans l'état valide pouvait se trouver dans des situations différentes. On considère ainsi que la tâche est :

- **en attente** lorsqu'aucune requête n'est en cours de traitement. Le scheduler et le coordinateur n'ont pas encore déterminé que la tâche pouvait s'activer.
- **active** dès qu'une requête lui est transmise par le coordinateur**.** L'algorithme du processeur est alors exécuté pendant que la tâche continue à collecter des données.
- **bloquée :** lorsque l'activité de la tâche est jugée suspecte. Pour l'heure, nous avons choisi de déterminer si une tâche est bloquée en fonction d'un délai maximal d'exécution proposé par l'expert comme nous l'avons évoqué dans la description du processeur.

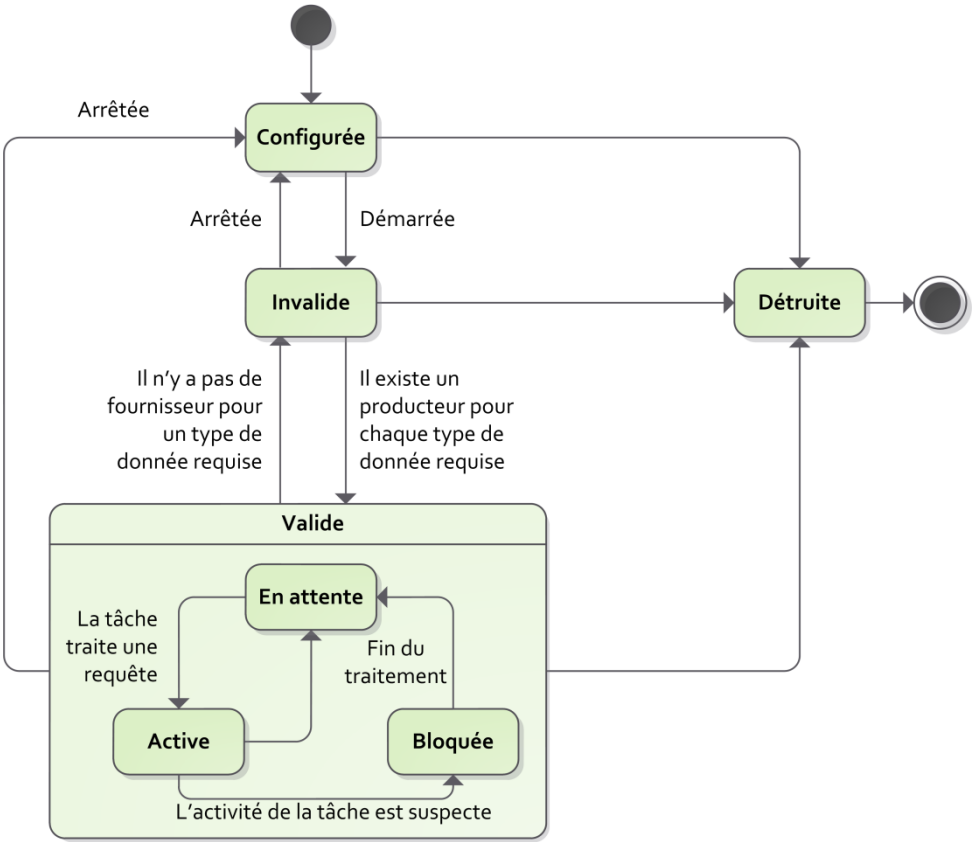

Figure 46 : Les états d'une tâche

Comme le montre la figure 47, l'état d'une tâche est complété par un ensemble de méta-informations. Elles sont utilisées et déposées par le mécanisme de sélection en cours pour récupérer et garder des indications permettant de guider les prochains choix. Par exemple, selon le mécanisme de sélection choisi, l'état peut être complété par une priorité, la description d'une inhibition, la quantité d'exécutions disponibles restantes. Ces valeurs ne font pas partie de la configuration de la tâche et sont déterminées par la configuration du mécanisme de sélection. Une valeur initiale leur est donnée, soit à la première requête soumise, soit à l'initiative du mécanisme de sélection. Ces informations sont réinitialisées si un changement intervient dans la logique du mécanisme de sélection.

La configuration d'une tâche se compose d'un ensemble de propriétés de configuration (figure 47). A chaque modification de l'une d'elles, le processeur est averti : il peut alors changer son comportement. Outre ces propriétés s'ajoute la configuration des autres sous-modules. Les ports d'entrée et sortie reçoivent la liste des types à produire et collecter ; le scheduler reçoit la configuration des conditions de déclenchement et le coordinateur reçoit la liste des types de données interdits.

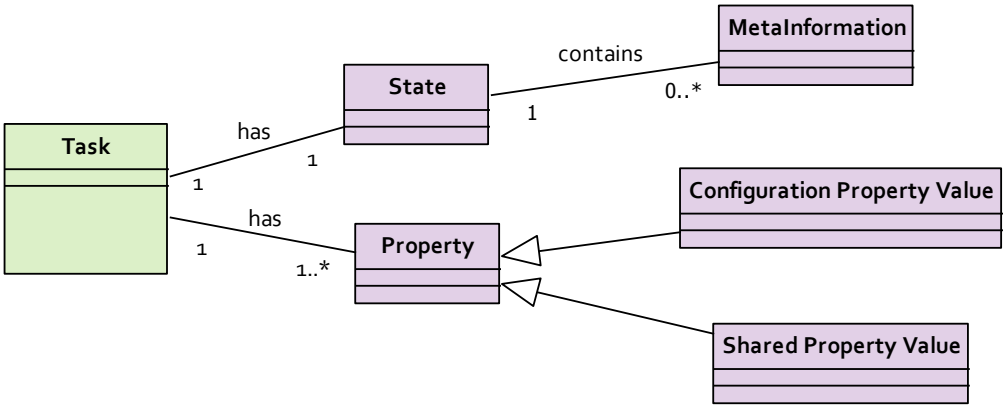

Figure 47 : Etat et configuration d'une tâche

Enfin, la figure 47 montre des propriétés dites partagées. Il s'agit de valeurs partagées avec d'autres tâches. Elles sont stockées dans la base de données commune aux tâches. Ces propriétés sont typées et peuvent être utilisées soit pour partager soit une configuration soit des données entre les tâches comme nous l'évoquions dans la description du processeur. La base de données est un service fourni par l'architecture de contrôle, nous la présentons plus loin.

1.7 NOTION DE TYPE DE TACHES

Nous avons introduit dans cette première partie deux notions classiques : l'implémentation et l'instance de tâches. L'implémentation d'une tâche défini l'implémentation de chacun des sous-modules et leur combinaison. Le concept d'implémentation correspond à celui de classe ou de type de composant. Une même implémentation de tâche peut avoir plusieurs instances. Le concept d'instance correspond à celui d'instance objet dans la programmation objet ou d'instance de composant. Le cycle de vie que nous avons décrit plus haut se réfère aux instances de tâche. Le framework pourrait être utilisé tel quel sans introduire de notion supplémentaire.

Toutefois, nous l'avons signalé pour l'implémentation du processeur, il existe plusieurs façons de réaliser une fonctionnalité donnée. Par exemple en JAVA, pour implémenter une tâche qui collecte la valeur du processeur, il est possible de se baser sur différentes API : avec ou sans DLL, spécifiques ou non à un système d'exploitation, plus ou moins précises. Ceci est lié à l'évolution des systèmes d'exploitation et des processeurs (double cœur, etc …). Lorsqu'il utilise cette tâche, l'administrateur ou l'expert qui construit le gestionnaire ne devrait pas se soucier des détails de l'implémentation. Le passage d'une implémentation à une autre devrait se faire de la façon la plus transparente possible (éventuellement automatiquement). L'expert doit pouvoir définir l'ensemble de tâches utilisées par le gestionnaire en termes de fonctionnalité et non en termes d'implémentation.

Au même titre que les services possèdent une spécification, il nous parait donc intéressant d'introduire une notion de type (ou spécification) de tâche afin de standardiser l'usage, la configuration et les données utilisées et produites par les catégories de tâches qui accomplissent les mêmes fonctionnalités. Chaque tâche implémente une spécification qui décrit son utilisation et l'opération qu'elle réalise. Un type de tâche contraint (figure 48) :

- **les entrées et les sorties** : les types de données utilisées en entrée et sortie de la tâche sont décrits. Il existe deux types de dépendances.
- **les propriétés de configuration** qui permettent de modifier le comportement du processeur. Chaque modification d'une de ces propriétés est signifiée au processeur.

- **les propriétés partagées** qui sont les propriétés stockées dans la base de données et partagées avec les autres bases.

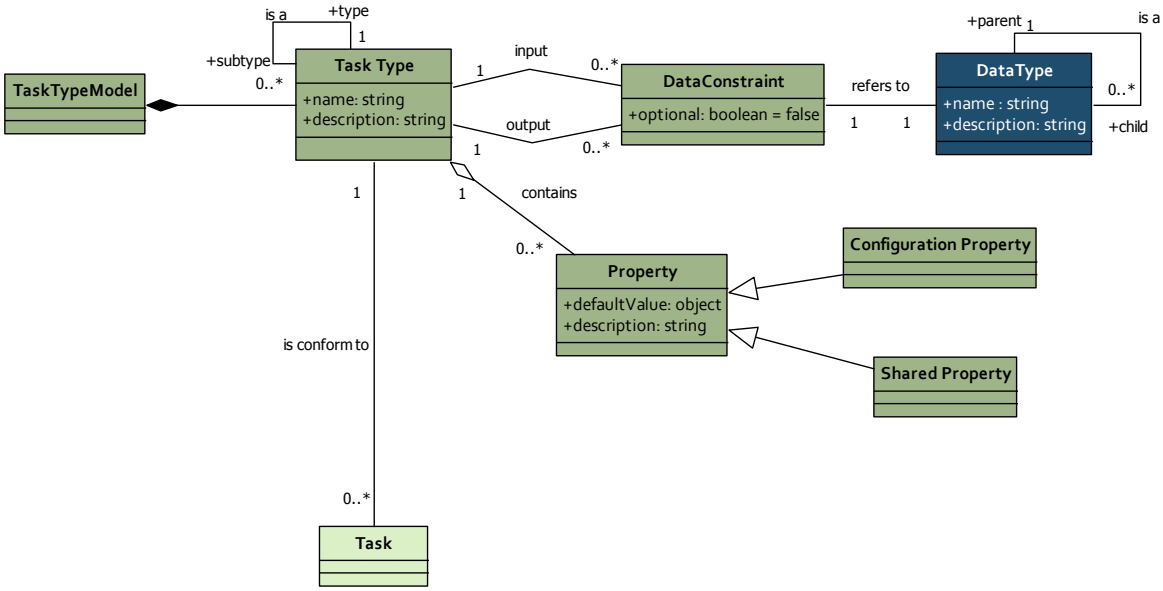

Figure 48 : Type de tâche

Pour être conforme à son type, une tâche doit proposer l'intégralité des propriétés de configuration et des propriétés partagées ; elle doit également respecter les contraintes d'entrée et sortie. Une contrainte sur les types d'entrée ou de sortie peut être obligatoire ou optionnelle.

Par défaut elles sont obligatoires, ce qui signifie que toutes les implémentations de la tâche devront déclarer et produire les données des types déclarés pour être conformes. En revanche, la tâche peut ne pas traiter ou ne produire que des sous-types des types définis par les dépendances optionnelles.

L'expert peut décider d'affaiblir une contrainte pour un type de donnée particulier en la rendant optionnelle. Dans ce cas, la tâche a le choix de la traiter ou non et peut changer d'avis en cours d'exécution. Les contraintes optionnelles sont surtout utilisées pour donner de la lattitude dans l'implémentation des tâches composites. Ces dernières peuvent alors choisir d'ignorer, pour les filtrer, certains types de donnée.

Tout type de donnée qui n'est pas défini par une contrainte (obligatoire ou optionnelle) ne peut être produit ou utilisé par la tâche. De cette façon, le type contraint fortement l'usage de la tâche et force le développeur à en stabiliser le comportement.

Comme pour les données, il est difficile d'introduire des contraintes sémantiques. La fonctionnalité est donnée de façon informelle dans une description. Pour être conforme, la tâche doit respecter cette description. De même, on associe à chaque propriété une description de la façon de l'utiliser et le type de donnée contenu.

Lorsque l'administrateur ou les couches supérieures choisissent une tâche, ils peuvent donc faire référence soit à son type, soit directement à sa spécification. Cette décision peut être automatisée, le choix d'une implémentation plutôt qu'une autre se fait alors sur la base de méta-informations ajoutées à chaque implémentation. Par exemple, dans notre implémentation nous

offrons la possibilité à l'expert de définir un service spécifique qui possède une méthode permettant d'effectuer ce choix.

En sus de cette spécification nous introduisons une classification entre les types. Comme illustré par la figure 48, un type peut avoir plusieurs enfants et chaque enfant n'a qu'un parent. Les sous-types doivent respecter les règles suivantes :

- les contraintes de données fortes sont les mêmes. Il n'est pas possible de les renforcer.
- les contraintes de données optionnelles peuvent devenir fortes. Tous les sous-types de donnée concernés par une contrainte faible peuvent appartenir à une nouvelle contrainte forte.
- il n'est pas possible de rajouter d'autres contraintes
- l'ensemble des propriétés de configuration et l'ensemble des propriétés partagées du père sont présentes.
- il est possible de rajouter de nouvelles propriétés.
- la tâche respecte le contrat informel défini dans la description du père.

Nous introduisons cette classification pour deux raisons.

La première est avant tout qu'elle permet d'identifier les groupes de tâches réalisant des opérations similaires. Par exemple, l'ensemble des tâches de collecte peuvent appartenir au type MONITEUR. Il est alors plus aisé de comprendre la structure d'un gestionnaire.

La seconde est que la définition de groupe permet d'exprimer des conflits entre des types de tâches et non entre implémentations de tâches. L'ajout d'une nouvelle tâche à une catégorie est ainsi plus aisé. Admettons par exemple que nous avons n tâches de même fonctionnalité en conflit parce qu'elles réalisent la même fonctionnalité. Sans type, ni classification, il faudrait indiquer tout les couples en conflit, soit $\sum_{i=1}^{i=n-1} i$ règles. Par exemple pour A,B,C,D on obtient (A,B), (A,C), (A,D), (B,C), (B,D),(C,D). Dans ce cas, plus le nombre de tâches est important et plus les conflits sont nombreux, plus le nombre de règles conséquent. Mais le principal problème vient de l'installation d'une nouvelle tâche, qui oblige systématiquement l'ajout d'une règle. Ajouter une tâche nécessite d'écrire (n-1) règles supplémentaires. En revanche, avec un type il suffit d'une règle indiquant « les tâches de type A sont en conflit ». En outre, la classification vient créer encore plus de souplesse en définissant des groupes de taille variable. La gestion des conflits s'en trouvent facilitée.

Nous définissons une règle de substituabilité très simple et stricte : deux tâches sont substituables si elles sont exactement du même type (pas un sous-type ou un type parent). Cette règle est suffisante pour la majorité des situations. Il serait possible en définissant des contraintes et des contrôles adaptés d'introduire du polymorphisme dans la classification ; le développeur pourrait alors utiliser et considérer une tâche d'un sous-type comme une tâche du type parent. Cependant, cette définition est rendue complexe par la présence des contraintes optionnelles et ne nous apparaît pas comme indispensable. Ici encore, l'objectif n'est pas de proposer un typage complet et complexe mais de simplifier la tâche du développeur.

2 CONTROLE DES TACHES ET DU GESTIONNAIRE

2.1 STRUCTURE GENERALE

Dans la section précédente, nous avons présenté le concept de tâche qui est au cœur de notre approche et proposé une architecture. Les tâches sont directement liées aux systèmes gérés et leur combinaison réalise une ou plusieurs boucles d'adaptation qui permettent de l'administrer. Le concept de tâche ne suffit pas. Il faut maintenant définir une architecture qui permet de gérer le pool de tâches et d'organiser leur action. C'est l'architecture de contrôle (Figure 49) qui assure la gestion des tâches et que nous présentons ici.

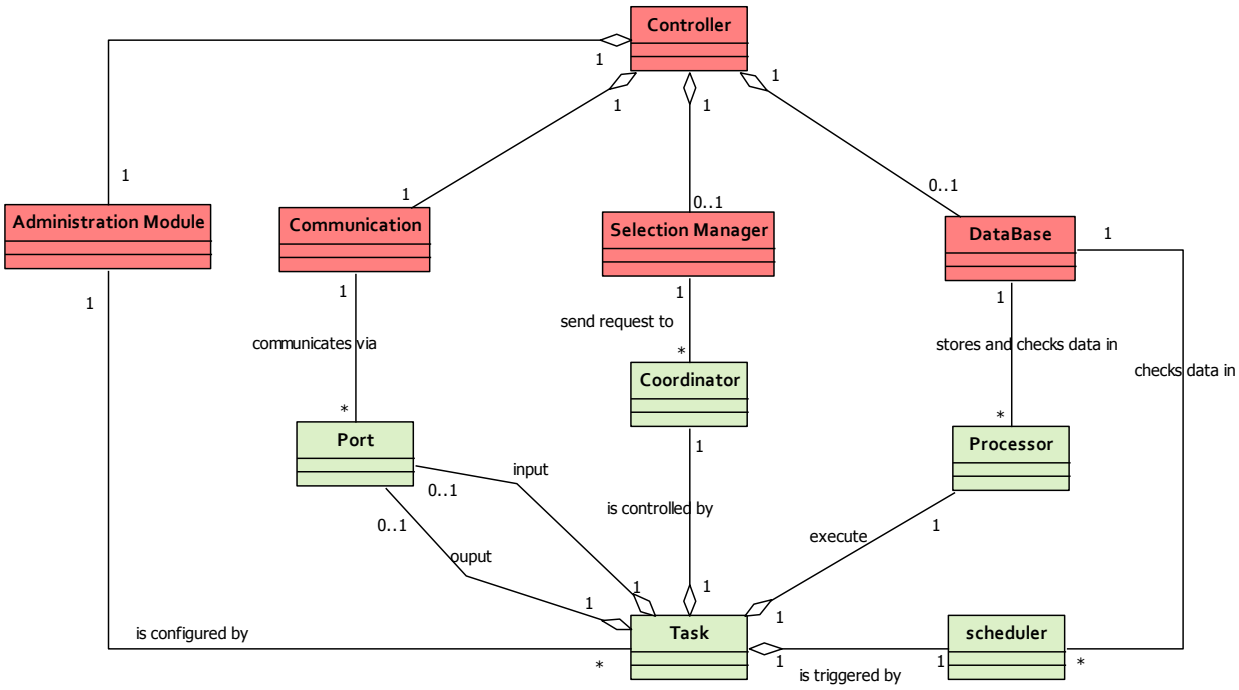

Figure 49 : Architecture de contrôle

Pour pouvoir fonctionner et former un gestionnaire autonomique, les tâches s'appuient sur cette infrastructure de contrôle. Cette dernière repose sur un contrôleur constitué de sous-modules spécialisés (figure 49) :

- **le module d'administration** qui est chargé de la découverte des implémentations et de la création, destruction, configuration et instanciation des tâches. Ce mécanisme construit un modèle de l'exécution des tâches qui permet l'observation du comportement du gestionnaire. Il ne doit pas être confondu avec les modules d'administration des tâches. A la différence de ces derniers, il gère non pas une seule tâche, mais l'ensemble des tâches.
- **le support de communication** qui permet aux tâches d'échanger les informations. C'est sur lui que s'appuient les ports de communication des tâches pour échanger les données. Nous verrons que sa nature peut varier : bus ou mécanisme de découverte.
- **le mécanisme de sélection** qui détermine en cas de conflit quelles tâches doivent être activées. Les coordinateurs de chaque tâche communiquent avec lui et lui soumettent leur requête.

- **la base de données** qui offre aux tâches la possibilité de partager des informations avec d'autres tâches. Elle peut notamment servir au stockage et au maintien d'un historique de traitement. L'algorithme de sélection peut également s'appuyer dessus pour trouver des informations contextuelles.

Au lancement du gestionnaire, le contrôleur est responsable de la création et du bon fonctionnement de chacun de ces mécanismes. Il fournit une interface unifiée d'observation et de configuration du pool de tâches qu'il supervise.

La figure montre que seuls deux de ces mécanismes sont obligatoirement présents : le gestionnaire de communication et le module d'administration.

Le gestionnaire de sélection n'est pas indispensable. Dans la plupart des cas, le développeur peut s'en priver en choisissant un découpage adéquat des différentes activités. Ainsi, les gestionnaires simples ou les premières versions de gestionnaires plus complexes ne présentent souvent pas de conflits.

Nous verrons que le stockage d'informations partagées est une facilité fournie au développeur. Les premières versions du framework n'en disposaient pas mais la pratique a montré que posséder une méthode standard d'échange et de maintien d'information simplifiait la tâche du développeur. Certaines situations comme la présence d'événements ponctuels appellent naturellement à l'usage d'une base centralisée. C'est le K de l'architecture MAPE-K d'IBM.

Contrairement aux sous-modules de la tâche, les modules du contrôleur sont relativement indépendants. Il n'y a pas de flux d'exécution séquentiel à travers eux. Toutefois, ces modules peuvent s'appuyer sur les mécanismes de communication (échange des messages d'administration) et sur la base de données (par exemple pour affiner la sélection). Du point de vue de la tâche, il existe un ordre d'appel de ces modules :

- le **module d'administration** crée et configure les tâches ;
- les ports de la tâche utilisent ensuite le **mécanisme de communication** pour échanger des données ;
- les coordinateurs de chaque tâche s'appuient sur le **mécanisme de sélection** pour soumettre (ou négocier) leurs requêtes ;
- le processeur dépose et collecte des données dans la **base de données** pour les échanger avec d'autres tâches.

Dans la suite nous présentons les différents modules du contrôleur dans cet ordre.

2.2 ADMINISTRATION DE L'ENSEMBLE DE TACHES : CYCLE DE VIE ET CONFIGURATION

Le premier rôle de l'architecture de contrôle est de conférer aux couches supérieures la possibilité d'observer et modifier la configuration du gestionnaire. Au sein du contrôleur cette activité est dédiée au module d'administration.

Le module d'administration propose un ensemble de procédures permettant la création, la configuration, et la destruction des tâches, la modification du mécanisme de sélection ainsi que l'observation de l'ensemble de tâches (figure 50).

Récapitulons les étapes et les informations nécessaires à ces quatre primitives de base que sont la création, la modification, la destruction et l'observation.

Création

Pour créer une tâche, l'administrateur ou la couche supérieure fournissent le nom de l'implémentation souhaitée. Le module d'administration se charge d'instancier la tâche et de lui fournir la configuration initiale désirée. Cette dernière est en partie détaillée par la description du

type contenant les dépendances et la valeur par défaut des propriétés. Elle s'accompagne de la déclaration de la configuration du scheduler et du coordinateur comme il suit.

Une majeure partie de la configuration du scheduler est décrite par l'implémentation. Certains paramètres de la configuration sont en effet liés à la façon dont le processeur traite les données – par exemple le paramétrage des conditions de déclenchement sur les données – et ils sont par conséquent fournis par le développeur. D'autres, tels que l'heure d'activation ou le délai d'attente entre deux activations, peuvent être modifiés à la création et durant l'exécution. Cette configuration peut se baser sur des propriétés, mais devant la variété des configurations possibles pour les schedulers, nous suggérons l'emploi d'un langage balisé comme XML que nous utilisons dans l'implémentation.

La configuration du coordinateur dépend du mécanisme de sélection choisi. Nous verrons dans la section suivante que cette configuration peut se contenter de la liste des rapports interdits (arbitrage) mais peut également concerner la configuration de la sélection proprement dite (négociation). Dans ce cas également, l'utilisation d'un langage balisé nous paraît intéressante.

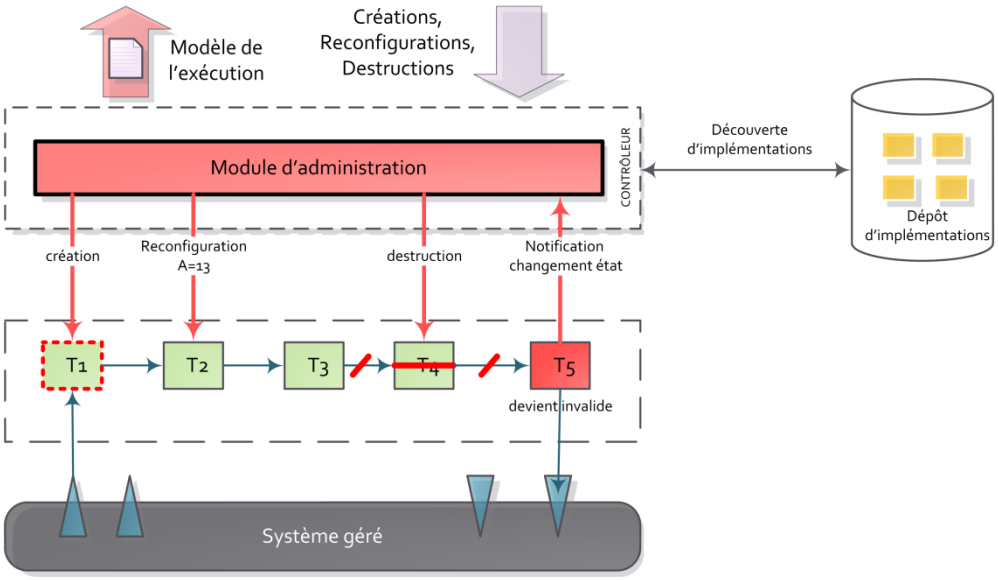

Figure 50 : Module d'administration

La création de la tâche échoue parfois lorsque les dépendances fonctionnelles ne sont pas satisfaites. C'est le cas lorsque le processeur de la tâche dépend de librairies, de services, de communications réseau qui ne sont pas présents sur le système. La tâche est alors dans l'état invalide. La couche supérieure, qui s'occupe de l'administration et de l'adaptation du gestionnaire, est informée de cette invalidité. Pour faire face à ces situations, le module d'administration peut être configuré pour choisir l'implémentation suivante sur la liste fournie par le service de configuration, ou simplement attendre que les dépendances fonctionnelles soient résolues.

Une fois la tâche instanciée et configurée, le module d'administration est notifié par les ports d'entrée de la tâche si les dépendances de données ne sont pas satisfaites. Dans ce cas encore, il joue le rôle de relai et avertit l'administrateur ou la couche supérieure. Cette invalidité peut être temporaire, c'est à la couche supérieur de proposer une solution.

Chaque tâche créée est pourvue d'un identifiant unique qui permet de l'identifier de façon unique dans le système. C'est le module d'administration qui garantit cette unicité.

Modification

Le module d'administration sert d'indirection entre les tâches et la couche supérieure en fournissant des interfaces permettant la modification de la configuration des tâches. C'est une couche d'abstraction qui permet de masquer la complexité des technologies sous-jacentes. Toute demande de modification doit faire référence à l'identifiant unique de la tâche.

Une modification particulièrement importante dans notre système concerne la configuration du mécanisme de sélection (non représenté figure 50). La faire évoluer permet de changer le comportement général du gestionnaire en modifiant la façon dont les tâches se concurrencent. Ici encore, du fait des nombreuses configurations possibles, un langage balisé nous paraît être un choix adéquat pour exprimer la configuration de l'algorithme d'élection, qu'il s'agisse de filtrage, d'arbitrage ou de négociation, qui sont les méthodes que nous présentons dans la section suivante. Nous donnerons à cette occasion des exemples de configuration.

La configuration des ports des tâches est possible, pourvu qu'elle respecte le type de la tâche en continuant à fournir et réclamer les types obligatoires. Cette modification prend tout son sens avec les composites qui filtrent alors certains messages pour un groupe de tâches comme nous le verrons plus loin.

Destruction

Du point de vue du gestionnaire, la destruction d'une tâche a plusieurs origines :

- **ordre direct** : lorsque les couches supérieures prennent la décision de détruire la tâche (pour ce faire, il suffit d'indiquer l'identifiant unique de la tâche à détruire).
- **disparition de l'implémentation** : lorsqu'une implémentation disparaît il est nécessaire de détruire les tâches associées.

La disparition de l'implémentation est un des rares cas de modification par le bas, c'est-à-dire ne provenant pas d'un ordre direct de la couche d'adaptation. Par exemple, sur une plateforme de type OSGi, elle survient lorsque l'administrateur décide de désinstaller un bundle contenant l'implémentation des tâches ou en cas d'erreur. Pour cette raison, le module d'administration doit informer la couche d'adaptation de sorte qu'elle puisse prendre les mesures adéquates, en choisissant par exemple une nouvelle implémentation.

La destruction d'une tâche a des effets sur les autres tâches. Dans notre exemple, figure 50, la suppression de la tâche T4 entraine l'invalidité de la tâche T5 dont les dépendances de données ne sont plus satisfaites.

Observation

Le module d'administration fournit les interfaces permettant d'observer le comportement du gestionnaire. Il fournit pour cela une description de l'activité des tâches en exécution. Il construit et maintient à jour un modèle de l'architecture en exécution. Celui-ci fournit une représentation fidèle de l'état et de la configuration des tâches et du contrôleur à un instant donné.

Pour construire son modèle, le module d'administration du contrôleur s'appuie sur le module d'administration de chaque tâche que nous avons présenté dans la première partie (figure 45, page112). La récupération des données peut se faire soit par consultation régulière, ce qui ne garantit pas l'exactitude du modèle, soit par notification. Cette dernière nécessite la mise en place d'une communication par publication/souscription. Elle peut se baser sur les mécanismes de communication que nous présenterons après, ou s'effectuer par appel direct via une interface spécifique. L'utilisation de notifications apporte un plus indéniable mais induit des coûts en communication pour une précision qui n'est pas forcément nécessaire.

Pour y remédier, il est possible de ne notifier que les modifications importantes, c'est ce que nous faisons en sélectionnant ces données.En effet, la représentation de l'ensemble des informations disponibles sur les tâches n'est pas réaliste car certaines évoluent trop rapidement et le maintien du modèle serait trop couteux. Parmi les informations, il faut donc faire un tri pour ne retenir que celles restant stables. Les autres données pourront être accédées par une consultation directe.

Le modèle contient les informations suivantes, qui sont relativement stables dans le temps et restent valables sur plusieurs minutes :

- le **nom, l'identifiant unique, le type et l'état de la tâche**. L'état de la tâche est donné sans les méta-informations qui, elles, peuvent changer à chaque cycle.
- la **valeur des propriétés de configuration** d'une tâche ;
- la **configuration du scheduler** ;
- la **configuration du coordinateur** ;
- la **configuration des ports d'entrée et de sortie**.

Les statistiques et la valeur des données stockées dans la base ne devraient pas être remontées car elles sont susceptibles de varier énormément.

Enfin comme nous l'avons mentionné dans la première partie, les tâches peuvent envoyer des messages d'administration à l'intention du module d'administration du contrôleur. Ces messages peuvent être associés au modèle quand leur nombre n'est pas important, ou stockés dans la base de données. La couche d'administration et d'adaptation du gestionnaire se servira de ces informations pour modifier la configuration des tâches.

2.3 MECANISME DE COMMUNICATION

Après l'intervention du gestionnaire de configuration, le gestionnaire autonomique est composé d'un ensemble de tâches qui communiquent de façon standard via un mécanisme commun de communication.

Nous avons vu dans la première partie que les tâches disposent de modules spécialisés pour communiquer : les ports d'entrée et sortie. L'implémentation de ces ports est fortement dépendante du mécanisme de communication proposé par l'architecture de contrôle. Il existe principalement deux moyens de faire communiquer les tâches entre elles :

- **indirectement via un bus à** message qui se charge du routage ;
- **directement via un mécanisme de découverte** qui met en relation les producteurs et consommateurs.

Notre framework permet d'utiliser l'une ou l'autre des méthodes. Dans notre implémentation nous avons expérimenté les deux. Chacune présente des avantages et des inconvénients que nous exposons ici.

La méthode la plus traditionnelle pour construire un système à base d'événements est d'utiliser un MOM (Message-Oriented Middleware) spécialisé dans leur distribution. La figure 52 illustre un échange de messages via un tel mécanisme. Chaque producteur de données publie les données sur un sujet du bus. Les tâches intéressées par un type de donnée particulier peuvent s'abonner à ce sujet. Pour chaque sujet le bus connait la liste des tâches abonnées et il se charge de la distribution et du routage des messages vers les tâches concernées.

L'utilisation d'un bus a des avantages.

Certaines implémentations de bus offrent des garanties sur l'ordre des messages. Par exemple, ils peuvent garantir que les messages seront fournis dans l'ordre de création à la tâche ; mais il est également possible de privilégier l'échange de messages plus prioritaires. Cette

caractéristique est mise à profit pour améliorer la réactivité du gestionnaire lors de situations critiques.

Le passage par une entité centrale augmente le contrôle de l'expert sur les messages échangés. Ainsi, le bus peut être utilisé pour faire du filtrage sur certains types de messages qu'il ne souhaite pas voir traiter. Il est alors possible de ne pas distribuer les messages à certaines catégories de tâches pour éviter les conflits. En contrôlant la distribution, la résolution des conflits peut se faire en amont et, pour la plupart des cas simples, il est possible de se passer de coordinateur et même de scheduler.

L'observation du comportement du gestionnaire devient plus aisée car il est facile de calculer et d'ajuster les fréquences de production et réception de messages. Il est alors possible de prévenir des tempêtes d'événements qui peuvent survenir lorsqu'un producteur inonde les autres tâches par un surplus d'informations. De nombreuses implémentations de MOM offrent ainsi la possibilité de bannir les producteurs excessifs et les consommateurs trop lents.

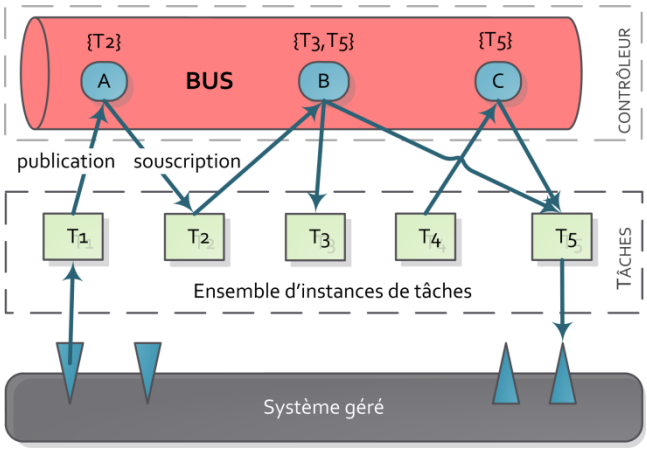

Figure 51 : Communication par bus

En outre, l'utilisation de nombreuses implémentations de bus existantes supporte la distribution et l'échange de messages sur plusieurs plateformes. Cependant cet avantage est contrebalancé par le fait que le reste de l'architecture (mécanismes de gestion de conflits, bases) doive supporter cette distribution. La distribution est une problématique que nous avons choisi d'ignorer dans ce travail.

Enfin, un avantage certain de cette méthode est qu'elle permet de s'appuyer sur des logiciels existants, sophistiqués et éprouvés. De nombreux middlewares fournissent des outils pour la communication entre applications ou services : par exemple OSGI propose l'Event Admin. On peut citer également les bus à messages tels que JORAM ou OPENJMS. L'implémentation de ports au-dessus de ces systèmes est très facile car il n'y a pas à gérer de tampon ou de synchronisation.

En revanche, l'utilisation d'un bus induit nécessairement des coûts en performance influençant négativement la réactivité du gestionnaire. Les implémentations de bus sont généralement assez centralisées, ce qui constitue un goulot d'étranglement ralentissant l'échange des messages. D'autre part, la défaillance du bus entraine nécessairement une perte d'informations et l'arrêt des échanges entre les tâches. Enfin, l'intégration des bus dans notre architecture ne se limite pas seulement à la réalisation des ports d'entrée et sortie. Les MOM sont des logiciels de grande taille possédant des configurations parfois complexes et qui nécessitent des compétences importantes pour être utilisés de façon optimale. Ils ont un impact significatif sur les plateformes de petite taille en consommant des ressources.

En comparaison, la communication directe offre de bien meilleures performances. Il s'agit schématiquement de distribuer le bus à travers les ports pour que chaque producteur gère la liste de ses consommateurs. Pour mettre cette communication en place, nous utilisons une approche à services. La figure 52 représente la même communication que la figure précédente, mais en échange direct. Ici, chaque tâche productrice de données publie un service dans le registre permettant au consommateur de s'abonner. Lorsqu'un consommateur apparaît, il cherche dans le registre des producteurs et s'abonne auprès d'eux. Alors, à chaque production de données, le producteur envoie les données aux tâches concernées.

Le principal avantage ici est la performance bien supérieure à celle d'un MOM. Il n'y a plus de goulot d'étranglement et la disparition du registre n'interrompt pas la communication entre les tâches qui se connaissent. Si elle survient, il suffit de créer un nouveau registre pour que les services se republient. Ceci n'engendre pas ou très peu de pertes de messages.

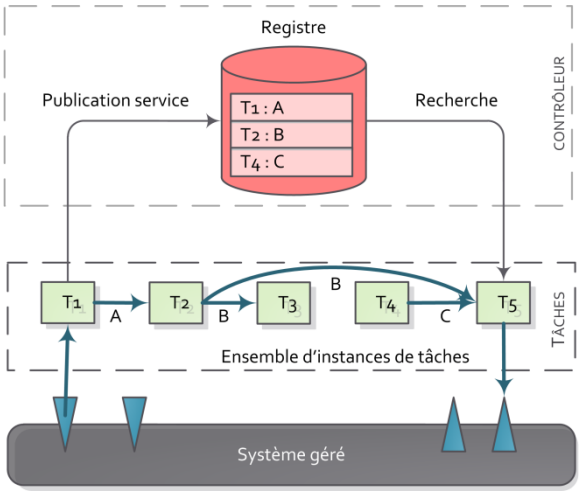

Figure 52 : Communication directe par découverte

Sur les plateformes à services de type OSGi, le registre est fourni et son utilisation a peu de conséquence sur les performances. L'appel entre tâches est un appel direct de méthode et offre donc des performances excellentes.

En revanche, l'implémentation des ports est plus délicate car il faut gérer des tampons et les threads pour éviter de bloquer la communication lors de l'envoi de messages. Les ports sont des unités fortement réutilisées, ce qui contrebalance cet inconvénient. Une fois stabilisée, leur implémentation offre un bon niveau de qualité. Notre implémentation des ports, complètement fonctionnelle, ne dépasse pas le millier de lignes. D'autre part les possibilités de filtrage et l'observation des événements échangés sont rendues plus difficiles.

Des deux méthodes nous préférons la dernière qui, si elle est plus difficile à mettre en œuvre, présente de meilleures performances, ce qui est important pour un gestionnaire. En revanche, dans un cadre davantage distribué avec des tâches sur plusieurs plateformes, l'utilisation d'un bus est plus raisonnable pour faire face à la complexité introduite par la gestion du réseau. Il n'est pas non plus exclu de se reposer sur plusieurs mécanismes à la fois, avec l'utilisation de plusieurs ports de sortie par exemple et un routage dépendant des données.

Remarquons qu'avec une implémentation adéquate, il est possible de changer dynamiquement au cours de l'exécution le mode de communication utilisé. Il faut pour cela être capable de changer les ports de communication de chacune des tâches, ce qui est possible grâce à l'architecture modulaire. Toutefois, la modification à l'exécution de la communication n'est souvent pas indispensable et les contraintes liées à sa mise en œuvre sont trop importantes. La

capacité à la modifier durant la conception et avant le déploiement est généralement suffisante. Nous verrons que notre implémentation ne permet la modification qu'avant le déploiement.

Ici nous voyons bien l'intérêt de séparer clairement les mécanismes de communication. Une même implémentation de processeur pourra être très facilement réutilisée avec différents mécanismes. Il existe d'autres implémentations possibles de ce mécanisme, en se basant par exemple sur les couches réseau avec des sockets et diffusion via multicast. Cette variété doit permettre l'utilisation des tâches dans des cadres différents et offre une certaine flexibilité lors du développement.

2.4 SELECTION ET RESOLUTION DE CONFLITS

Avec les deux mécanismes précédents, nous sommes capables de concevoir un gestionnaire complet composé d'un pool de tâches dont l'activité est surveillée par le gestionnaire de configuration. Elles échangent leurs données via un support de communication décrit plus haut et peuvent déjà constituer des boucles d'adaptation complètes. Dans l'essentiel des cas, ils sont suffisants, notamment lorsque le gestionnaire est conçu de toutes pièces et que l'expert maitrise parfaitement le cycle de développement des tâches. Souvent, il n'est pas nécessaire d'introduire de conflit ou de concurrence entre les tâches.

Cependant, comme nous l'avons expliqué lors de la première section, lorsque le nombre de tâches devient important et que les objectifs à atteindre pour le gestionnaire sont variés et dépendent du contexte, l'apparition de conflits est inévitable. Tous les systèmes basés sur des événements et ne possédant pas un flux d'informations statiquement définies finissent par être confrontés à ce problème. Les systèmes événementiels induisent nécessairement des conflits.

De plus, la mise en concurrence de plusieurs algorithmes spécialisés dans la résolution d'un même problème mais proposant des solutions différentes est intéressante. Les algorithmes de planification sont par exemple connus pour offrir des plans solutions différents pour une même situation. Selon les données, certains sont plus rapides, d'autres fournissent des solutions plus optimales, enfin d'autres ne terminent pas. La mise en concurrence de plusieurs tâches permet l'exploration de diverses solutions possibles.

Ainsi est-il possible de concevoir un gestionnaire constitué de tâches alternatives ou simplement conflictuelles parce qu'elles cherchent à remplir des objectifs qui s'opposent dans un contexte particulier. L'objectif du mécanisme de sélection est d'organiser la coopération entre les tâches, en anticipant ou en arbitrant les conflits.

Il est important que le mécanisme de sélection mis en œuvre soit aussi indépendant que possible. Il doit être capable de faire sa sélection entre les différentes tâches sans expertise – ou avec un minimum d'expertise –sur la fonctionnalité et sans se soucier de son implémentation. Sans une telle séparation, la modularité et l'indépendance des tâches en souffriront. C'est pourquoi les tâches doivent fournir les informations suffisantes pour que le mécanisme de sélection n'ait pas besoin d'informations supplémentaires.

Les informations sur lesquelles sont basées les décisions de l'algorithme de sélection sont variées. Lorsqu'il agit en amont de l'exécution, l'algorithme peut se baser sur :

- **une priorité sur la tâche** : la tâche la plus prioritaire s'exécute alors. Elle peut s'accompagner d'une priorité sur les données. Dans ce cas, les données les plus prioritaires sont d'abord attribuées, ce qui change le classement.
- **une quantité de jetons disponible** : à chaque requête exécutée, la tâche perd une certaine quantité de jetons. Lorsqu'elle ne peut plus payer, elle s'arrête. La tâche la plus riche s'exécute ; les tâches regagnent du capital au fil du temps.
- **le nombre d'exécutions de la tâche** : qui est une variante moins sophistiquée des jetons.

- **des événements extérieurs** avec entre autres le moment de la journée ou la mémoire disponible dans le système.
- **la nature des informations échangées :** le mécanisme peut modifier son choix en fonction des dernières informations échangées au sein de la tâche et de leur contenu.
- **les résultats antérieurs de la tâche** : le module de statistiques de la tâche évalue ses performances et notamment son temps d'exécution. En fonction du contexte, lorsqu'une réactivité accrue est demandée, l'algorithme peut par exemple privilégier les tâches les plus rapides.

Pour collecter les informations qui lui sont nécessaires, le mécanisme se base donc non seulement sur la requête fournie par la tâche, mais aussi sur les informations données par le module de statistiques, les données échangées directement, celles stockées dans la base de données et les messages d'administration. Le mécanisme peut également implémenter des routines de récupération d'information contextuelle dans l'environnement – on préféra cependant le déploiement de tâches prévues à cet effet. Les données produites par ces tâches seront utilisées par le mécanisme de sélection ; dans la solution d'arbitrage, par exemple, l'arbitre peut réutiliser les implémentations des ports que nous avons présentées pour les collecter.

Comme nous l'avons signalé dans la description de la tâche, le mécanisme de sélection peut à tout moment modifier l'état d'une tâche par l'ajout de méta-informations. Ces dernières permettent de garder en mémoire des données qui lui sont utiles pour guider son choix : nombre de jetons, durée d'inhibitions par exemple. A l'installation d'une tâche, il est important que l'expert ou, comme c'est préférable, le mécanisme de sélection puisse lui attribuer un état initial

Il existe de nombreuses variantes d'implémentation des mécanismes de sélection, ce qui justifie notre choix de séparer clairement les opérations de sélection, en rendant modulaire le coordinateur sur la tâche et en associant un module spécialisé au contrôleur. Les développeurs peuvent ainsi écrire leurs propres algorithmes et les réutiliser d'un gestionnaire à l'autre.

Il existe plusieurs façons de gérer les conflits entre les tâches :

- **par synthèse** en modifiant en aval la nature de la réponse apportée par chaque tâche. Une tâche de synthèse est alors utilisée ; elle reçoit la contribution de chaque tâche et choisit la réponse appropriée.
- **par filtrage** en ne distribuant les messages qu'aux tâches que l'on souhaite voir activées. Le mécanisme de sélection est alors fusionné avec le mécanisme de communication, le rôle du scheduler est réduit et il n'y a plus de coordinateur.
- **par arbitrage** en faisant intervenir un arbitre auquel l'exécution de chaque tâche est soumise.
- **par négociation** en créant les conditions d'une négociation entre les coordinateurs de chaque tâche. Ici la sélection est opérée par chaque coordinateur qui implémente un algorithme de sélection particulier. L'architecture de contrôle se cantonne à la mise en relation des tâches et/ou à la distribution des requêtes.

La première solution fonctionne en aval de l'exécution des tâches en concurrence, tandis que les trois autres proposent de résoudre les conflits en amont. Il est possible de combiner les deux approches avec une vérification a priori et a posteriori Chacun des ces mécanismes présente des avantages, et certains peuvent être utilisés en parallèle pour faciliter la mise en œuvre de la coopération entre les tâches. Etudions-les dans l'ordre.

Utilisation d'une tâche de synthèse

Il est possible de faire coopérer les tâches en effectuant la synthèse de leurs propositions en aval. Cette méthode ne fait pas vraiment partie du contrôleur. Nous l'avons signalé lors de la présentation du rôle des tâches.

Nous proposons de faire appel à une tâche de synthèse (figure 53) qui, sur la base de plusieurs propositions apportées par des tâches en concurrence et en s'appuyant sur l'historique d'exécution et les statistiques, fournit une solution synthétique et cohérente.

Ici l'objectif n'est pas d'éviter l'exécution des tâches en conflit mais au contraire de profiter de l'expertise de chacune d'entre elles. Si on soumet à deux planificateurs un même problème et que chacun propose un plan solution différent, la tâche de synthèse peut par exemple évaluer le temps de mise en œuvre et choisir le plan le plus rapide ou celui qui demande le moins de ressources, ou encore adopter le plan le plus précis. Parfois, il est en effet aisé de comparer deux solutions pour déterminer laquelle est la meilleure sans pour autant connaître à l'avance la méthode permettant de l'obtenir. C'est dans ces cas que l'exploration de plusieurs chemins est intéressante.

L'utilisation d'une fenêtre temporelle pour attendre la fin éventuelle de l'exécution de chaque tâche est souvent indispensable. Le scheduler de la tâche de synthèse doit être configuré de façon à attendre que plusieurs propositions arrivent, sans pour autant que cela influence trop négativement les performances du gestionnaire.

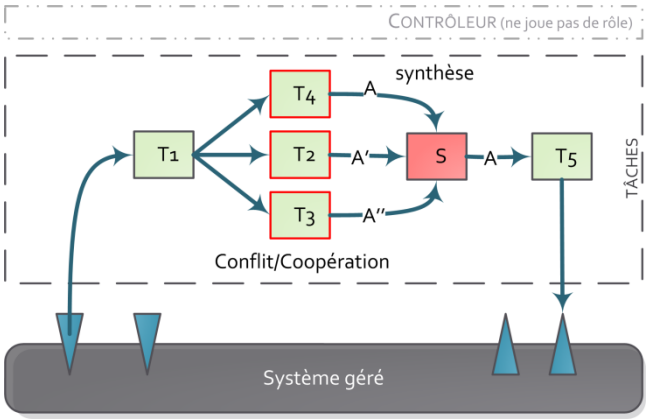

Figure 53 : Tâche de synthèse

Dans les cas où la réactivité prime sur le reste, la tâche peut simplement ne considérer que la première réponse et filtrer les suivantes. Ainsi il est possible d'évaluer plusieurs solutions et de choisir la plus rapide, ce qui permet l'optimisation des performances du gestionnaire.

Pour fonctionner plus efficacement il, est possible d'accorder des droits particuliers à la tâche de synthèse. Pour effectuer une évaluation plus précise, la tâche est capable d'identifier chacune des tâches qui soumettent une requête et de se servir de leur module de statistiques. Comme nous l'avons expliqué, le processeur n'a théoriquement pas accès aux identifiants des tâches productrices afin de maintenir la séparation des préoccupations. Ici, il faut contrevenir à ce principe en fournissant cette information, pour donner une plus grande latitude à la tâche de synthèse.

Cette pratique présente une limite : elle ne peut pas s'appliquer aux tâches agissant sur le système. Comme elle agit en aval, les exécutions des tâches ont déjà eu lieu ; si plusieurs tâches ont effectué des modifications sur le système, il est difficile pour la tâche de synthèse d'annuler les actions accomplies. Pour permettre la résolution de ces conflits particuliers, il faut agir en amont avec les méthodes que nous présentons dans la suite.

Filtrage avant distribution

Nous avons évoqué le mécanisme de filtrage (figure 54) lors de la présentation du mécanisme de communication. Sa mise en œuvre nécessite un bus.

Le principe est l'anticipation des conflits en évitant que les tâches ne reçoivent de données conflictuelles. A chaque réception d'un message par le mécanisme de communication, l'algorithme de sélection détermine les tâches autorisées à le recevoir. Par conséquent, les tâches qui souscrivent à des données n'ont pas la garantie de les recevoir.

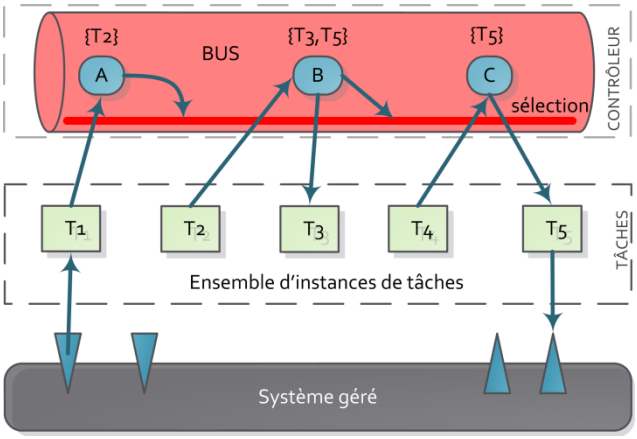

Figure 54 : Sélection par filtrage

Le principal avantage du filtrage est sa simplicité de mise en œuvre et sa performance. Les décisions sont prises avant de consulter les tâches et le mécanisme garantit qu'aucune tâche ne s'exécutera sans autorisation. Il n'est pas nécessaire de disposer d'un coordinateur, ni d'échange entre le mécanisme de sélection et la tâche.

En revanche l'utilisation d'un scheduler est rendue complexe : que faire lorsque la tâche a déjà reçu l'autorisation de traiter une donnée et que son scheduler estime qu'il ne faut pas la traiter ? Par exemple, imaginons deux tâches T1 et T2 qui s'intéressent au rapport de type (A ET B) pour T1 et A pour T2. Lors de la réception d'une donnée de type A, le mécanisme de sélection choisit de distribuer le rapport à la tâche T1 – parce qu'elle est plus prioritaire par exemple. T2 ne reçoit donc pas la donnée et ne s'exécutera pas. Pour autant, la tâche T1 ne peut pas encore traiter le rapport puisqu'il lui manque B et elle doit attendre cet événement. Si aucune donnée B n'arrive ou qu'elle arrive trop tard, aucune des tâches n'aura pu s'exécuter. Alors qu'avec une distribution aux deux et une demande d'autorisation, la tâche T2 aurait eu sa chance..

Le risque est donc que le gestionnaire ignore des données. A ce problème, deux solutions

On peut d'abord imaginer remonter ces contraintes au niveau du bus. Cependant, donner au bus la connaissance des facteurs de déclenchement d'une tâche n'est pas réaliste lorsque le nombre de tâches est important. Il n'est pas possible de remonter les exigences de chaque tâche car cela revient à déplacer les schedulers au niveau d'un mécanisme de sélection centralisé avec toutes les conséquences sur les performances et la maintenabilité.

L'autre solution est de faire attendre le bus avant de répartir les données. Quand le bus reçoit un message, il arme un timer dont la durée est précisée par l'expert. A chaque expiration du délai, le bus distribue les données. Avec cette fenêtre temporelle, il est possible de distribuer les rapports par groupe et d'éviter ainsi le problème décrit ci-dessus. Cependant, cette solution ne s'applique pas à certaines conditions de déclenchement. C'est notamment le cas de celles qui déterminent l'activité de la tâche en fonction de facteurs extérieurs aux données : heure, nombre de messages collectés ou autres. Enfin, un manque de réactivité est inhérent à cette méthode ; elle est toutefois obligatoire et reste présente dans toutes les solutions que nous présenterons.

En résumé, pour les gestionnaires simples, disposant de peu de tâches, la solution du filtrage est efficace et peu couteuse à mettre en œuvre. Nous avons utilisé cette technique avec succès dans les premières versions de notre architecture. Elle a cependant l'inconvénient de

reposer sur un bus centralisé qui peut connaître des défaillances et qui constitue un goulot d'étranglement. L'efficacité diminue ainsi lorsque le nombre de tâches est important. D'autre part, cette méthode crée des cas de famine dans lesquels certaines tâches ne reçoivent pas de messages et n'assurent pas que les données seront traitées. Une mauvaise configuration de ce système peut alors conduire à des gestionnaires inopérants. Pour éviter une telle situation, il faut faire appel à une solution d'arbitrage ou de négociation telle que décrite ci-après.

Sélection par arbitrage

La mise en œuvre d'un arbitre est beaucoup plus souple mais également plus complexe à implémenter. Elle repose sur la communication entre les coordinateurs de chaque tâche et un arbitre appartenant au contrôleur. Avec cette méthode, le coordinateur de chaque tâche soumet une requête qui exprime les besoins de la tâche pour s'activer. L'arbitre prend sa décision en trois étapes (figure 55) :

1. **un collecteur reçoit les requêtes**. Il arme un timer à la première requête et attend l'arrivée de nouvelles requêtes. Lorsque le délai arrive à expiration et qu'il n'y a pas de sélection en cours, il envoie l'ensemble des requêtes à l'algorithme de sélection pour l'élection. Le collecteur peut recevoir des requêtes alors qu'une sélection est en cours, il attend alors la fin de l'exécution de l'algorithme. Il est possible dans ce cas d'ajouter un délai d'expiration aux requêtes, cependant la vitesse de l'algorithme rend ce mécanisme superflu. Le collecteur accompagne chaque requête de l'état des tâches en conflit lorsqu'il est configuré pour le faire.
2. **un algorithme de sélection procède à l'élection des requêtes**. Sa configuration permet de changer son fonctionnement. Il a la possibilité de baser son raisonnement sur l'état de la tâche, le contexte ou les données. Il faut alors qu'il implémente les mécanismes de collecte du contexte ou qu'il s'appuie sur d'autres tâches. A l'issue de son exécution, l'algorithme fournit un ensemble de requêtes élues et la liste des nouveaux états (éventuellement de tâches qui n'ont pas fait de requêtes).
3. **un mécanisme de distribution notifie les tâches concernées** qui peuvent alors commencer leur exécution ou soumettre de nouvelles requêtes. Il se charge également de modifier les états des tâches.

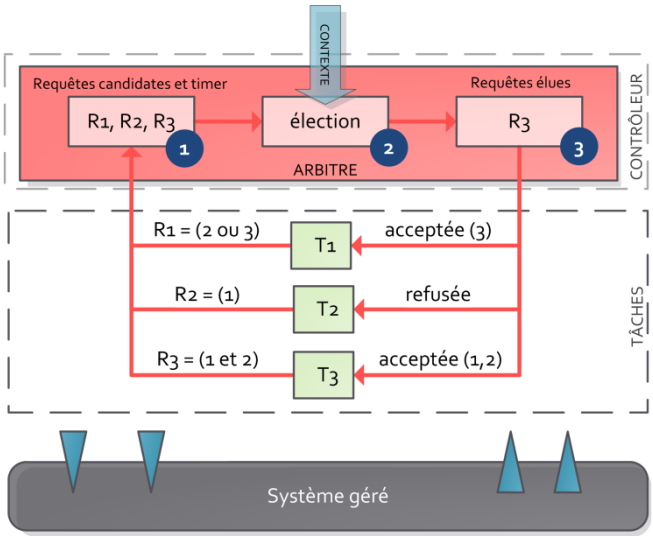

Figure 55 : Sélection par arbitrage

Le découpage de l'arbitre est important pour éviter l'activité redondante du codage de la communication entre les coordinateurs et l'arbitre. Il permet aussi de séparer la gestion des threads : une requête par tâche au maximum.

La communication peut s'appuyer sur le mécanisme de communication que nous avons décrit. Cependant, comme ce mécanisme est asynchrone, le coordinateur de chaque tâche doit alors veiller à ce qu'une tâche ne soumette pas plusieurs requêtes à la fois ; cette mise en œuvre peut s'avérer complexe. Une solution alternative est de procéder par appel direct avec un blocage au niveau du collecteur de telle sorte que les coordinateurs des tâches sont bloqués.

Nous fournissons une implémentation de collecteurs et distributeurs, ce qui permet au développeur de se focaliser sur l'implémentation de l'algorithme et non sur la synchronisation. Nous souhaitons autant que possible favoriser la réutilisation du code et, dans l'idéal, que la mise en œuvre d'un nouvel algorithme ne demande pas la modification des coordinateurs des tâches.

La nature de la requête peut varier en fonction de l'algorithme choisi et des informations nécessaires. Etudions trois variantes.

Cette requête peut être une simple demande d'exécution contenant uniquement le nom de la tâche. Dans ce cas l'arbitre ne pourra prendre de décision qu'à partir des informations dont il dispose possède sur la tâche (la priorité, le nombre de jetons), n'en ayant aucune sur les données. Bien que très sommaire, cette forme de requête permet la prise en compte d'un grand nombre de situations. En effet dans bien des cas, connaitre la nature des données que chaque tâche souhaite traiter n'est pas essentiel ; c'est notamment le cas si les tâches conflictuelles ne traitent qu'un seul type de donnée.

Une version plus complexe de requête est constituée de la liste des données que la tâche souhaite traiter. A partir de cette information, l'arbitre peut baser son choix en accordant la priorité aux requêtes les plus importantes et urgentes. Cela évite que des tâches avec des priorités élevées inhibent l'exécution des autres tâches en ne traitant que des requêtes peu importantes. Cette information devient utile lorsque l'activité des tâches est variée et que ces dernières traitent des données de natures différentes. Pour éviter d'engendrer un trafic inutile, il est préférable de n'envoyer que l'entête des données qui contient le type et l'identifiant unique. Un arbitre qui veut faire un choix en fonction du contenu des données pourra s'y abonner comme le fait n'importe quelle tâche en passant par le mécanisme de communication.

Enfin, la tâche peut donner une version plus complète d'une requête, sous la forme d'une expression composée de coordinateurs (ET et OU) telle que nous l'avons décrite dans la présentation du coordinateur. La mise en œuvre de cette méthode requiert que le scheduler fournisse une requête exprimée comme nous l'avons détaillé dans la description des tâches. Cette information donne beaucoup de latitude à l'arbitre, qui peut alors ne satisfaire que partiellement une requête en répondant par une liste de données autorisées aux tâches concernées. Ce mécanisme n'est qu'une extension du précédent mais offre beaucoup de souplesse. C'est celui que nous utilisons actuellement et nous fournissons une implémentation de scheduler, coordinateur et arbitre réutilisable. Il est possible de fonctionner en mode dégradé, en fournissant systématiquement des expressions constituées de « ET » ce qui revient, comme pour le cas précédent, à ne fournir qu'une simple liste.

Prenons un exemple. La figure 55 illustre l'utilisation d'un arbitre. Sur cette figure, les trois tâches T1, T2 et T3 expriment des besoins différents. Toutes ces tâches veulent traiter la donnée 1. La tâche T1 exprime une alternative en proposant (1 ou 2 ou 3) tandis que la tâche T3 a besoin absolument de 1 et 2 ensemble. La tâche T3 est ici la plus prioritaire puisqu'elle obtient de traiter sa requête en entier. Si la tâche T2 qui était en concurrence sur la donnée 2 ne peut pas s'exécuter, la tâche T1 peut traiter la donnée 3 qui n'était pas réclamée par T3. En revanche, T1 ne pourra traiter ni 1 ni 2 qui sont consommées par T3.

Ainsi, il y a plusieurs façons de répondre à une requête et il se dégage, dans ce cas, plusieurs possibilités pour un arbitre. Citons en deux.

Il faut d'abord distinguer les tâches qui ne doivent pas être exécutées ensemble de celles qui ne doivent pas traiter les mêmes données. Nous appelons ces deux cas exclusion et exclusivité :

- deux tâches qui fonctionnent en exclusivité peuvent s'exécuter en même temps, tant qu'elles ne traitent pas les mêmes données : c'est le cas des tâches de notre exemple ci-dessus.
- deux tâches qui s'excluent ne doivent pas fonctionner ensemble – dans notre exemple seule T3 peut alors s'exécuter.

L'algorithme que nous proposons dans notre implémentation utilise un tableau dans lequel l'exclusion ou l'exclusivité de chaque tâche sont exprimées. Il est possible de formuler cette information en type de tâche.

D'autre part, il existe plusieurs politiques possibles. Doit-on privilégier le nombre de tâches s'exécutant ou la quantité de données traitées ? Exécuter plusieurs tâches à la fois permet d'explorer plus de solutions, mais par le jeu des exclusions il est possible que certaines portions de requête ne soient pas traitées et en conséquence moins de données le sont. A contrario, s'assurer que le plus de données possibles sont traitées permet de prendre en compte un plus grand nombre de stimuli. Il n'y a pas de bonne solution a priori. Souvent un algorithme si complexe n'est pas nécessaire. Nous avons expérimenté plusieurs algorithmes, l'un qui privilégie les données les plus prioritaires, l'autre qui ne se base que sur la priorité des tâches.

Un aspect particulier est à prendre en compte pour la conception de l'algorithme. Il faut considérer le temps de réaction des tâches lorsque l'on souhaite gérer les conflits. Souvent les solutions mises en œuvre par les tâches ont un temps d'action important. Dans l'intervalle, il ne faut pas qu'une tâche concurrente vienne détruire le travail accompli. Une solution pour remédier à ce problème est l'inhibition des tâches concurrentes de la tâche qui s'exécute. Il est même possible d'inhiber la tâche elle-même pour éviter qu'elle ne s'exécute trop souvent. Par exemple, lors de l'expression d'un conflit, il est possible d'indiquer pour chaque cas un temps pendant lequel les requêtes seront systématiquement refusées. Ce délai d'inhibition est conservé au niveau de l'état de chaque tâche.

En résumé, par rapport au filtrage, le mécanisme d'arbitrage offre beaucoup de flexibilité. En revanche, il entraine des communications supplémentaires entre l'arbitre et les coordinateurs. Il est possible cependant de modérer ce coût en configurant les coordinateurs pour qu'ils n'expriment de requêtes que lorsque nécessaire. Il est possible par exemple de se baser sur la nature des données. Ainsi le nombre de requêtes peut être fortement contenu.

D'autre part, comme pour le filtrage, on peut lui reprocher son caractère centralisé qui peut soulever des problèmes de passage à l'échelle. C'est une des raisons pour laquelle nous avons introduit la notion de composite que nous présentons plus loin.

Négociation

Il existe une variante décentralisée du mécanisme de l'arbitrage : la négociation. Avec cette méthode, chaque coordinateur de tâche négocie avec les autres tâches. Pour y parvenir, il faut que les tâches conflictuelles soient regroupées pour leur permettre de se découvrir. L'expert attribue à chaque tâche un identifiant de groupe ou les connecte à un canal commun.

Les tâches n'ont pas besoin de se connaitre ni d'avoir d'expertise sur leur fonctionnement, mais simplement de connaître le groupe auquel elles appartiennent. Une tâche peut éventuellement appartenir à plusieurs groupes différents. Une façon de grouper les tâches est d'associer celles qui traitent une donnée particulière. Dans ce cas, le coordinateur sait à quel

groupe envoyer une requête. Nous ne gérons pas l'utilisation de plusieurs groupes dans notre implémentation.

Les coordinateurs communiquent au moyen d'un bus comme le montre la figure 56, ou via un mécanisme de découverte. La communication peut là encore s'appuyer sur le même mécanisme de communication que décrit précédemment ou bien être indépendante. La diffusion est la méthode la plus simple à mettre en œuvre ; le coordinateur envoie sa requête à chaque tâche du groupe y compris celles qui ne sont pas intéressées. Cela évite de gérer un système d'abonnements.

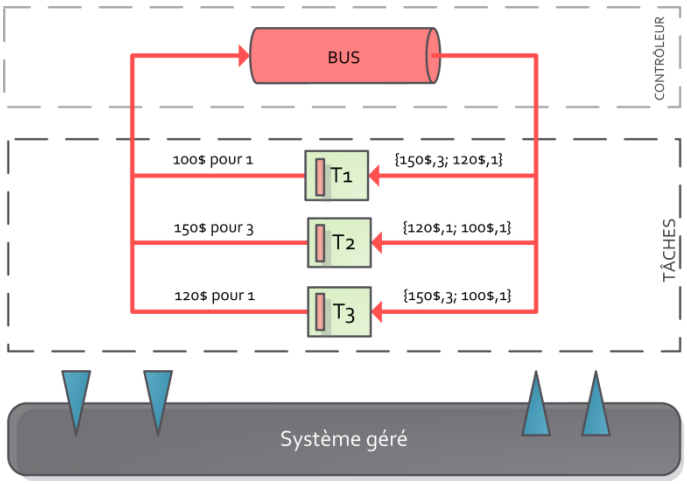

Figure 56 : Sélection par négociation

Alors que dans la solution proposée par l'arbitrage les coordinateurs avaient un rôle passif vis-à-vis de la sélection, dans cette méthode ils sont au cœur du processus de décision. Chaque coordinateur implémente l'algorithme de sélection qui produit l'information partagée avec les autres tâches.

Cette information de sélection est très différente selon la nature de l'algorithme. On peut ainsi envisager l'échange :

- **d'un message d'inhibition** : lorsqu'une tâche désire s'exécuter elle calcule et envoie une inhibition aux autres tâches. Si elle ne reçoit pas une inhibition plus forte dans la période de temps, elle décide alors de s'exécuter.
- **d'une somme d'argent investie** : chaque tâche dispose d'un capital initial fixé par l'expert. A intervalles de temps réguliers, les tâches regagnent du capital. Lorsqu'une tâche souhaite s'exécuter, elle propose une somme d'argent associée à la donnée souhaitée. Si aucune contre-offre n'est reçue dans un intervalle fixé, elle s'exécute et paie la somme promise. Dans le cas de la figure 56, les tâches T2 et T3 s'exécutent et perdent respectivement 150$ et 120$.
- **d'une estimation du temps d'exécution** : lorsqu'une tâche veut s'exécuter, elle fournit aux autres tâches une estimation du temps. La tâche qui se prétend la plus rapide l'emporte sur les autres.
- **d'une estimation des ressources consommées** : le principe est ici le même que ci-dessus.

Cette méthode implique que chaque coordinateur échange dans un langage commun, mais l'implémentation des coordinateurs proprement dite peut varier. Chaque tâche décide ainsi des critères qui conditionnent son exécution. Cette stratégie suppose que le mécanisme de communication est suffisamment robuste pour que chaque tâche reçoive les requêtes. En effet, si dans le cas de l'arbitre une perte de requête engendre la non exécution de la tâche, avec cette

méthode il peut se produire le contraire. Une tâche s'exécute tant que des requêtes concurrentes ne viennent pas remettre en question son action.

Avec cette méthode également, il est nécessaire d'avoir une fenêtre temporelle pour attendre l'arrivée éventuelle de requêtes concurrentes. C'est le coordinateur qui se charge de l'observation du canal et de l'évaluation du temps d'attente qui varie en fonction du contexte. De même, l'utilisation d'une inhibition est nécessaire, quelle que soit la nature de l'information échangée, pour garantir qu'une tâche n'essaiera pas de trouver une solution à un problème déjà en voie de résolution.

L'avantage de la négociation est qu'il n'y a pas de mécanisme centralisé. Chaque coordinateur reçoit une configuration initiale qui permet d'insérer la tâche dans le groupe. L'évaluation des besoins de chaque tâche peut s'effectuer en utilisant différentes implémentations de coordinateur, ce qui apporte de la souplesse.

En revanche, certaines évaluations sont complexes et doivent être cohérentes. Il est nécessaire entre autre que les tâches partagent la même échelle de valeur. Par exemple, la somme proposée par une tâche ne doit pas être systématiquement plus élevée que celle des autres tâches. A situation égale, les tâches doivent proposer la même information.

Comme il n'existe pas de configuration centralisée, il est difficile d'appréhender le comportement des tâches et de détecter les problèmes. Ce mécanisme est plus délicat à mettre en œuvre et l'insertion d'une nouvelle tâche peut demander la modification potentielle de plusieurs tâches. La complexité de l'implémentation se déplace du contrôleur vers les tâches mais le coordinateur est a priori réutilisable.

En définitive, la négociation constitue donc une alternative intéressante à l'arbitrage et convient à des gestionnaires de complexité modérée. Sur des gestionnaires simples, nous avons pu reproduire les résultats obtenus avec un arbitrage centralisé.

Synthèse sur les mécanismes de sélection

Le découpage proposé par le framework permet la mise en œuvre de différentes stratégies de résolution de conflit. Il n'y a pas de solution prévalant sur une autre ; leur utilisation dépend fortement des conflits à gérer.

De toutes les stratégies, l'utilisation d'une tâche de synthèse est la plus intéressante pour l'exploration de plusieurs solutions possibles. La meilleure solution sera choisie en fin de boucle. Cette stratégie est compatible avec les solutions de résolution en amont, qui peuvent alors laisser s'exécuter plusieurs tâches.

Le filtrage et l'arbitrage sont des solutions centralisées qui ont potentiellement un impact sur les performances. Le filtrage convient bien aux gestionnaires simples et permet de réaliser des tâches sans scheduler, ni coordinateur. Il nuit en revanche à l'expression de conditions d'activation complexes et peut créer des situations de famine. L'arbitrage est une solution en apparence plus complexe à mettre en œuvre mais dont l'essentiel du code est réutilisable : coordinateur, collecteur et distributeur pour l'architecture que nous donnons. Il permet de couvrir une large palette de cas possibles. L'avantage des solutions centralisées réside dans la localisation de la configuration dans un seul module ; il est ainsi plus facile de comprendre le comportement de la sélection. En outre, il est envisageable de modifier dynamiquement l'algorithme de sélection car les coordinateurs des tâches n'ont pas besoin d'êtres modifiés. Il est également facile de l'activer et de le désactiver en fonction du contexte.

La négociation est une variante de l'arbitrage qui évite le recours à un module externe. Elle est plus difficile à mettre en œuvre car il faut s'assurer de donner à chaque tâche une configuration cohérente. La configuration se répartit sur un ensemble de modules. Il est également plus difficile de changer les mécanismes de sélection durant l'exécution. En revanche,

cette configuration décentralisée devient un avantage pour l'intégration de nouvelles tâches si le mécanisme est simple. Potentiellement, l'ajout d'une nouvelle tâche ne demande pas de modification sur d'autres parties du système.

Nous avons pu expérimenter chacune de ces techniques. Dans notre implémentation stabilisée, nous proposons une implémentation d'un coordinateur passif avec un arbitre qui se base sur les priorités et un autre qui se base sur des jetons. L'arbitrage nous semble la méthode la plus simple à réutiliser et à configurer car l'implémentation des coordinateurs ne change pas.

2.5 BASE DE DONNEES

Le dernier module appartenant au contrôleur est une base de données permettant le stockage des données partagées entre chaque tâche. Elle est utilisée par les processeurs des tâches, les schedulers et le mécanisme de sélection. Les processeurs s'appuient dessus pour échanger des données propres à la gestion. Les conditions de déclenchement des schedulers l'observent en attente d'un événement. Enfin, le mécanisme de sélection l'utilise pour récupérer des informations contextuelles.

Nous avons choisi de rendre cette base optionnelle. Dans les premières versions de notre approche nous n'avions pas éprouvé la nécessité de son utilisation pour plusieurs raisons. D'abord, chaque tâche possède un scheduler qui permet de conserver les données échangées. Ensuite le processeur n'est pas sans état et chaque processeur est libre de stocker des données d'une exécution à l'autre.

L'échange direct de messages suffit pour la conception d'un gestionnaire simple, dont l'activité est relativement linéaire et qui raisonne sur des données ponctuelles et éphémères. C'est particulièrement le cas des gestionnaires qui n'ont pas besoin de raisonner sur les exécutions précédentes, ou de ceux dont les tâches n'ont pas besoin de partager cet historique de traitement. En effet, comme les processeurs ont un état, ils peuvent stocker leur historique de traitement ; si le flux d'exécution passe systématiquement par eux d'une boucle à l'autre, l'historique est à jour.

Ici encore, le problème vient du fait que plusieurs tâches peuvent vouloir traiter des informations en parallèle en cas de coopération ou de mise en concurrence. Si les tâches sont utilisées par alternance, il devient difficile pour elles de conserver un historique à jour.

Une première solution est que chaque tâche produise un événement particulier à chaque action qui vienne mettre à jour les informations stockées par les autres tâches. Cependant, cette méthode induit des communications supplémentaires et il faut que chaque tâche traite ce nouveau type de message. Ensuite, l'ajout d'une nouvelle tâche pose un problème : elle n'a pas d'information sur l'historique. Lors de l'ajout d'une nouvelle tâche au système, il faut qu'elle soit capable d'adopter un comportement initial cohérent par rapport aux autres tâches. Dans un système dynamique comme le nôtre ce n'est pas possible.

Il est également possible que chaque tâche fournisse un service permettant aux autres tâches d'accéder aux informations qu'elles désirent. Dès lors, une nouvelle tâche peut consulter les tâches actives pour s'intégrer naturellement au gestionnaire. Toutefois cette solution est complexe à mettre en œuvre car elle implique que les tâches se connaissent et qu'elles soient capable de déterminer quelle tâche a accompli la dernière action. Comme pour les ports de communication, on peut faire appel à un registre. Bien que séduisante, cette solution pose un problème : la désinstallation d'une tâche risque d'engendrer une perte d'informations.

La solution la plus simple pour un partage d'historique et de données est donc l'utilisation d'une base de données (figure 57). Chaque tâche dispose d'une référence vers cette base et le processeur peut y stocker et consulter des informations. Pour rendre davantage transparente l'utilisation de la base de données pour le processeur, il est même possible d'utiliser des aspects qui se chargent de la collecte et la récupération des données comme nous le verrons pour notre implémentation.

L'un des avantages de l'utilisation d'une base de données est que sa réalisation peut se reposer sur des logiciels existants, de grande qualité. Leur réutilisation est importante car l'implémentation d'une base de données efficace est rendue difficile notamment par la gestion des accès concurrentiels. D'autre part, une base de données peut donner des garanties sur le stockage fiable des informations en mettant en œuvre des procédures de duplication et de sauvegarde des données stockées. Cette redondance rend ainsi le gestionnaire plus robuste.

En revanche, la base de données pose des problèmes en cas de distribution de tâches. Cette dernière induit fatalement une latence dans la consultation des données et un manque de réactivité du gestionnaire. D'autre part, les informations stockées occupent rapidement de l'espace de stockage. Toutefois, il est possible d'implémenter la base de façon hiérarchique, de sorte que les données les plus fréquemment échangées entre les tâches se trouvent stockées localement. Il est très probable en effet que les tâches qui partagent des informations soient proches : un groupe de moniteurs ou un groupe d'analyseurs. C'est sur ce principe, nous le verrons plus loin, que nous proposons de répartir la base pour les composites. Ces derniers permettent alors d'organiser les groupes de tâches.

Il est possible d'ajouter un raffinement à la base en lui permettant de notifier la modification d'une donnée aux tâches abonnées (figure 57). On rejoint alors les techniques évoquées pour le mécanisme de communication. Chaque tâche s'abonne aux propriétés qui l'intéressent et est notifiée des modifications. Cette capacité est intéressante pour les schedulers qui peuvent alors s'abonner à des événements en provenance de la base. Elle permet l'implémentation de conditions de déclenchements plus réactifs.

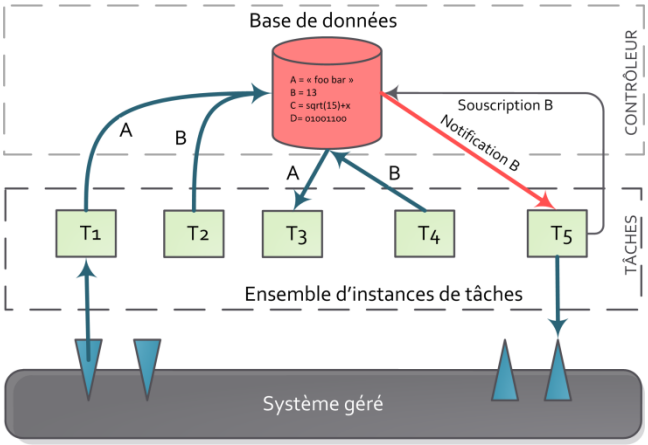

Figure 57 : Base de données

Dans notre architecture la base occupe un rôle passif vis-à-vis des données qu'elle gère. La raison est que nous considérons que cette base est un complément au mécanisme de communication principal. Pour des questions de performances, il ne nous parait en effet pas opportun d'utiliser cette base pour l'ensemble des échanges. Dans certains systèmes en revanche, les systèmes blackboard en particulier, elle joue un rôle central. Elle a un rôle plus important au niveau des données et participe notamment à leur classification et à leur intégration. Cette base est ainsi capable d'établir des relations entre des données de nature différente pour les mettre en évidence. Il est envisageable de proposer de tels mécanismes dans notre architecture, mais leur étude dépasse le cadre de ces travaux.

3 ADMINISTRATION DU GESTIONNAIRE : CONSTRUCTION ET ADAPTATION

Nous avons décrit l'ensemble des mécanismes qui permettent la création et l'agencement des instances de tâches. Une fois lancé, le gestionnaire fonctionne indépendamment et tant que les conditions d'utilisation du gestionnaire n'évoluent pas, les deux couches que nous avons présentées suffisent. Dans cette partie nous nous focalisons sur la construction du gestionnaire et sur les outils qui permettent son adaptation. Nous donnons également des pistes pour l'adaptation automatique.

Contrairement aux deux couches inférieures, l'administration du gestionnaire peut se composer de différents modules indépendants focalisés sur la modification du gestionnaire. Ils s'appuient sur les interfaces fournies par le module d'administration du gestionnaire. Voyons d'abord la construction puis étudions les outils d'adaptation.

3.1 CONSTRUCTION

Pour construire le gestionnaire, la couche d'adaptation se base sur les interfaces fournies par le module d'administration de la couche de contrôle. Ce dernier offre en soi une méthode programmatique de construction des gestionnaires, mais il nous apparaît cependant nécessaire de fournir un ADL permettant de construire le gestionnaire. Cet ADL est interprété par un module de construction ; il est alors plus facile de développer des outils – interface homme machine ou scripts – au dessus de ce module pour gérer la modification du gestionnaire par grandes unités logiques plutôt que par tâches.

Le module de construction est à l'origine de la création des couches inférieures et de leur configuration. La construction est guidée par une **description de gestionnaire souhaité** qui est une représentation abstraite de la configuration du gestionnaire désiré par l'administrateur ou un module d'auto-adaptation. Dans notre approche, toute modification du gestionnaire passe obligatoirement par la modification de ce modèle.

Le premier rôle du module de construction est d'analyser la description et de construire la version initiale du gestionnaire. Par la suite, toute modification sur le modèle abstrait est répercutée sur le modèle concret lorsque cela est possible.

La description du gestionnaire souhaité (figure 58) définit l'ensemble des tâches. La description d'une tâche contient la configuration des schedulers, coordinateurs et ports à utiliser. Chaque tâche fait référence soit à un type.

Sur la figure, l'implémentation du scheduler apparaît comme un attribut. La description doit y faire référence sans quoi ce sont le scheduler et le coordinateur par défaut – ils ne font rien – qui seront choisis. Pour ces composants aussi, l'idée d'un typage pourrait être introduite, mais pour ne pas complexifier outre mesure le framework, nous ne le faisons pas.

Comme nous l'avons expliqué pour la couche de contrôle, leur configuration varie en fonction de l'implémentation. C'est pourquoi ici, nous avons ajouté l'attribut *configuration* de type *XML*, car c'est le langage qui offre le plus de flexibilité. C'est à l'implémentation que revient la responsabilité de comprendre et de parser cette configuration. Nous n'avons pas représenté la configuration des ports qui, pour les tâches atomiques, est fournie en grande partie par le type de tâche (mise à part l'implémentation des ports choisie)

La description peut être donnée sous forme d'un arbre lorsque des tâches composites sont utilisées. Chaque nœud est une tâche rattachée à un contrôleur. Il existe un contrôleur principal par gestionnaire autonomique auquel toutes les tâches sont subordonnées.

La description du contrôleur contient donc la description de chaque élément qui le compose. Ici nous ne représentons que le mécanisme de sélection (arbitre) qui est le plus

important. Les autres mécanismes (communication, base de données et administration) sont moins susceptibles d'êtres modifiés et une version basique du framework ne fournit pas de moyen de changer leur implémentation. Ici encore, il serait possible de typer le mécanisme de sélection pour ne pas faire référence à une implémentation, et la configuration est fournie dans un langage que l'implémentation doit comprendre.

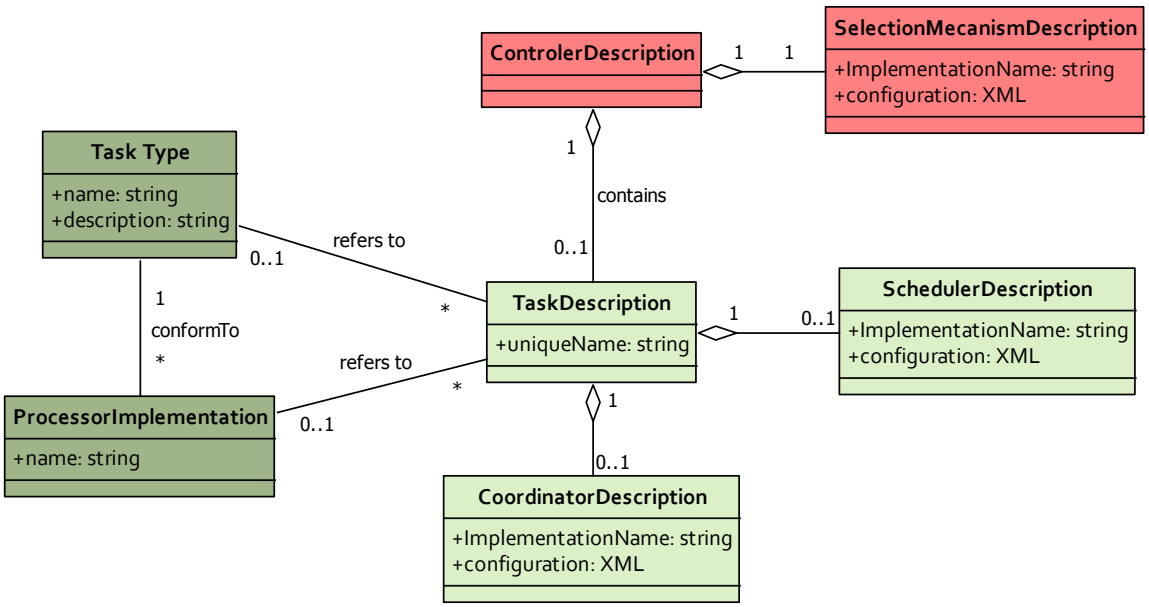

Figure 58 : Les composants essentiels de la description

Le module de construction a une autonomie limitée. Il prend peu de décisions, c'est le rôle soit de l'administrateur soit du module d'auto-adaptation. Etudions ses responsabilités.

La première est de trouver les implémentations des processeurs de tâches. En effet, si le type de tâche est toujours donné ou déduit, donner l'implémentation du processeur est optionnel. Le constructeur a alors la charge de découvrir une implémentation de processeur conforme au type de tâche. Ainsi, grâce à cette délégation, le développeur peut se focaliser sur la fonctionnalité et non l'implémentation.

Lorsque le type de tâche est indiqué, le module doit découvrir une implémentation de processeur de ce type précis dans le dépôt d'implémentations (figure 59). Par défaut, la première trouvée sera utilisée, mais il est possible de guider ce choix en fournissant une référence vers un service de comparaison indépendant (figure 59). Le rôle de ce service est de fournir un classement des différentes implémentations sur la base des méta-informations qu'elles donnent. Ces informations sont susceptibles de concerner entre autres la qualité de service garantie ou une estimation de la quantité de ressources nécessaires. A intervalles de temps réguliers, ou par notification, le module d'administration consulte le dépôt.

Si aucune implémentation correspondante n'est trouvée, le module de construction informe l'administrateur ou le module d'auto-adaptation. Il peut ainsi, sur ordre de la couche supérieure, reporter la création de la tâche, qui surviendra alors à l'apparition d'une implémentation appropriée dans le dépôt.

L'autre responsabilité du module de construction est d'assurer la cohérence avec le modèle de type de tâches. Les modèles de type de tâches et de type de données sont, comme le

dépôt des implémentations, stockés indépendamment des gestionnaires, de sorte qu'ils peuvent être utilisés pour la construction de plusieurs gestionnaires.

A chaque destruction d'un type de tâche, les tâches associées doivent à leur tour être détruites. La destruction d'un type de donnée entraine la suppression des types de tâches associés et par conséquent des tâches de ce type.

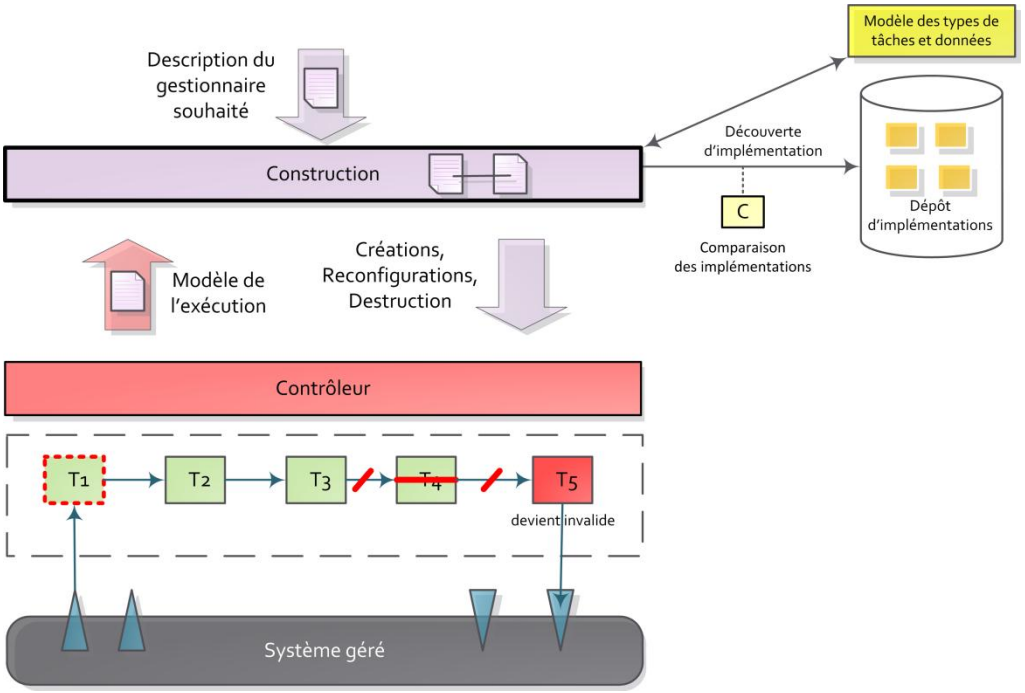

Figure 59 : Construction du gestionnaire

La gestion de la synchronisation entre type de tâches et dépôt d'implémentations dépend de l'implémentation du framework. Il est par exemple possible de supprimer les implémentations qui ne sont pas conformes à un type ou de maintenir un lien inverse qui interdit la suppression d'un type qui est présent dans le dépôt.

Enfin, à tout moment, le module de construction observe le **modèle de l'exécution** fourni par la couche de contrôle et relève les incohérences avec **la description de gestionnaire souhaité.**

Il est capable de remédier à certaines erreurs. Par exemple, lors de la désinstallation d'une implémentation, il peut si la description de la tâche fait référence à un type trouver une implémentation alternative. De même, si une nouvelle implémentation apparaît elle peut être comparée à l'existante pour éventuellement remplacer la tâche. De plus, si une tâche est bloquée, il peut décider de sa désinstallation et choisir une autre implémentation. Enfin, il peut être implémenté pour interpréter et réagir aux messages d'administration envoyés par les tâches.

Il informe l'administrateur ou le module d'auto-adaptation si d'autres problèmes surviennent. D'autre part, il fournit une représentation synthétique qui lie modèle de l'exécution et description abstraite, ce qui permet le raisonnement sur le comportement du gestionnaire.

3.2 ADAPTATION

Notre framework permet donc la construction et la configuration de gestionnaires autonomiques. Les informations fournies par les couches inférieures doivent permettre son adaptation.

L'adaptation du gestionnaire peut se faire de deux façons (figure 60) :

- **manuelle** : à l'aide d'une interface graphique ou de script, l'administrateur vérifie le fonctionnement du gestionnaire et prend les décisions nécessaires.
- **automatique** : un module spécialisé analyse et modifie le gestionnaire. Il existe toute une gradation d'automatisations de cette adaptation. Le gestionnaire autonomique est vu ici comme l'application à adapter ; le module d'adaptation est lui-même un gestionnaire autonomique. Il n'y a pas d'implémentation générique possible de cette adaptation ; le framework ne peut que fournir les touchpoints nécessaires à la modification du gestionnaire.

Nous verrons que notre implémentation fournit une interface graphique qui permet à l'administrateur d'observer en temps réel le comportement du gestionnaire.

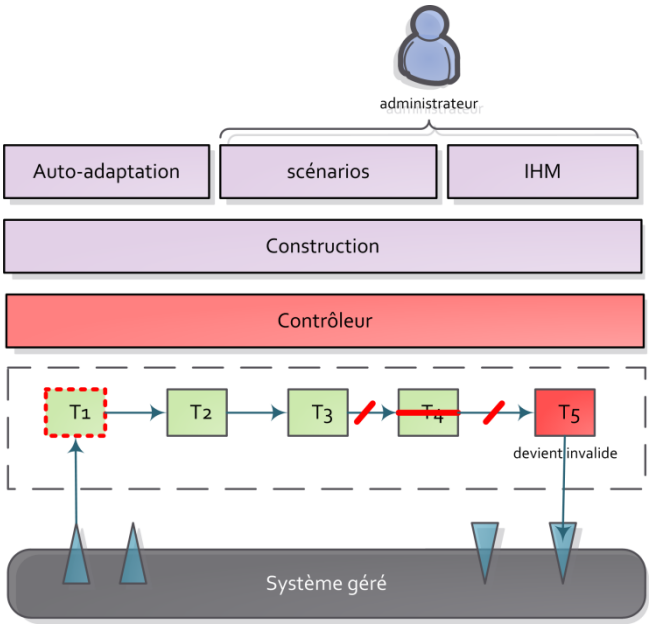

Figure 60 : Adaptation du gestionnaire

Pour faciliter l'administration, une partie de l'adaptation peut être réalisée à partir de scénarios. Chacun d'eux décrit une configuration du gestionnaire autonomique et influence la façon dont les tâches coopèrent. Par exemple, un scénario peut mettre l'accent sur la gestion de la mémoire en privilégiant les tâches peu gourmandes, tandis qu'un autre privilégiera le processeur. Cela permet de décrire la configuration souhaitée du gestionnaire en fonction d'un contexte particulier.

Sur la base de ces scénarios construits par un expert, il est aisé de concevoir un module sensible au contexte qui déclenche le scénario au moment approprié. Par exemple, il est possible de réduire la consommation du gestionnaire pendant la journée et de permettre des analyses plus poussées de nuit. C'est un premier pas vers l'automatisation.

Un module d'auto-adaptation doit être capable de transformer les objectifs de haut-niveau spécifiés par l'administrateur en modifications de l'ensemble des tâches présentes et de leur agencement. La réalisation d'un module d'adaptation est une opération complexe qui justifierait à lui seul un travail de recherche. Nous ne donnerons que des exemples simples. La mise en œuvre de scénarios est notamment une version très sommaire de transformation d'objectifs de buts en actions. Ils peuvent être utilisés avec un cron pour automatiser les modifications.

Il faut souligner que sans adaptation le gestionnaire fonctionne correctement. Les adaptations de son comportement sont des événements ponctuels dans la vie d'un gestionnaire. Dans la plupart des cas, une automatisation complète de l'adaptation du gestionnaire n'est pas nécessaire car le nombre de tâches impliquées reste contenu. L'ajout des scénarios nous paraît suffisant. Dans les cas plus complexes, il sera nécessaire de développer un module ad-hoc chargé d'observer l'activité des tâches. La notion de type de tâches offre ici une abstraction qui permettra de rendre ce code aussi peu dépendant que possible de l'implémentation effective du gestionnaire.

4 TACHES COMPOSITES

L'architecture que nous avons présentée permet l'implémentation d'un framework pleinement fonctionnel. Nous introduisons ici un raffinement qui permet de prendre en compte les problèmes posés par le passage à l'échelle : les composites (figure 61).

Le passage à l'échelle est la raison essentielle des composites. Lorsque le nombre de tâches est important ou qu'elles traitent des problèmes variés et complexes, le nombre de contributions risque d'engorger rapidement les moyens de communication. De même, avec une seule base de données, le partage d'information entre les tâches devient complexe. Enfin, avec une gestion des conflits centralisée sur un seul arbitre, la configuration de l'algorithme de sélection devient difficile.

Les composites sont également un outil pour masquer et filtrer des informations à certaines tâches et pour déployer et désactiver des pans entier de gestionnaires. Ils permettent donc de raisonner avec des blocs de granularités plus fortes lorsque cela est nécessaire.

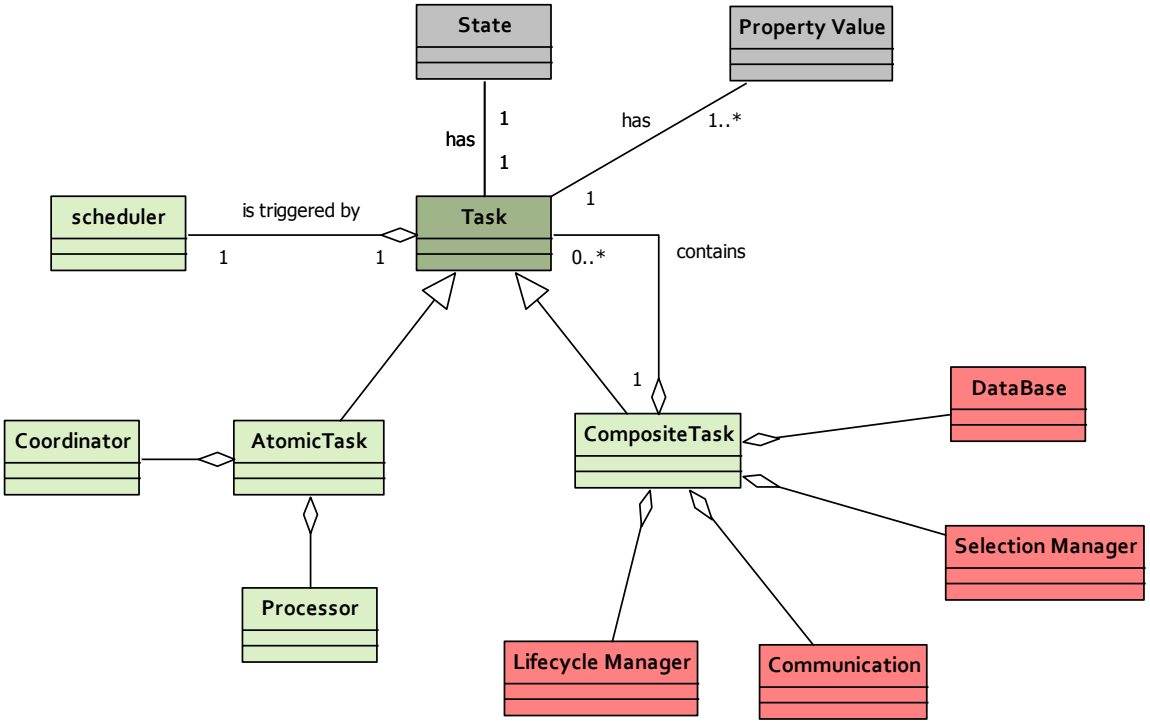

Figure 61 : Représentation d'un composite

Le composite est une unité d'encapsulation qui permet la formation de groupes de tâches cohérents, soit parce qu'elles jouent le même rôle, soit parce qu'elles sont en conflit, soit parce qu'elles s'occupent d'un problème particulier.

Un premier découpage consiste par exemple à regrouper les tâches de monitoring, décision et exécution. On retrouve alors les blocs de granularités fortes que sont les blocs MAPE traditionnels. Il est alors par exemple possible de remplacer rapidement un bloc d'analyse pour changer radicalement la façon dont le gestionnaire apporte des solutions.

Un autre découpage – qui peut s'utiliser de façon complémentaire – consiste à regrouper les tâches qui sont en conflit, pour pouvoir se consacrer à cette gestion avec un mécanisme de sélection dédié. Il est entre autres envisageable de regrouper deux tâches de planification dont l'activation dépend d'un contexte particulier surveillé par le mécanisme de sélection (la présence d'une alarme par exemple). Plusieurs tâches en conflit qui réalisent les mêmes fonctionnalités peuvent ainsi être considérées et utilisées de façon transparente comme une seule tâche.

Un dernier découpage consiste à réunir les tâches chargées de la réalisation d'un objectif particulier. Par exemple, un ensemble de tâches peut être regroupé parce qu'il s'occupe de l'optimisation de la batterie, tandis qu'un autre veille à la mémoire. Selon les objectifs à réaliser, tel ou tel groupe sera inhibé.

Du point de vue des tâches qui coopèrent avec une tâche composite et du point de vue du mécanisme de sélection, une tâche composite se comporte donc comme n'importe quelle autre. L'architecture d'une tâche composite emprunte des éléments de l'infrastructure de contrôle et de l'architecture des tâches (Figure 62).

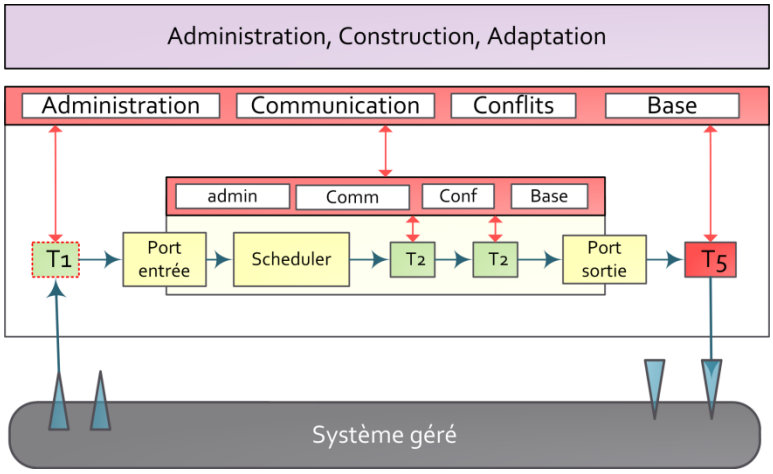

Figure 62 : Un composite interagit comme une autre tâche

Un composite est donc composé de sept éléments suivants.

Le port d'entrée et le port de sortie sont configurables et permettent de définir la nature des messages qui seront échangés entre l'intérieur et l'extérieur du composite. Nous avons cependant limité les modifications possibles en les soumettant aux types de la tâche composite. Ainsi seules les données décrites dans le type de façon optionnelle peuvent faire l'objet de modifications. De cette façon, nous forçons le développeur à garantir un minimum de stabilité dans le comportement de la tâche.

Le scheduler fonctionne sur le modèle que nous avons décrit pour les tâches atomiques. Il filtre les données et décide du moment opportun de leur traitement.

Le module d'administration offre les interfaces nécessaires à la modification de l'ensemble des tâches. En cela, il correspond en tout point au module d'administration que nous

avons présenté. Il permet l'observation de l'état des tâches et fournit une partie de l'architecture en exécution.

Le module de communication est très semblable à ce que nous avons décrit. Cependant, il faut souligner ici que les moyens de communication utilisés en interne et en externe ne sont pas nécessairement les mêmes. Les ports du composite requièrent la prise en compte des deux types de communication et les tâches déployées à l'intérieur du composite doivent toutes supporter le moyen de communication interne.

Le module de gestion de conflits joue ici deux rôles : mécanisme de sélection classique en interne et coordinateur pour l'extérieur. Il soumet donc des requêtes à ses composites parents. Nous établissons ici une distinction entre les données produites en interne qui ne nécessitent pas d'autorisation et les données de provenance externe. Pour ce faire, le mécanisme de sélection reçoit une copie des entêtes des données provenant de l'extérieur. Une élection se déroule en plusieurs étapes :

1. Lorsqu'il reçoit un ensemble de requêtes en provenance des tâches qui le composent, le composite procède à une élection. Cette dernière permet de déterminer quelles données les tâches du composite souhaitent utiliser.
2. Il calcule une nouvelle requête qu'il soumet à son père et attend une autorisation.
3. En fonction du résultat, il calcule les autorisations définitives et les nouveaux états des tâches.
4. Il distribue les résultats.

Cet ordre peut varier en fonction de l'implémentation des mécanismes de sélection et de leur complexité. L'expérience nous a montré que c'est celui qui engendre le moins de famine et offre le plus de souplesse. L'implémentation d'un mécanisme de sélection pour un composite est donc une tâche plus complexe du fait de la communication avec le parent. Cependant il est possible de proposer un mécanisme générique qui prend en charge la communication avec le parent, ce que nous avons fait dans notre implémentation de sorte que la tâche du développeur demeure relativement aisée.

La base de données est distribuée et chaque composite possède une version localisée des informations qui concernent les tâches qui le composent. De par leur type, les composites connaissent l'ensemble des propriétés partagées. Seuls les changements sur ces propriétés sont répercutés aux parents, de sorte que chaque base gère un nombre limité de données. Lorsqu'une tâche essaye d'accéder à une donnée qui n'est pas présente dans la base locale, cette dernière se charge automatiquement de récupérer la donnée chez son parent.

Pour résumer, un gestionnaire autonomique peut être vu comme une version particulière d'un composite. Du point de vue de la construction des gestionnaires, l'intégration des composites ne change rien. Nous verrons que dans l'implémentation, nous avons choisi de garder un seul point centralisé d'administration car cette dernière nous parait de cette façon plus aisée. Nous n'avons donc qu'une seule représentation de l'architecture souhaitée et de l'architecture en exécution. La distribution est cependant envisageable et peut s'avérer intéressante pour auto-adapter des parties seulement de la boucle.

5 SYNTHESE

Dans ce chapitre nous avons décrit l'architecture générale de notre framework. Notre objectif est d'offrir le maximum de flexibilité au développeur en lui permettant de modifier chacun des aspects essentiels du gestionnaire.

D'abord nous avons présenté l'architecture des tâches d'administration qui repose sur cinq types d'éléments remplaçables, configurables et extensibles, gérant chacun un aspect particulier du comportement de la tâche. Les ports d'entrée et de sortie gèrent la communication avec le monde extérieur ; les tâches communiquent de façon asynchrone via l'échange de messages qui contiennent des données. Le scheduler planifie les traitements et décide de l'activation de la tâche. Le coordinateur vérifie que la tâche a les autorisations nécessaires pour s'exécuter. Et enfin, le processeur, cœur de la tâche, effectue les traitements spécifiques.

Pour faciliter le travail du concepteur et augmenter le niveau d'abstraction, nous avons introduit les notions de types de tâches et de données. Elles permettent de définir l'architecture d'un gestionnaire en faisant abstraction de la plateforme d'exécution et des techniques utilisées. Le concepteur peut ainsi se focaliser sur les fonctionnalités des tâches sans se soucier des contingences techniques. Les types de données garantissent un minimum de cohérence dans les échanges entre tâches.

Ensuite, nous avons détaillé l'infrastructure de contrôle des tâches qui offrent les mécanismes nécessaires à la collaboration des tâches. Ces mécanismes comprennent la communication, la supervision et l'administration, les mécanismes de sélection et une base de données permettant le partage d'informations persistantes. Nous avons ainsi discuté de l'implémentation de chacun de ces mécanismes.

Le mécanisme de sélection est l'un des plus importants car il permet la gestion des conflits entre tâches. Nous avons souligné qu'il était facultatif. Les tâches ne demandent d'autorisation que lorsque la situation l'exige. Les conflits peuvent être réglés en amont ou en aval. En amont, nous avons discuté des différentes possibilités allant du filtrage à la négociation, en passant par l'arbitrage. Dans de nombreuses situations ces méthodes sont fonctionnellement équivalentes, par la suite nous mettrons d'avantage l'accent sur l'arbitrage. Cette méthode consiste à utiliser un arbitre qui décide quelles tâches pourront s'activer en fonction d'une configuration fournie par le concepteur du gestionnaire. En aval, la résolution des conflits peut se faire par l'utilisation d'une tâche de synthèse qui, à partir des informations fournies par les autres tâches, choisit la meilleure solution. Dès lors, il n'y a plus réellement conflit mais coopération. Nous avons présenté quelques mécanismes de sélection et l'architecture permet au développeur de les étendre ou d'en proposer de nouveaux.

Enfin, nous avons présenté les outils d'administration du gestionnaire. Nous avons notamment expliqué comment celui-ci est construit. Nous proposons l'utilisation de deux modèles. L'un représentant l'architecture souhaité, l'autre représentant l'architecture en exécution. L'administrateur ou l'expert se focalise sur la modification du modèle souhaité qui est plus abstrait, tandis que le module de construction du gestionnaire se charge de trouver les implémentations des tâches à utiliser. Le modèle de l'architecture en exécution contient un ensemble de statistiques sur l'exécution des tâches et permet notamment d'identifier les tâches bloquées ou inappropriés.

Cet ensemble d'informations peut être utilisé par l'administrateur ou par un module d'auto-adaptation pour modifier dynamiquement le comportement du gestionnaire. Nous proposons notamment la modification du gestionnaire via des interfaces standards sur lesquelles il est possible de construire une interface d'administration graphique.

Nous n'avons pas développé les techniques d'auto-adaptation du gestionnaire, bien qu'elles nous paraissent essentielles, car elles représenteraient en elles-mêmes un travail de

recherche indépendant. Le framework que nous avons présenté permet déjà la conception de gestionnaires flexibles.

Dans les chapitres suivants, nous détaillerons une implémentation particulière de ce framework nommée Ceylan, et son utilisation pour la construction d'un gestionnaire.

Chapitre 6 - **Réalisation**

Ce chapitre présente notre implémentation du canevas présenté.

Dans un premier temps nous abordons la question des technologies utilisées. Dans le chapitre précédent nous nous sommes efforcé de rester très générique et de faire le moins possible référence à une technologie d'implémentation. Il existe de nombreuses solutions permettant d'implémenter le framework, éventuellement de façon incomplète ou peu dynamique. Nous avons développé un framework complet et mis l'accent sur la facilité de développement et le dynamisme, ce qui nous conduit à proposer un modèle à composants basé sur OSGi et iPOJO.

Par la suite, nous exposons l'implémentation des tâches, atomiques d'abord avec la présentation des ports, schedulers et coordinateurs, puis composites avec notamment l'implémentation des mécanismes de sélection. Tout au long de la description nous donnons des extraits de l'ADL qui permet de décrire l'architecture du gestionnaire souhaité.

Enfin nous présentons les outils d'administration de la plateforme et notamment l'interface graphique qui fournit une représentation de l'architecture en exécution et permet sa modification dynamique.

1 MODELE DE DEVELOPPEMENT

1.1 APPROCHE A SERVICES ET MODELE A COMPOSANTS

Dans le chapitre précédent nous avons défini le modèle conceptuel de notre framework. L'architecture présentée peut s'implémenter avec des technologies extrêmement variées qui lui confèreront des forces et des faiblesses différentes. Nous avons par exemple évoqué l'utilisation de services OSGi, de services Web, de simples composants et proposé de se baser directement sur la couche réseaux ou utiliser des outils spécialisés pour la communication des tâches.

Notre objectif est de proposer une implémentation efficace et dynamique de l'architecture tout en faisant bénéficier les développeurs d'un modèle de développement simple, qui leur permet de se consacrer à la réalisation des fonctionnalités autonomiques sans se soucier du dynamisme. Deux choix sont particulièrement importants pour l'implémentation du framework :

- **le modèle d'interaction**. Une caractéristique majeure de notre framework est le dynamisme. Le modèle d'interaction choisi doit permettre la modification dynamique de l'ensemble de tâches et de la configuration globale du framework.
- **le modèle de programmation**. Le développement de gestionnaires doit être le plus simple possible pour que le développeur puisse se focaliser sur les fonctionnalités de la tâche et non sur la gestion d'aspects orthogonaux.

Le dynamisme et la forte indépendance entre les tâches nous incitent à utiliser l'approche à services comme **modèle d'interaction**. Dans le domaine de l'informatique autonomique les approches servent souvent de cas d'application ou sont utilisées pour gérer l'hétérogénéité. Nous proposons, ce qui est plus rare dans le domaine, de nous appuyer sur les propriétés des architectures à services pour apporter du dynamisme : en particulier celui nécessaire à l'implémentation des tâches.

L'implémentation des tâches en tant que services permet de garantir un faible couplage entre elles. Elles peuvent ainsi être déployées, installées, utilisées et détruites sans impact sur les autres tâches. Nous verrons que les ports d'entrée et de sortie de chaque tâche proposent des services qui permettent d'établir une communication entre les tâches.

Le choix du **modèle de programmation** utilisé est important. Dans le chapitre précédent, nous avons toujours fait mention des constituants de la tâche ou du contrôleur en termes de module pour éviter de faire référence à un modèle de programmation particulier. Ces modules peuvent être développés de plusieurs façons en utilisant par exemple des modules de type ADA, des objets, ou des composants. Il est naturellement possible d'utiliser chacune de ces technologies mais l'effort de développement ne sera pas le même. Nous avons décidé de produire un modèle à composants pour les raisons qui suivent.

Chaque module d'une tâche doit pouvoir être développé, testé et configuré indépendamment des autres modules. L'objectif est de fournir un ensemble de modules réutilisables qui accélèrent le développement de la tâche. L'assemblage de ces modules doit se faire le plus tard possible. Pour ces raisons, l'utilisation de composants s'impose pour le développement de chacun des modules que nous avons décrit précédemment. Les objets ou les scripts sont des unités de développement trop petites ou mal définies. Dans notre implémentation, les modules seront implémentés sous forme de composants déployés sur une plateforme à services.

Les développeurs de tâches doivent pouvoir se focaliser sur le développement des algorithmes de gestion uniquement. Les utilisateurs du framework ne devraient pas gérer le dynamisme et les aspects spécifiques à la plateforme. Il leur faut déjà être des experts dans le système dont le gestionnaire à la charge : avoir les connaissances suffisantes pour être capable de recueillir les bonnes données au bon moment, décider que suffisamment de données ont été

reçues, effectuer des analyses pertinentes, et décider des actions à effectuer. Cette tâche est en soi déjà très complexe. On ne peut demander une double compétence et la prise en charge du dynamisme.

L'expert qui assemble le gestionnaire et le configure ne devrait pas avoir à prendre en considération des problématiques liées à la plateforme d'exécution. Il faut qu'il puisse exprimer la composition des modules d'une tâche et décrire l'ensemble de tâche du gestionnaire avec un minimum de références à la plateforme d'exécution sous-jacente. L'architecture de la tâche devrait être décrite uniquement en termes spécifiques au domaine : scheduler, ports, coordinateurs, arbitres par exemple.

Idéalement, la réalisation d'un gestionnaire s'appuie sur une bibliothèque de composants existants que le concepteur du gestionnaire utilise et complète. Cette bibliothèque est constituée des composants permettant l'échange d'informations entre les tâches (ports et schedulers), de gérer les conflits (coordinateur et arbitres) et des processeurs permettant la médiation et l'agrégation de données.

Seule la partie spécifique du comportement du gestionnaire doit rester à la charge des développeurs : le développement des processeurs. Les développeurs des processeurs ne sont pas nécessairement les assembleurs et concepteurs du gestionnaire. Leur tâche peut se cantonner à la réalisation d'une portion très spécialisée du gestionnaire, qui sera réutilisée par la suite. Ainsi, l'expert procédant à l'assemblage des différentes portions du gestionnaire n'a pas à posséder de connaissance sur la technologie utilisée.

Notre objectif est donc de proposer un modèle à composants spécifique qui offre un langage d'assemblage spécialisé dans la création des tâches. Ce langage permet de définir les différents constituants des tâches et la configuration de l'infrastructure de contrôle telle que nous l'avons définie dans le chapitre précédent. Ce langage est interprété par l'implémentation du framework qui se charge de l'instanciation du gestionnaire sur la plateforme d'exécution choisie.

Le développement du se divise en plusieurs activités. En utilisant une approche de conception ascendante, ces activités sont les suivantes :

- **définition des types de tâches et de données utilisées.** L'expert identifie les grandes fonctionnalités nécessaires à son gestionnaire et les types de données nécessaires. Cette définition s'effectue dans un langage indépendant de la plateforme.
- **implémentation des sous-modules de la tâche.** Les développeurs implémentent les ports, coordinateurs, schedulers nécessaires à la création des tâches. Ils fournissent une implémentation des processeurs qui réalisent les fonctionnalités décrites. Ces implémentations dépendent naturellement de la plateforme d'exécution sous-jacente, mais la configuration des sous-modules doit se faire dans un langage indépendant.
- **implémentation des sous-modules des tâches composites.** Les développeurs réalisent les arbitres nécessaires à la gestion des conflits. Ici encore, si leur implémentation est propre à la plateforme d'exécution, la configuration doit être indépendante.
- **implémentation des tâches atomiques.** La structure d'une tâche atomique est décrite dans un langage indépendant de la plateforme. Seule la référence vers les implémentations des sous-modules est donnée. Une implémentation de tâche peut être instanciée à plusieurs reprises.
- **implémentations du gestionnaire et des tâches composites.** Cette description également indépendante de la plateforme fait référence aux types définis précédemment. Elle définit un ensemble d'implémentations de tâches composites qui pourront être utilisées plusieurs fois. Le gestionnaire est une tâche composite particulière.

Ainsi, seules les implémentations des sous-modules sont fortement dépendantes de la plateforme d'exécution. La force d'un modèle à composant est de séparer fortement les concepts propres à la plateforme d'exécution des concepts propres au domaine. Si le modèle à composants que nous proposons est porté sur plusieurs plateformes, la même définition de gestionnaire pourra être utilisée sans modification car elle est indépendante de la plateforme.

1.2 PLATEFORME D'EXECUTION UTILISEE : OSGI, IPOJO ET CILIA

Notre implémentation se base sur une pile de technologies illustrée par la figure 63. Nous discutons de ces choix dans ce qui suit.

Le développement intégral d'une solution ad-hoc n'est pas concevable compte tenu de la complexité des mécanismes et des interactions entre les composants de notre framework. Il existe de nombreuses plateformes à services. Implémenter l'architecture c'est faire un choix guidé par les performances souhaitées, le besoin de dynamismes, la volonté de réutiliser un patrimoine existant, la distribution envisagée et bien sûr par la capitalisation des connaissances. Aussi avons-nous naturellement choisi de réaliser notre framework au-dessus de la plateforme OSGi pour les raisons subséquentes.

D'abord, le dynamisme et la capacité de modifier le gestionnaire dynamiquement nous apparaissent essentiels. La réalisation ad-hoc de protocole de découverte et de déploiement de tâches nous semble délicate, compte tenu de ces mécanismes. Or, la plateforme d'exécution doit permettre l'installation et la désinstallation de modules dynamiquement à l'exécution pour autoriser l'ajout de tâches. Nous avons évoqué dans l'état de l'art deux plateformes matures offrant cette propriété : Fractal et OSGi. Nous préférons OSGi à Fractal pour deux raisons. D'une part les architectures à services permettent la découverte dynamique de nouveaux services, ce qui facilite l'implémentation d'un dépôt. D'autre part, l'équipe ADELE a une forte expertise sur les architectures à services et sur les plateformes à services OSGi ; de plus nous disposons d'applications domotiques qui pourront être utilisées pour la validation.

Ensuite, nous souhaitons concevoir des gestionnaires réactifs, ce qui exclut les technologies consommatrices de puissance de calcul pour les échanges telles que les web services. Pour être performants, les appels entre tâches doivent être aussi directs que possible avec un minimum de coût en communication. Le coût d'un appel entre deux services OSGi est très comparable à un appel de méthode Java. Il faut également souligner que OSGI fournit un bus à événements complet : Event Admin, ce qui permet de tester les deux approches de communication, directe et indirecte.

En outre, la distribution n'est pas prise en compte dans l'implémentation du framework que nous proposons. Le principal défaut d'OSGi n'est donc pas un problème dans notre cas. Il faut également noter que de nombreux travaux visent à permettre des communications transparentes entre services de plateformes et de technologies différentes (SAM[108], ROSE[18]).

Enfin, les versions actuelles d'OSGi sont implémentées en Java. C'est est un langage très populaire qui offre beaucoup de librairies facilitant le développement. Par exemple, de nombreuses librairies permettent d'analyser des fichiers XML. D'autres sont consacrées à la gestion de la concurrence et permettent la réalisation simple de verrous, files bloquantes ou rendez-vous. Des technologies comme JMX qui permettent d'administrer les applications, d'observer des attributs et d'appeler des méthodes à distances sont également intéressantes pour notre implémentation. Nous utiliserons la version la plus récente de Java.

D'un point de vue technologique la plateforme à services OSGi nous apparaît donc idéale pour l'implémentation du framework.

[18] http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

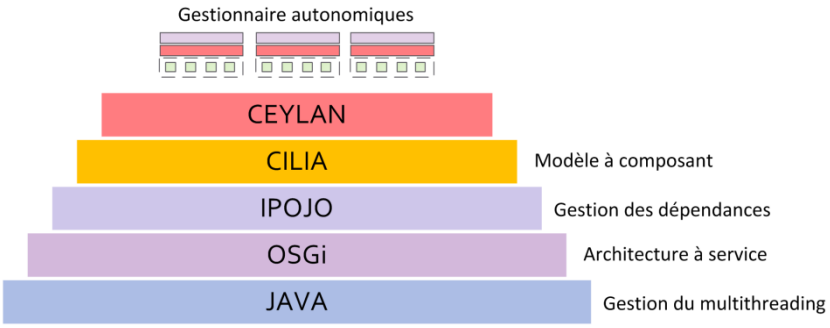

Figure 63 : Technologies utilisées par l'implémentation de notre framework : CEYLAN

Mais la gestion de la disponibilité dynamique est encore trop complexe sur OSGi, car il faut la gérer dans chaque service en mêlant code fonctionnel et non fonctionnel. Le développement d'un modèle à composant au dessus d'OSGi requerrait trop d'efforts. Dans notre approche, le dynamisme est présent dans chaque module du framework. Prenons l'exemple de la tâche : les ports d'entrée et de sortie doivent découvrir d'autres tâches et s'adapter à leur arrivée, ce qui requiert des vérifications dans le registre de services ; le scheduler découvre et utilise la base de données ; le coordinateur doit envoyer ses requêtes au mécanisme de sélection qu'il ne connait pas avant le déploiement ; et enfin le processeur également utilise la base de données. La gestion de ces échanges s'avère rapidement complexe et fastidieuse car il faut réagir adéquatement à l'apparition et la disparition de nouvelles tâches.

Pour cette raison, nous avons choisi d'utiliser au-dessus d'OSGi le modèle à composants orienté service iPOJO décrit dans l'état de l'art. Ce modèle de programmation à composants orienté service permet la mise en œuvre d'une architecture à services en utilisant des composants fournissant des services. Ils sont faiblement couplés et communiquent en respectant le paradigme à services via le service de registres fourni par la plateforme.

iPOJO possède de nombreuses caractéristiques intéressantes pour l'implémentation de notre framework.

Il sépare clairement les aspects fonctionnels et orthogonaux dans le développement en introduisant la notion de conteneur. Chaque composant est constitué de deux parties, le cœur qui réalise la fonctionnalité des services et la membrane qui se charge des aspects non fonctionnels ou redondants.

La membrane intègre un composant spécialisé (figure 64) fourni par iPOJO, dont le rôle principal est d'automatiser la gestion des dépendances de services. iPOJO masque la complexité de l'interaction des services de l'approche à services. Il fournit une machine d'injection et d'introspection de méthodes et propriétés qui lui permet notamment d'injecter automatiquement les dépendances entre services. Alors que la gestion d'une dépendance de services peut réclamer des centaines de lignes sur une plateforme OSGi simple, sur iPOJO elle en demande moins d'une dizaine. Cette propriété est indispensable compte tenu du nombre de dépendances de services à gérer pour notre implémentation.

Mais la gestion des dépendances n'est pas le seul service rendu par la membrane. L'une des forces d'iPOJO est la possibilité d'étendre et de modifier cette membrane. Les composants appartenant à la membrane s'appellent les *handlers* (figure 64). Un *handler* est un composant qui prend en charge un aspect récurrent du traitement d'une application ou d'un domaine particulier. Ils sont développés indépendamment et peuvent être déployés séparément sur l'infrastructure d'exécution. Tous les handlers ont accès à la machine à injection et peuvent lire ou modifier la valeur des attributs et utiliser les méthodes fournies et utilisées par le cœur du composant : le POJO.

Ces deux notions de conteneur et de handler (figure 64) permettent de simplifier la gestion du cycle de vie des sous-modules de la tâche. Gérer chaque module indépendamment est une tâche ardue. Lorsqu'une tâche est créée, il faut garantir la création de chacun des sous-composants (ports, scheduler, coordinateur entre autres) et lorsqu'elle est détruite la destruction de chacun d'eux. Si un module est invalide, que sa configuration est incorrecte ou incomplète, c'est l'ensemble de la tâche qui doit être considéré comme invalide. Il va de soit que cette gestion ne peut être laissée au développeur de la tâche car c'est une activité redondante et complexe qui conduit à des erreurs. iPOJO facilite grandement la gestion de cet aspect.

Dans la section suivante, nous montrons comment l'utilisation du conteneur et des *handlers* permet de remédier à ce problème. Nous verrons que la membrane peut gérer la création et le cycle de vie de la tâche et nous discuterons de l'implémentation des ports, scheduler et coordinateur. Les implémenter sous forme de handlers est une des solutions car ils réalisent un aspect particulier et récurrent dans le domaine que nous avons délimité. Le processeur, la partie spécifique de la tâche, est encapsulé dans la membrane formée par ces composants. La tâche peut alors être considérée comme un composant unique qui sera être créé, utilisé, détruit comme n'importe quel autre composant.

Un autre avantage d'iPOJO est qu'il définit une séparation claire entre les types de composants et leurs instances. Pour chaque type de composant, iPOJO crée un service appelé fabrique. Chaque type de composant peut alors être facilement instancié plusieurs fois en utilisant la fabrique associée à son type. Ces fabriques sont découvrables et servent de base au dépôt de tâches dont nous avons parlé. Dans notre implémentation, nous nous servons de ces services particuliers pour découvrir de nouvelles tâches et de les créer.

Enfin, il faut souligner qu'iPOJO a beaucoup été étendu pour supporter des technologies variées et complexes. Par exemple, il existe, aujourd'hui des handlers pour exposer et modifier des propriétés de configuration par injection, envoyer et recevoir des événements de façon transparente, exposer et publier des méthodes via JMX, fournir des logs, ou encore exposer un service OSGi sous la forme d'un service Web. Cette boîte à outils facilite considérablement le développement non seulement du framework mais aussi des processeurs. Les développeurs des processeurs peuvent s'appuyer sur ces handlers pour utiliser des technologies complexes.

iPOJO, de part son extensibilité, est donc une plateforme idéale pour le développement d'un nouveau modèle à composants. D'ailleurs, des travaux parallèles à cette thèse se servent d'iPOJO pour implémenter un modèle à composants spécifique à la médiation : Cilia.

Si les premières versions du prototype que nous avons développé se basaient directement sur iPOJO ([144][145][146]), celle que nous présentons dans la suite est construite au-dessus de Cilia. En effet, le modèle du médiateur de Cilia est très proche du modèle de la tâche. Un médiateur Cilia (figure 64) est constitué de cinq éléments : un ou plusieurs ports d'entrée et sortie, un scheduler, un processeur et un dispatcher.

Le rôle de ces éléments est sensiblement le même que ceux développés dans le chapitre précédent. Cilia ne propose pas de coordinateurs car les chaines de médiations ne sont pas soumises à conflits et il ajoute la notion de dispatcher qui permet d'orienter les données vers un port de sortie particulier. Dans notre implémentation, nous ne nous en servirons pas puisque nous avons décidé de limiter le nombre de ports de sorties.

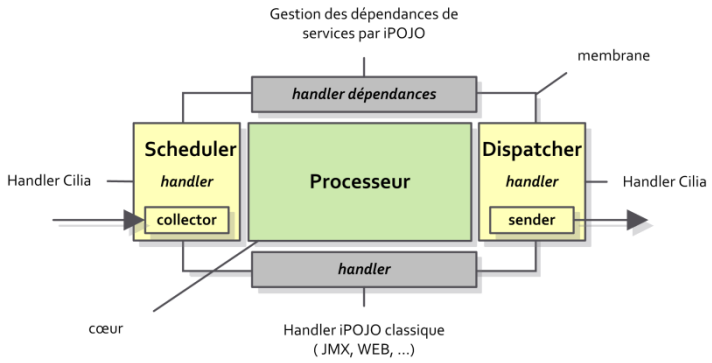

Figure 64 : Un médiateur Cilia sous la forme de composant iPOJO

Dans Cilia, l'architecture des médiateurs constituant une chaine de médiation est spécifiée dans un fichier de description d'architecture. Ce fichier décrit l'architecture des médiateurs en termes de différents composants de médiation et leurs propriétés de configuration. L'extrait de code figure 65 illustre l'architecture d'un médiateur Cilia :

```
<cilia version="1.0">
 <mediators>
  <mediator id="MyMediator">
   <collector ref="jms" topic="apps/topic"/>
   <scheduler ref="periodic-scheduler" delay="5" time-unit="SECOND"/>
   <processor component="org.example.mediation.MyProcessor"/>
   <dispatcher ref="default-scheduler"/>
   <sender ref="jms" topic="MyMediator/topic"/>
  </mediator>
 </mediators>
</cilia>
```

Figure 65 : Déclaration d'un médiateur dans Cilia

Cet extrait montre comment Cilia permet de faire abstraction de la plateforme d'exécution. La description de la structure du médiateur se fait uniquement en des termes métiers appartenant au domaine de la médiation. Nous verrons dans la suite que nous implémentons les tâches sous la forme de médiateur Cilia. Nous proposons un langage spécifique aux concepts du domaine des tâches proche de celui de Cilia.

Cilia est un projet récent qui ne fournit pour l'instant que des implémentations réduites et spécifiques à la médiation des schedulers et ports. En particulier les données et les médiateurs ne sont pas typés. Toutefois, la logique de communication entre ces composants est déjà implémentée et robuste. D'autre part, comme il s'appuie sur iPOJO, Cilia est extensible et permet de redéfinir et de réutiliser les implémentations existantes et de rajouter le coordinateur. En étendant Cilia, nous bénéficions de l'ensemble des outils de monitoring et de tests en cours de développement pour la plateforme.

Dans les sections suivantes, nous détaillons l'implémentation de chacun des constituants de notre framework.

2 IMPLEMENTATION D'UNE TACHE

Dans cette section, nous discutons de l'implémentation de la tâche et de ses sous-modules. La mise en œuvre de l'architecture de la tâche réclame des choix qui auront un impact sur les performances générales du système dans lequel le framework est déployé.

Dans un premier temps nous abordons la déclaration des types de données et de tâches qui sont par la suite utilisés par l'ensemble des composants de notre framework. Puis nous décrivons la déclaration d'une implémentation de tâche et son ADL.

Ensuite, nous nous intéressons à la façon d'implémenter les sous-modules de la tâche. Nous verrons que chacun des ports, coordinateurs et schedulers peuvent être développés sous forme d'un composant autonome proposant un service, ou d'un handler.

Par la suite, nous détaillons l'implémentation de chacun des sous-modules de la tâche : les ports, le scheduler, le coordinateur et le processeur.

2.1 TYPE DE DONNEES ET DE TACHES

Avant d'aborder l'implémentation des tâches, présentons d'abord les mécanismes de gestion des types. Les modèles de types de tâches et les types de données sont gérés par un service spécialisé déployé sur la plateforme, de sorte que chaque gestionnaire puisse accéder à leur description.

Le service maintient une représentation des types sous forme d'arbre. La déclaration des types peut se faire de trois façons. D'abord de façon programmatique en utilisant l'API proposée. Le service publie une API qui comprend des primitives pour consulter, ajouter et supprimer des types. De même qu'une classe Java ne peut pas être modifiée après son déploiement, nous interdisons la modification d'un type. Ensuite, en utilisant l'IHM (que nous détaillons plus loin) qui fournit une représentation arborescente des types. Enfin en utilisant une description XML.

Les fichiers XML peuvent être chargés directement via une commande OSGi ou en passant par l'interface graphique (figure 66). L'analyse des fichiers XML est réalisée par un service indépendant qui s'appuie sur l'API du service de gestion des modèles. Ce service est implémenté au moyen de l'API StAX fourni par Java. Cette séparation permet de faire évoluer le format XML indépendamment de la représentation interne. Ce service est également capable de sauvegarder une représentation en XML des types chargés par le service de gestion des types. Ceci permet de sauvegarder dans un fichier les modifications effectuées via l'IHM.

Comme nous l'avons expliqué dans le chapitre précédent, la hiérarchie de types de donnée permet de spécifier les types des données qui transitent entre les différentes tâches du gestionnaire. Un type de donnée possède un nom, un type père, ainsi que des clés, avec leur valeur par défaut, qui spécifient les données contenues dans le type de donnée. Un sous-type de donnée doit posséder au minimum toutes les clés de son père, et peut en ajouter de nouvelles.

La figure 66 illustre la déclaration d'un type de donnée ALARM dont le père est le type ANALYSE, il est constitué d'une clef à laquelle s'ajoutent les clefs du parent. Sa durée de validité est de 10 secondes comme l'indique l'attribut « *expire* ». Dans notre implémentation les données des clefs sont *Serializable* et peuvent donc être initialisées avec une valeur textuelle, comme dans le cas où la valeur de l'attribut *on* est *true*. Nous n'avons pas implémenté de mécanisme pour typer les attributs car cela ne présente pas d'intérêt particulier pour notre prototype. A l'initialisation, la valeur par défaut est affectée si possible, une erreur déclenche une exception spécifique. Il serait facile d'ajouter ce mécanisme en s'appuyant sur les types fournis par JAVA.

Les types de tâche sont plus complexes. Outre une description de sa fonctionnalité, chaque type de tâche déclare les types d'entrées et de sorties, les propriétés de configuration et les propriétés partagées. Les types de tâche sont donc liés aux types de données. Lors de la

déclaration d'un type de tâche, le service des modèles vérifie que les types de données mentionnés existent. Il déclenche une exception si ce n'est pas le cas.

La figure 66 illustre également la déclaration d'un type de tâche. Le type *Alarm* étend le type *plan,* ce qui signifie qu'il partage les mêmes propriétés et qu'il respecte les contraintes sur les données fixées dans son parent. La conformité est vérifiée à chaque déclaration de type et une exception est déclenchée en cas d'erreur. Dans le cas du type Alarm, par exemple, le type de donnée ANALYSE.DISK est déclaré optionnel ; pour qu'Alarm soit valide, il faut que ce type de donnée ou un de ces parents soit déclaré optionnel dans le type Plan. Les propriétés de configuration peuvent avoir des valeurs par défaut tout comme les propriétés de la base. Cette valeur est assignée au démarrage de la tâche. Un enfant peut écraser la valeur assignée à une propriété du parent en la déclarant de nouveau et en lui affectant une nouvelle valeur.

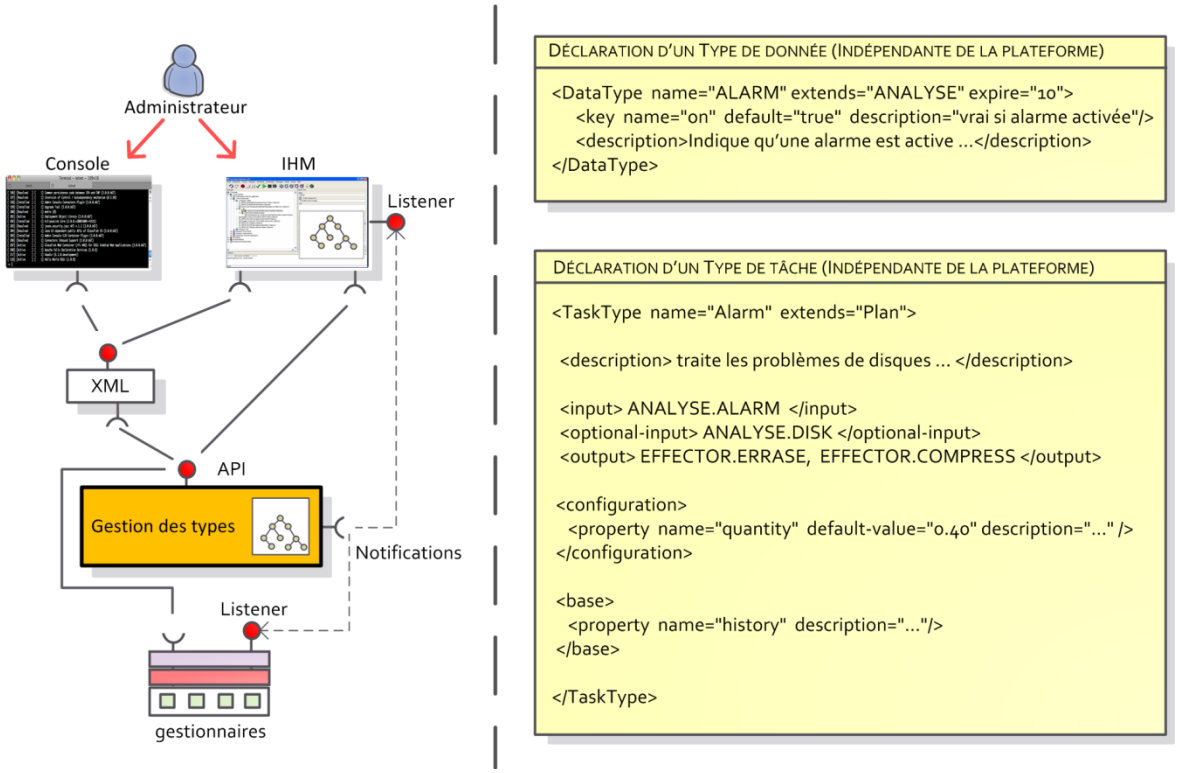

Figure 66 : Modèle de données et de tâches

Nous verrons que les types sont utilisés et vérifiés par les ports et le module d'administration des tâches. Les types sont notamment employés par les implémentations des tâches. Les vérifications se font à l'exécution, il n'y a pas pour l'instant de vérification statique. Cette vérification est réalisable – il serait par exemple possible de vérifier que les types existent lorsqu'ils sont déclarés dans une implémentation – mais elle nous parait trop couteuse à mettre en oeuvre dans le cadre de cette implémentation.

Enfin, le service de gestion offre un service de notifications. La connaissance des types présents sur la plateforme est importante car leur présence conditionne l'existence des tâches et leur création. Les composants déployés sur la plateforme peuvent s'abonner pour recevoir des notifications à chaque ajout et disparition de types. Pour cela ils publient un service *TaskModelListener* ou *DataModelListener*. Le service de gestion des types découvre les abonnées grâce au registre de services fourni par OSGi – la gestion de cette dépendance est assurée par iPOJO. Chaque disparition ou apparition d'un nouveau type fait l'objet d'une notification comme le montre la figure 66.

2.2 NATURE DES SOUS-MODULES D'UNE TACHE

Attachons nous maintenant à la description de l'implémentation des sous-modules de la tâche. Nous nous basons, comme nous l'avons expliqué, sur Cilia et iPOJO. Une première décision consiste à déterminer sous quelle forme seront implémentés les sous-modules de la tâche : handler ou composant externalisé et indépendant ? Cette décision a en effet des conséquences sur les performances, la complexité du code et la capacité à modifier la structure de la tâche à l'exécution.

La première solution consiste à implémenter les sous-modules sous forme de **handlers**. Les sous-modules d'une tâche sont développés sous forme de composants et intégrés dans le même conteneur configurable. Au cœur du conteneur se trouve le processeur qui réalise la fonctionnalité principale. La membrane est quant à elle notamment constituée des ports, du scheduler et du coordinateur.

Implémentés sous forme de handlers (figure 67 (a)), ces derniers sont ainsi définitivement associés au processeur et à la tâche. La gestion de la tâche et de son cycle de vie est grandement facilitée car la tâche s'utilise comme n'importe quel composant. Il n'est pas nécessaire de coder la gestion du cycle de vie du coordinateur, du scheduler et des ports, cette gestion est prise en charge par iPOJO. En cas d'invalidité d'un handler, c'est toute la tâche qui est déclarée invalide.

Pour créer une nouvelle implémentation d'un module, le développeur fait dériver son implémentation d'une version basique existante proposée par une classe abstraite. Cette classe prend en charge les aspects redondants et la communication entre les handlers. Par exemple, pour le scheduler la gestion du tampon est réalisée par la classe abstraite. Le développeur ne doit coder que la méthode spécifique qui effectue la planification des traitements. Le handler ainsi développé peut utiliser d'autres handlers (JMX, Web) et la gestion des dépendances fournie par iPOJO. C'est un point particulièrement important qui facilite le développement des conditions de déclenchement sensibles au contexte et dépendant donc d'autres services.

L'utilisation des handlers a une limitation de taille. L'implémentation des handlers ne peut pas, en effet, être modifiée dynamiquement à l'exécution. Le cœur et la membrane forment un nouveau type de composant dont la structure est figée. Avec cette solution, il n'est donc pas possible de modifier l'implémentation du scheduler lorsque la tâche est active.

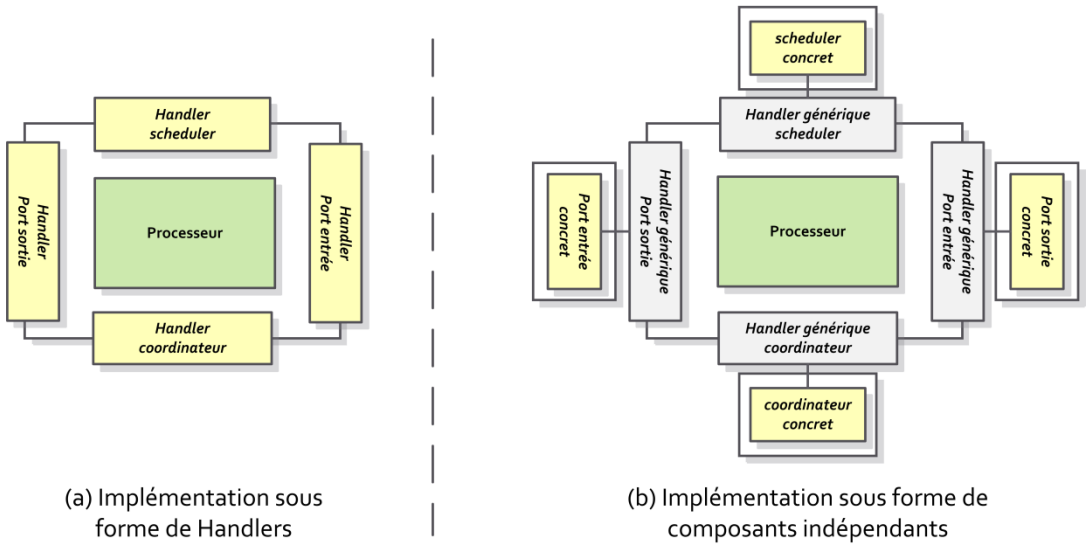

Figure 67 : Implémentation des sous-modules de la tâche

Pour remédier à cela, **une autre solution** consiste à implémenter la gestion du cycle de vie et la communication des sous modules dans la membrane (figure 67 (b)) et d'**externaliser** les sous-modules. Les handlers de la membrane effectuent alors le travail de la classe abstraite évoquée ci-dessus. Les implémentations concrètes sont réalisées sous la forme d'un composant indépendant, qui propose une interface standardisée. Durant l'exécution, les handlers découvrent ces implémentations dans le registre de services et créent une instance. Grâce à ce découplage, l'instance peut être remplacée par une nouvelle instance créée à partir d'une implémentation différente.

Les composants implémentant l'algorithme concret possèdent toutes les caractéristiques d'un composant iPOJO traditionnel. En particulier, ils utilisent la gestion de la dépendance fournie par iPOJO et peuvent employer les autres handlers. Il existe deux conséquences négatives à cela.

D'abord, ces composants sont considérés par la plateforme comme des instances autonomes et leur cycle de vie est donc indépendant de celui du processeur. Les handlers génériques doivent donc prendre en charge la gestion de cycle de vie, ce qui implique : la création des implémentations lors du démarrage de la tâche, leur destruction lors de l'arrêt, la mise en conformité de leur état avec celui de la tâche et réciproquement. Ainsi lorsqu'un composant est invalide (le scheduler par exemple), la tâche et les autres composants (ports, coordinateur) doivent être arrêtés.

Ensuite, l'usage de cette méthode a des conséquences en termes de performances et en particulier de mémoire. Il faut huit composants, et non plus quatre, pour fournir les services proposés par la membrane de la tâche. La multiplication du nombre d'instances peut influencer négativement les performances d'OSGi et d'iPOJO. Cet aspect est cependant à relativiser car la limite est élevée, l'expérience montre qu'iPOJO supporte sans difficulté la création de plus d'une dizaine de milliers d'instances.

Notre implémentation comme nous l'avons souligné se base sur Cilia. La solution choisie par cette plateforme est un mélange des deux méthodes. Les ports d'entrée sont des composants externalisés tandis que les autres modules appartiennent à la membrane (handlers). Nous préférons pour l'instant également une implémentation sous forme de handler. Nous ne disposions pas, en effet, du recul nécessaire sur l'influence de cette pratique sur les performances, et la charge de la gestion du cycle de vie est un risque important d'erreur.

Notre implémentation est donc conforme à la figure 67 (a). La première limitation de notre approche vient de l'impossibilité de remplacer les implémentations des constituants de la tâche. Pour effectuer une modification sur la structure de la tâche, il faut supprimer la tâche intégralement pour la redéployer. Leur configuration peut en revanche être modifiée durant l'exécution, ce qui offre déjà beaucoup de flexibilité. La conception des médiateurs Cilia est en cours d'évolution et supportera à terme le changement dynamique du scheduler. Nous aurons alors une idée plus précise de l'influence sur les performances et nous pourrons bénéficier de ces progrès. L'implémentation sera alors conforme à la figure 67 (b). Les implémentations actuelles des sous-modules peuvent être très facilement portées d'une version à l'autre : modification de quelques lignes.

2.3 DECLARATION D'UNE IMPLEMENTATION DE TACHE ATOMIQUE

La figure 68 représente une tâche telle qu'elle est implémentée avec les quatre handlers implémentant les sous-modules de la tâche et deux handlers supplémentaires qui se chargent de l'administration et du calcul de statistiques.

Les implémentations de tâches sont décrites dans un langage indépendant de la plateforme d'exécution. Dans Ceylan, la description s'effectue dans le langage XML. La figure 68 illustre les éléments principaux de la déclaration d'une tâche. Une implémentation fait référence à un type, porte un nom unique, et contient la description de chacun des constituants de la tâche. La

déclaration de l'implémentation d'une tâche atomique peut être extrêmement succincte et se réduire à la déclaration du processeur. Pour chaque constituant, le développeur fait référence au nom unique de son implémentation (attribut *ref*) et fournit sa configuration initiale. Cette configuration est spécifique à la fonctionnalité des sous-modules qui l'interprètent. Par convention, cette configuration doit être indépendante de la plateforme. Nous par la suite donnons un exemple de configuration pour chacun des modules.

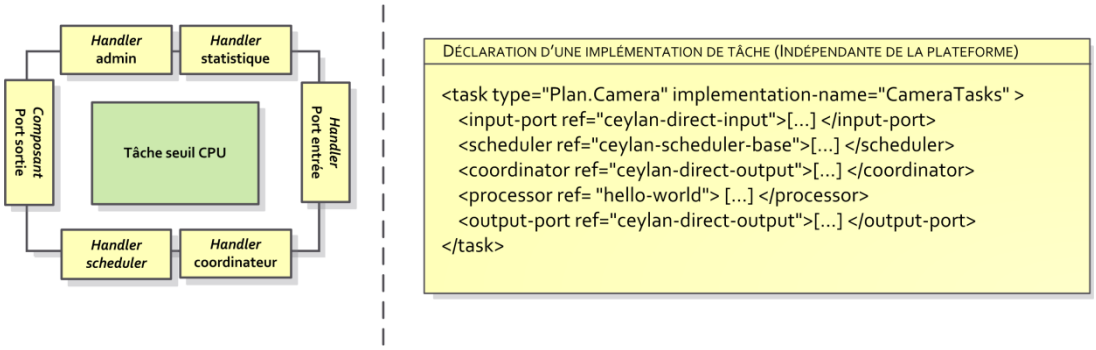

Figure 68 : Description d'une implémentation de tâche

Le modèle donné ici en exemple est le modèle indépendant de la plateforme. Les implémentations, qui sont désignées par l'attribut *ref*, sont elles décrites dans un modèle dépendant de la plateforme. L'implémentation de sous-modules est réalisée en iPOJO. Ce dernier divise l'implémentation en deux parties : le code source qui est l'implémentation du cœur, et un fichier XML qui décrit la membrane. Dans ce fichier XML, spécifique à la plateforme iPOJO, se trouve notamment la description des propriétés de configuration, des dépendances et des handlers utilisés (JMX, ...). Pour éviter la déclaration systématique des handlers optionnels (ports, coordinateur) et des handlers systématiquement présents (administration), nous avons défini une surcouche à ce langage. Pour autant cette déclaration reste spécifique à la plateforme. Nous donnerons un exemple de déclaration d'une implémentation pour le processeur.

2.4 IMPLEMENTATION DE PORTS

Nous avons réalisé deux implémentations des ports : avec communication indirecte via un bus et communication directe par service, en suivant les principes décrits dans le chapitre précédent.

Nos ports héritent de l'implémentation abstraite de port fournie par Cilia. Les ports Cilia sont des composants indépendants et non des handlers. Concrètement, pour le développeur de notre framework, il existe peu de différences entre l'implémentation d'un composant et celle d'un handler. Sur la figure 69 les ports sont représentés dans le conteneur par commodité.

Lorsque les tâches n'ont qu'un port voire aucun, comme la tâche de monitoring de CPU de la figure, aucune instance n'est créée. C'est le cas du premier composant de la figure.

La première version, **la communication indirecte**, implémentée utilise l'*Event Admin*. Ce dernier est un service standard proposé par la plateforme OSGi pour échanger des dictionnaires d'objets de façon événementielle. Il se base sur des *topics* qui peuvent être créés dynamiquement ; les clients s'abonnent à ces topics sur lesquels les producteurs publient. L'implémentation des ports d'entrée et de sortie est donc très simple (quelques lignes), ils publient et reçoivent chaque donnée sans se soucier du routage.

L'implémentation pour les tâches composites pose cependant une difficulté : il faut gérer la confidentialité des échanges entre les tâches du composite. Or il n'y a qu'une seule instance d'Event Admin par plateforme. Pour y parvenir, les topics doivent comporter une indication sur la provenance des informations, de sorte que les consommateurs puissent s'abonner aux tâches

appartenant au même composite. Les tâches qui appartiennent à un composite partagent toutes un identifiant unique de groupe qui préfixe les topics. Dans les hiérarchies plusieurs noms de données peuvent se succéder, par exemple : */guid/guid/TypeDonnée*.

En pratique cette implémentation est relativement efficace mais la présence d'une seule instance d'*Event Admin* par plateforme ralentit les communications. On constate cependant des ralentissements en cas d'échanges nombreux. D'autre part, l'*Event Admin* dispose d'un mécanisme de bannissement pointilleux qui peut conduire au bannissement de certaines tâches et donc à la perte de données.

Notre deuxième implémentation permet une **communication directe** entre les tâches. Chaque port de sortie fournit un service *DataProducer* (figure 69) dont les propriétés de service indiquent les données produites et l'identifiant de groupe associé au composite. Dans le cas illustré, la tâche *moniteur de CPU* produit le type MONITOR.CPU. Les ports de sortie se lient à ce service en fournissant leur identifiant unique à la primitive *bind* et publient un service *DataListener* qui décrit les données souhaités. Ce service est découvert par le port de sortie – en se reposant sur iPOJO – qui peut alors envoyer ses données. En cas de disparition des tâches abonnées, iPOJO notifie le producteur de la disparation du service – il n'y a donc pas d'action nécessaire de la tâche abonnée.

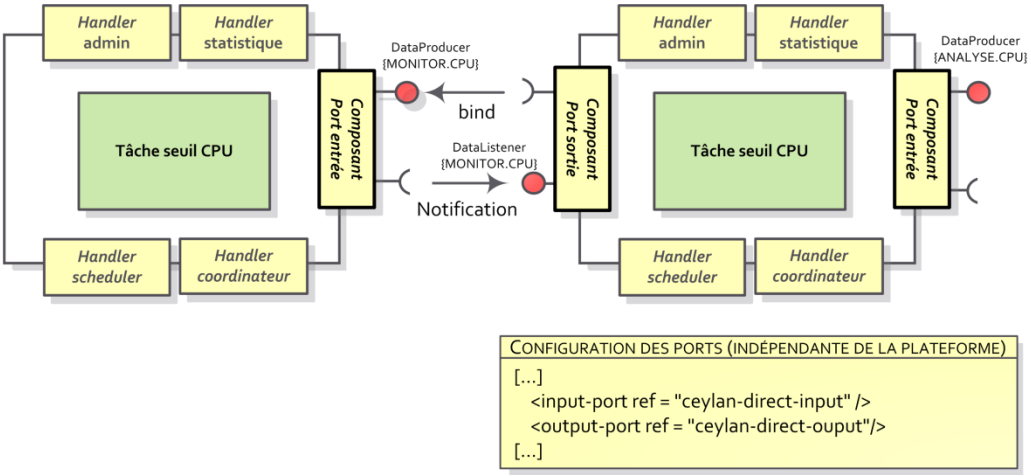

Figure 69 : Implémentation des ports avec communication directe

L'utilisation du service *DataProducer* pourrait être évitée, mais sa publication permet aux ports d'entrée de déterminer que toutes les données requises sont disponibles. Dans le cas contraire, le port d'entrée passe dans l'état invalide et par conséquent la tâche également.

La configuration de nos implémentations des ports (figure 69) est très succincte et ne comporte que la fabrique d'implémentations. En effet, la majorité des informations concernant les données entrantes et sortantes sont décrites dans le type (figure 66) et dans la déclaration du processeur. C'est notamment ce dernier qui fixe la liste des types optionnels comme nous le verrons plus loin. Une configuration plus poussée est possible en insérant du XML entre les balises. Le développeur aura alors à charge l'analyse de ce XML qui lui est remis sous forme d'une représentation arborescente propre à iPOJO qui ressemble à DOM. Au démarrage de la tâche, lors de la création des ports, cette configuration lui est transmise via la méthode « *configure* ». A tous moments la configuration peut être changée via une méthode « *reconfigure* ».

L'implémentation actuelle des ports d'entrée et de sortie garantit que les producteurs ne seront pas bloqués par un consommateur bloqué et que la collecte continue lorsque la tâche traite les données. Elles utilisent pour cela des pools de threads dont la taille évolue en fonction des besoins. Nous nous appuyons sur l'API *java.util.concurent* de java 5.

2.5 IMPLEMENTATION DE SCHEDULER

L'implémentation des schedulers est plus délicate car elle requiert la gestion de thread, du tampon et des conditions de déclenchement. Nous fournissons une classe abstraite qui prend en charge ces aspects et une implémentation d'un scheduler consacré aux types de données. Le développeur peut l'étendre pour implémenter son propre scheduler.

L'implémentation de notre classe abstraite se base sur le scheduler fourni par Cilia également sous la forme d'une classe abstraite. En supplément, elle gère le tampon des données, leur expiration, et assure que le processeur ne traite qu'une seule requête à la fois. L'implémentation du tampon est basée sur un simple *Set* java étendu pour rajouter le mécanisme d'expiration des données. Pour ce faire les données sont marquées.

A chaque réception d'une nouvelle donnée, le scheduler dépose les données dans le tampon puis utilise le thread unique – utilisation d'un pool d'un thread – pour lancer la condition de déclenchement. Si la tâche est en cours de traitement, les conditions de déclenchement ne sont pas analysées ce qui évite la génération de requêtes à partir de données qui pourraient autrement expirer avant la fin du traitement. La condition de déclenchement est appelée par la méthode abstraite *schedule* qui renvoie une requête formée d'une expression (*RequestExpression*), conformément à ce que nous avons décrit dans le chapitre précédent. Le processeur peut être mis à contribution lorsqu'il implémente l'interface *SchedulingMethod*. Il fournit alors une méthode *schedule* dont le rôle est de compléter ou modifier la requête (figure 70)

Sur la base de cette classe abstraite, nous avons réalisé une implémentation de scheduler qui permet d'exprimer des contraintes sur les données reçues et stockées dans la base (figure 70). La configuration de ce scheduler est constituée d'une expression et d'un tempo. L'expression est construite à partir d'opérateurs ET (^) et OU (,) et permet d'indiquer les types de données qui doivent se trouver présents dans le tampon pour activer la tâche. Il est possible également d'exprimer des conditions sur la valeur des propriétés dans la base. Ici, l'expression concerne la présence de données de type ALARM ou NONALARM auquelles s'ajoute une condition sur la résolution d'une caméra (propriété stockée dans la base). La balise tempo permet de configurer un délai minimal entre deux vérifications du scheduler pour éviter que la tâche ne s'active trop souvent – dans ce cas, l'attente est de 2 secondes.

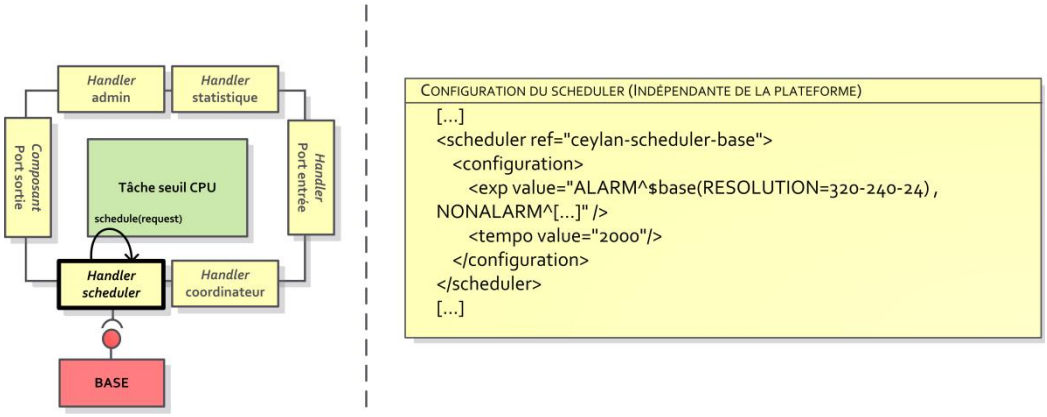

Figure 70 : Implémentation du scheduler

Nous fournissons également un scheduler de base qui ne comprend que le tempo. C'est cette version qui est utilisée par défaut avec un délai d'attente nul. Pour les moniteurs, nous fournissons un scheduler qui déclenche l'exécution de la tâche à intervalle de temps régulier configurable. La classe concrète n'est donc pas déclenchée uniquement par l'implémentation de la classe abstraite, elle peut se réveiller sur des conditions internes ou externes et réveiller la classe abstraite via une méthode *notify*. Il est possible de se baser directement sur ces implémentations

pour étendre les capacités du scheduler : ajouter un compteur du nombre de rapports, automatiser la vérification, entre autres.

La configuration est propre à chaque scheduler. iPOJO propose un format particulier de représentation des documents XML et appelle une méthode *configure* à l'initialisation des handlers. La configuration initiale se charge de vérifier que la configuration est correcte et d'attribuer des valeurs par défaut aux propriétés. Les reconfigurations sont possibles au moyen de la méthode *reconfigure*. Nous respectons le format utilisé par iPOJO pour cette méthode également. L'analyse du XML est sensiblement la même dans les deux méthodes mais certains attributs peuvent ne pas être modifiables après la configuration initiale, c'est un choix du développeur.

2.6 IMPLEMENTATION DU COORDINATEUR

L'implémentation du framework est en constante évolution. Les premières versions se basaient sur du filtrage tel que nous l'avons décrit dans le chapitre précédent. La version actuelle se base sur un arbitrage avec des coordinateurs passifs (figure 71). L'implémentation du coordinateur est relativement aisée. Nous ne fournissons pas de classe abstraite mais une simple interface dans ce cas, car la réalisation de coordinateurs pour la négociation est trop différente.

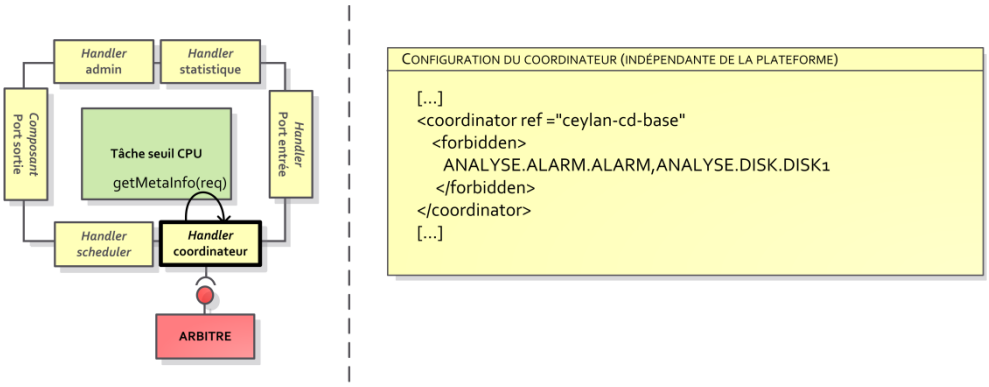

Figure 71 : Implémentation du coordinateur

Après réception de la requête du scheduler, le coordinateur consulte la liste des types qui nécessitent une autorisation et créée une nouvelle requête. Cette dernière contient des informations supplémentaires dont le nom de la classe et les méta-informations que le processeur désire incorporer. Pour ajouter ces dernières le processeur doit implémenter l'interface *CoordinatorMethod* et la méthode *getRequestMetaInfo* qui les calcule à partir de la requête. La requête peut ainsi comprendre une estimation du temps de traitement ou des ressources nécessaires. La requête construite, le scheduler l'envoie à l'arbitre qui le bloque le temps de l'élection.

La configuration du coordinateur (figure 71) se base sur les mêmes principes utilisés par le scheduler par l'appel de deux méthodes : *configure* et *reconfigure*. Dans notre cas, seule la liste des types nécessitant une autorisation est passée.

Les tests de négociation réalisés se sont basés sur une version différente de l'implémentation sans de notion claire de coordinateur. Pour cette raison, nous ne fournissons pas encore de coordinateur basé sur un protocole de négociation.

2.7 IMPLEMENTATION DU PROCESSEUR ET CONFIGURATION DE LA TACHE

Le processeur est le cœur du composant iPOJO qui constitue la tâche. Il réalise comme nous l'avons expliqué la partie spécifique du traitement.

Pour implémenter un nouveau processeur, il suffit d'étendre l'interface *Processor* qui est très semblable à celle de Cilia. Elle comporte deux méthodes principales : *process* et *configure.* La méthode *process* prend en entrée les données à traiter et renvoie les données générées. Les moniteurs peuvent envoyer des données au moyen d'un objet particulier injecté qui possède une méthode *push*. Les handlers d'administration et de statistiques jouent un rôle particulier. Ils sont associés à l'implémentation du processeur de façon transparente.

Le handler de statistiques (figure 72 a) se charge de l'appel de la méthode process. Il connaît ainsi les données d'entrée et de sortie et le temps d'exécution de la tâche. Ce handler détecte également les blocages de la tâche. Pour ce faire il appelle la méthode *process* via un thread dédié en utilisant un *ExecutorService* de Java. Il accorde un délai maximal à la tâche pour s'exécuter, en cas d'échec Java détruit le thread et renvoie une erreur. La tâche est alors considérée comme bloquée et le handler d'administration en est informé.

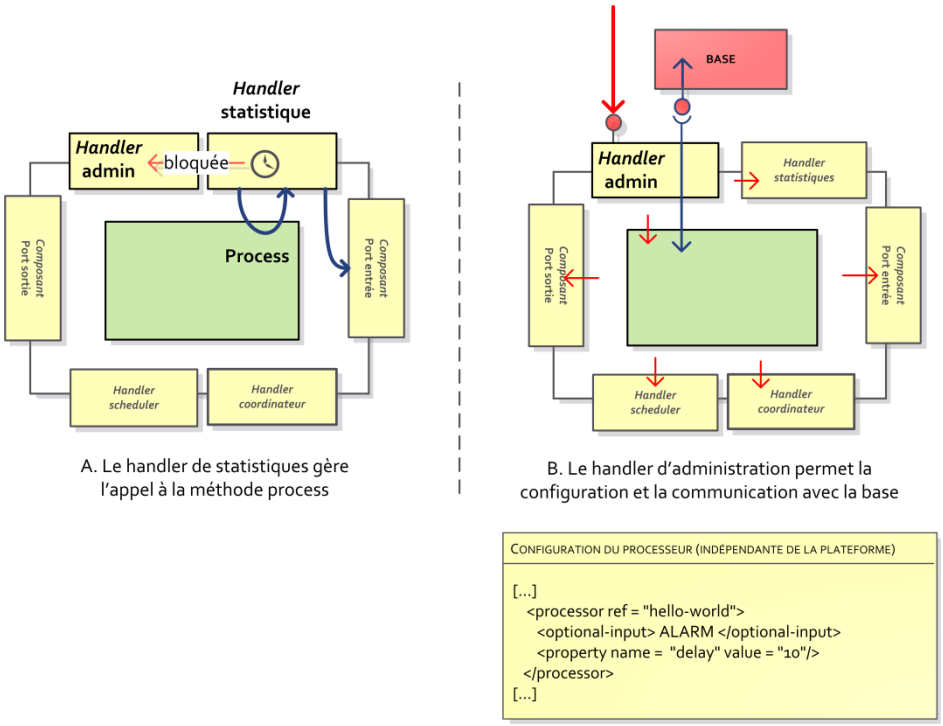

Figure 72 : Rôle des handlers administration et de statistiques

Le handler d'administration propose un service qui permet la configuration du processeur et des handlers. Elle s'effectue par l'appel des méthodes *configure/reconfigure* des handlers – la représentation XML de la configuration est alors passée en paramètre – et par l'appel de la méthode *configure* du processeur qui accepte un dictionnaire de propriétés. A l'initialisation de la tâche, les valeurs des propriétés des handlers sont décrites comme nous l'avons présenté dans les extraits de XML fournis. L'extrait de la figure 72 montre la configuration des données optionnelles d'entrée et sortie ainsi que la configuration des propriétés du processeur. La description de ces propriétés est conforme à la celle fournie par le type de la tâche.

Le handler d'administration est aussi responsable de la communication entre la base de données et le processeur. Nous utilisons pour cela la machine à injection d'iPOJO. Lors de la lecture dans le processeur d'une valeur, la récupération de cette dernière dans la base par le handler qui l'injecte est transparente. De même, lorsque la valeur est modifiée, elle est propagée dans la base. Ainsi le développeur n'a pas à se soucier de la communication avec la base. Les champs injectés sont indiqués lors de l'implémentation du processeur dans le fichier de

configuration de la membrane qui est spécifique à la plateforme. Par exemple, dans le cas de la figure 73, la propriété définie dans le type *nbSeuil* est associée à l'attribut *sharedCountThreshold*.

Enfin le handler d'administration est responsable du stockage de l'état de la tâche. Il envoie des notifications de modifications, et les statistiques lorsque la tâche est configurée pour cela, aux composants implémentant l'interface *TaskListener*. Pour éviter l'envoi de trop nombreuses notifications, les services *TaskListener* exposent une propriété de service qui indique le nom des tâches auxquelles ils souhaitent s'abonner.

```
public class CheckCPU implements Processor{

public List<CeylanData> process(List datas) {
 [...]
 if (((Double) datas.getData(monitorCPU).get(keyCPUValue)) > threshold) {
 result.add(new CeylanData(outTypeCPUth, null));

 sharedCountTheshold++;
 [...]

return result;
}

public void configure(String propertyName, Object newPropertyValue) {
 if(propertyName.equals(confTH){
    threshold = newPropertyValue;
    [...]
}

}
```

IMPLÉMENTATION DU PROCESSEUR (DÉPENDANT DE LA PLATEFORME)

```
<processor classname="fr.lig.CheckCPU" implementation-name= "CheckCPU"
type ="AnalyseurCPU"  />

<input>
 <data type="MONITOR.CPU" field="inTypeMonitorCPU" >
  <key name="CPU" field= "keyCPUValue"/>
 </data>
</input>
<output>
 <data type="ANALYSE.CPU" field="outTypeCPUTh" />
</output>

<base>
 <property name="nbSeuil" field="sharedCountTheshold"/>
</base>

<configuration>
 <property name= "Threshold" field="confTH"/>
</configuration>

</processor>
```

Figure 73 : Exemple d'implémentation d'un processeur

La description de l'implémentation du processeur (XML figure 73) contient une référence au code source (*classname*) et possède un nom unique (*implementation-name*). Une difficulté de l'utilisation du XML est qu'il faut gérer les multiples redondances de références au nom des données. Nous ne voulions pas que cette référence apparaisse plus d'une fois par implémentation (dans la source et dans le XML). C'est pourquoi nous proposons d'injecter le nom des données dans des attributs à l'initialisation de la tâche. Dès lors, Le développeur n'est plus obligé de déclarer le nom des données sous forme de variable statique dans son code. En cas de changement du nom des types de données, les modifications se limitent au XML.

3 IMPLEMENTATION DES COMPOSITES

Abordons maintenant l'implémentation de l'architecture de contrôle et des tâches composites. Comme nous l'avons expliqué dans le chapitre précédent, les composites regroupent un ensemble de tâches dont la coopération est guidée par les mécanismes de contrôles.

iPOJO introduit la notion de composite mais de façon extrêmement rigide, si bien qu'il est difficile d'ajouter de nouveaux éléments à un composite durant l'exécution. Pour cette raison, nous avons choisi d'implémenter les composites sous forme d'un ensemble de services partageant un même identifiant de groupe. L'encapsulation est moins forte, en particulier il n'est pas possible d'empêcher un service extérieur de modifier les tâches, mais le dynamisme est très simple à gérer. En outre, il est également plus facile de remonter le modèle de l'exécution. Cependant c'est une implémentation imparfaite car du point de vue de la plateforme l'instance d'un composite sera considérée comme plusieurs instances de tâches.

La figure 74 donne un exemple de description utilisée pour la déclaration d'une tâche composite. Comme les implémentations de tâches atomiques, l'implémentation possède un nom unique. Chaque tâche peut être déclarée soit avec son type, soit avec son implémentation et il est possible de donner une reconfiguration du scheduler et des propriétés. Les propriétés de configurations de la tâche sont mappées par le composant d'administration sur une ou plusieurs tâches (attribut configuration).

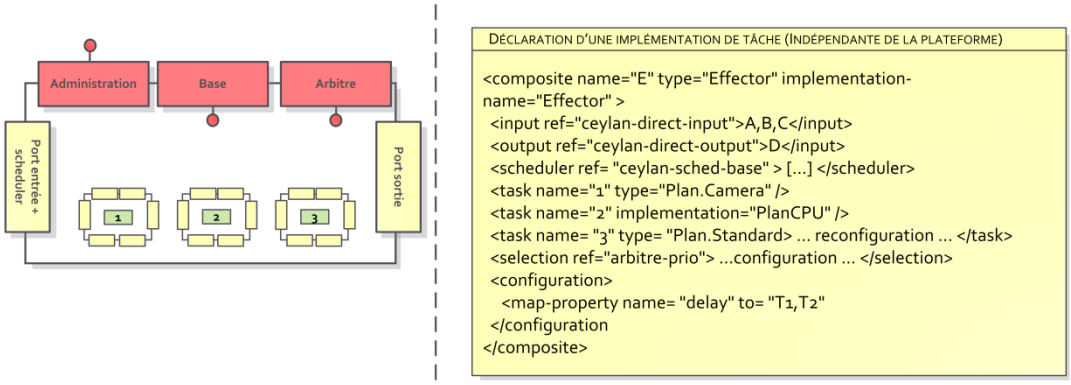

Figure 74 : Déclaration d'une implémentation de tâche composite

3.1 LA COMMUNICATION, L'ADMINISTRATION ET LA BASE DE DONNEES

L'implémentation de la communication, de la base et de l'administration ne pose pas de difficulté car elle repose entièrement sur la plateforme OSGi et iPOJO.

Il n'est pas nécessaire d'implémenter une infrastructure de communication car elle est fournie par la plateforme OSGi (Event Admin ou communication directe). L'implémentation des ports d'entrée et de sortie de la tâche est très similaire à celle décrite auparavant. Ces ports sont implémentés sous forme de médiateurs. Le port d'entrée possède un scheduler. Nous nous reposons sur les implémentations décrites plus haut.

La base d'un composite ne possède pas de configuration puisque l'ensemble des propriétés partagées sont décrites dans le type. Elle propose un service *Base* qui comprend deux primitives *get* et *set* permettant de modifier les propriétés. Elle requiert le service de la base de son parent – comme le fait l'arbitre sur la figure 75 – pour partager les propriétés définies dans le type. Notre implémentation est très simple puisqu'elle repose sur un dictionnaire de propriété stocké en mémoire vive.

L'implémentation du module d'administration n'est pas complément fidèle à l'architecture que nous avons présentée. Il nous sert essentiellement d'indirection pour la configuration des

propriétés mais ne se charge pas de la gestion du cycle de vie des tâches ou de la remontée du modèle d'exécution. C'est le module de construction de la couche d'adaptation qui s'en occupe. La principale raison vient de l'utilisation des services qui rend ce type d'implémentation plus facile et plus rapide comme nous le verrons plus loin.

3.2 MECANISMES DE SELECTION

L'implémentation de l'arbitrage est plus complexe car elle requiert la gestion de la synchronisation. Comme nous l'avons expliqué, nous nous sommes focalisé sur l'implémentation de mécanismes d'arbitrage. Nous avons implémenté deux types d'arbitre. Le premier se base sur les types de données et des priorités sur les tâches, le deuxième se focalise sur une somme de jetons disponibles et regagnés avec le temps.

L'arbitre propose un service *Selection* aux coordinateurs. Il n'est composé que d'une procédure *putRequest* qui prend en argument la requête construite par un coordinateur et renvoie le nouvel état et la liste des rapports autorisés. Cette méthode bloque les coordinateurs tant que l'élection ne s'est pas terminée. L'arbitre requiert le service *Selection* de son parent et lui envoie à son tour les requêtes. L'arbitre demande uniquement l'autorisation pour les données d'un type de données interdites qui proviennent de l'extérieur. Pour que l'arbitre puisse déterminer la provenance d'une donnée, le port d'entrée lui envoie l'entête de chaque donnée extérieure.

Pour simplifier le développement, l'arbitre est composé de deux parties (figure 75) : un handler générique et l'algorithme. Le handler générique prend en charge les aspects récurrents : les communications avec les coordinateurs, la synchronisation et la communication avec le parent. L'algorithme est l'implémentation du mécanisme de sélection. Il implémente les méthodes de l'interface *SelectionMechanism* : *election*, *getStates* et *configure*.

L'enchainement des appels est pris en charge par le handler. Après réception des requêtes, la méthode *election* est la première appelée. Elle calcule la liste des requêtes élues à partir de l'état des tâches, de la configuration de l'algorithme et des informations contextuelles qu'elle a récolté. Après quoi le handler vérifie les autorisations et convient si nécessaire avec le père de la liste des requêtes définitives. A partir de ces dernières, l'algorithme de sélection calcule la liste des nouveaux états. Enfin le handler affecte les nouveaux états aux tâches et distribue les autorisations en débloquant les tâches en attente. Cet enchainement évite la demande d'autorisation au père et permet le calcul des états à partir des requêtes qui seront réellement exécutées.

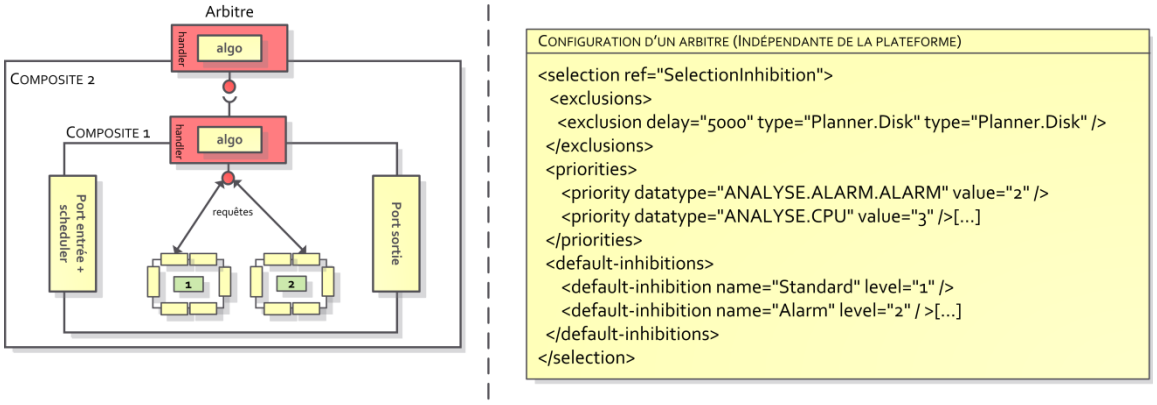

Figure 75 : Configuration d'un arbitre.

Les algorithmes et leur configuration sont très variés. La figure 75 donne un exemple de la configuration d'un arbitre basé sur la priorité des tâches et des données. Nous avons implémenté plusieurs algorithmes basés sur les priorités. Le plus simple se contente d'attribuer les requêtes à

la tâche la plus prioritaire. Les autres requêtes sont validées (éventuellement partiellement) si elles ne contiennent pas une donnée en traitement. Pour tous les algorithmes, l'exécution d'une requête peut être associée à une inhibition durant laquelle les tâches indiquées ne pourront plus s'exécuter. Dans le cas décrit par la figure 75, les tâches seront bloquées pendant cinq secondes. Le second algorithme donne une priorité aux données et aux inhibitions de sorte qu'une tâche inhibée puisse être réveillée par une donnée de forte priorité. La configuration est plus complexe et verbeuse mais l'algorithme permet d'éviter qu'une tâche inhibe d'autres tâches pour un problème mineur.

Notre implémentation d'arbitrage par jetons est plus simple. Chaque tâche se voit attribuer un nombre de jetons initial et une priorité. L'algorithme trie les tâches par priorités puis par nombre de jetons restants. Une exécution coute un jeton et les tâches regagnent un jeton à intervalle régulier configurable pour chaque tâche. Nous montrerons un exemple d'utilisation de ces algorithmes dans le chapitre suivant.

Pour l'instant la configuration de l'arbitrage est centralisée. Cela facilite l'implémentation des coordinateurs et la compréhension du gestionnaire mais peut compliquer l'ajout de nouvelles tâches. Pour y remédier, nous envisageons de distribuer certaines informations propres à la tâche, telles que sa priorité ou son nombre de jetons.

Nous n'avons pas implémenté d'arbitre sensible au contexte mais le framework le permet. L'arbitre peut être implémenté sous la forme d'un médiateur Cilia classique ce qui lui permet de collecter des données en provenance de l'extérieur. L'utilisation de l'implémentation des ports que nous avons présentée est ainsi rendu possible. Cette réutilisation donne au concepteur de l'arbitre la possibilité de se consacrer sur l'algorithme proprement dit.

4 CONSTRUCTION ET ADMINISTRATION DU GESTIONNAIRE

Une partie importante du framework est la construction du gestionnaire et l'ensemble des outils qui permettent d'adapter son comportement.

4.1 CONSTRUCTION DU GESTIONNAIRE

L'implémentation que nous présentons ici n'est pas exactement conforme à ce que nous avons exposé dans le chapitre précédent. Pour simplifier la mise en œuvre du framework et des composites, nous avons centralisé la construction et la remontée de l'architecture des gestionnaires. La remontée de l'architecture n'est donc pas distribuée dans chacun des contrôleurs.

Pour faciliter la modification de la représentation des gestionnaires, la construction du gestionnaire repose sur l'utilisation de trois services (figure 76) :

- **un service responsable de la représentation du gestionnaire à l'exécution** qui s'appuie sur les services d'administration de chacune des tâches et s'abonne aux événements d'administration ;
- **un service responsable de la représentation abstraite du gestionnaire souhaité** qui stocke une représentation arborescente de la structure du gestionnaire souhaité. Il vérifie que les types existent auprès du service de gestion des types ;
- **un service responsable du maintien de la cohérence entre les deux modèles** qui permet leur modification des deux modèles et essaie de faire correspondre le modèle de l'exécution avec le modèle abstrait.

Ces trois services gèrent l'architecture des gestionnaires présents sur la plateforme. Le service de cohérence propose les interfaces d'administration pour modifier les gestionnaires. Il répercute les modifications sur les deux autres services, de sorte qu'il n'y a qu'une interface d'administration ; cela simplifie, du point de vue de l'implémentation, le maintien de la cohérence.

Comme pour la gestion des types (figure 66), un parseur XML indépendant vient se greffer sur les interfaces de gestion et l'administration est rendue possible via la console ou via une interface graphique que nous présentons après.

La récupération du modèle à l'exécution est facilitée par l'utilisation des services. Comme nous n'utilisons pas de composite iPOJO, la récupération et la construction du modèle de l'exécution peuvent être effectuées très facilement par le constructeur. Il suffit de s'abonner aux services d'administration des tâches qui sont tous disponibles puisque non masqués par l'encapsulation d'iPOJO. Cette méthode très simple permet de conserver un modèle de l'exécution à jour.

Pour créer une nouvelle tâche, nous reposons sur les fabriques de service créées par iPOJO pour chaque implémentation de tâches atomiques et de handlers. Une factory (fabrique) est un service particulier qui permet la création d'instances de composants à partir d'une implémentation. Dans notre cas, ces factories exposent des meta-informations qui fournissent des indications sur les propriétés d'une implémentation, ce qui permet de choisir la factory. Le choix d'une factory est soit aléatoire, soit basé sur un service *ImplementationSelection* dont la référence est fournie par le développeur dans l'ADL. Ces services n'implémentent qu'une méthode *select* qui prend en paramètre les noms des factories à comparer et renvoie la factory choisie. Pour ajouter une nouvelle tâche à un composite, il suffit de lui attribuer l'identifiant de groupe correspondant. Les liaisons entre la tâche et le composite se font simplement par découverte de services.

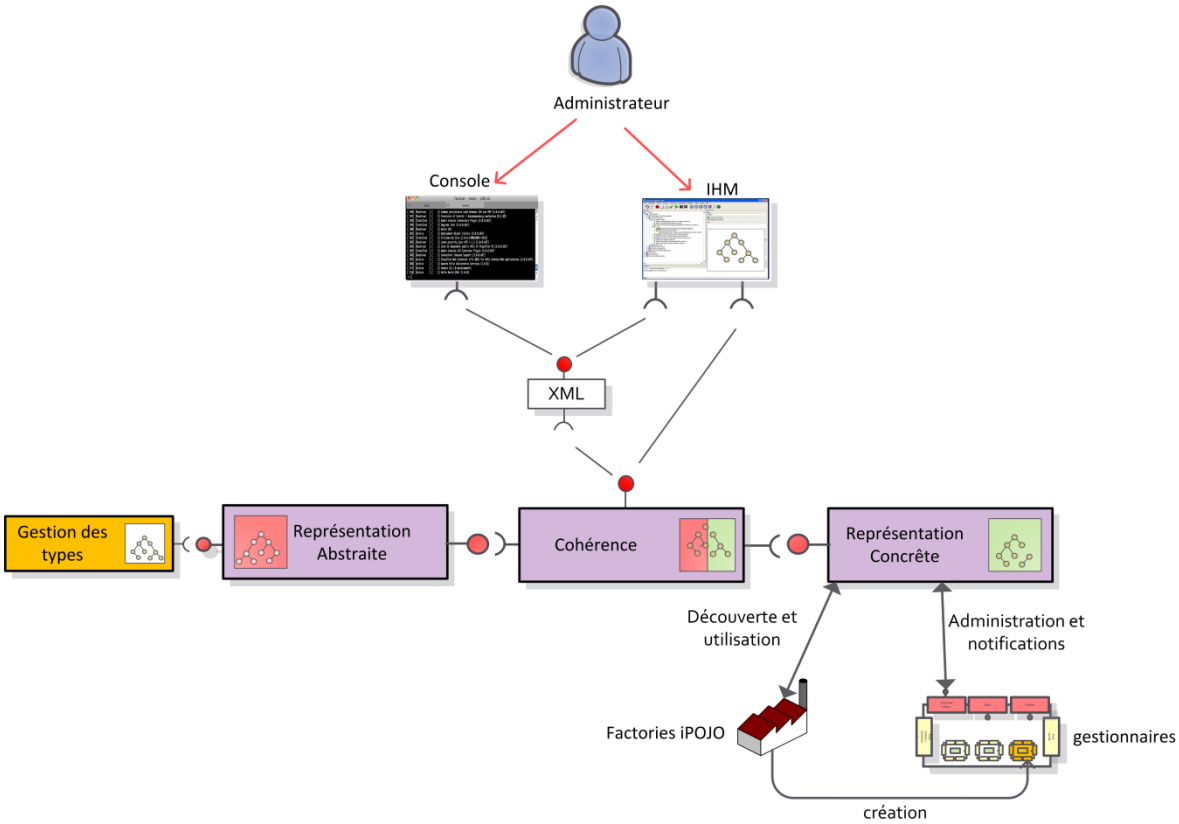

Figure 76 : Construction du gestionnaire

Notre implémentation de la construction du gestionnaire et de la remontée de l'architecture en cours d'exécution est donc pleinement fonctionnelle même si elle centralisée. Tant que le nombre de tâches est réduit, cette centralisation est un avantage car le maintien de la cohérence entre les deux modèles est facilité. Nous envisageons cependant de répartir la récupération du modèle à l'exécution et l'administration des tâches pour se conformer à l'architecture que nous avons présentée, et garantir le passage à l'échelle. Du point de vue de l'expert, il n'y aura pas de modification entre la version centralisée et la version repartie de l'adminstration des gestionnaires.

4.2 ADMINISTRATION DU GESTIONNAIRE

L'administration du gestionnaire joue un rôle très important car elle fournit les moyens de modifier dynamiquement le comportement du gestionnaire. Nous avons construit une interface graphique qui autorise l'autorisation et la modification du comportement des tâches ainsi qu'un interpréteur de script qui permet de construire des scénarios.

L'interface graphique est un outil qui permet de créer et modifier la configuration des gestionnaires pendant l'exécution. C'est un client lourd développé en SWT, la bibliothèque graphique utilisée par Eclipse. Ce choix nous permettra, à terme, de proposer un plug-in pour cette plateforme de développement très populaire. D'autre part SWT est une librairie très performante. Nous souhaitions posséder une représentation honnête de l'architecture des gestionnaires en cours, ce qui nous à conduit à favoriser les clients lourds.

L'interface graphique est indépendante de la plateforme OSGi et communique avec les gestionnaires via JMX qui s'est imposé comme un standard d'administration sur les plateformes JAVA. Ce choix permet de l'exécution de l'interface sur une machine différente, ce qui évite de

faire porter la charge de l'interface aux systèmes exécutant les gestionnaires et facilite l'administration à distance. D'autre part, les méthodes et notifications fournies par JMX peuvent être utilisées par des applications tierces pour administrer la plateforme.

Notre interface (figure 77) s'inspire fortement d'Eclipse, en offrant un système d'onglets et de barre de navigation qui permet de changer rapidement d'onglets. La partie supérieure de l'interface propose trois onglets : configuration des gestionnaires, configuration des types de données et configuration des types de tâches qui permettent de visualiser et modifier ces différents aspects du gestionnaire dynamiquement.

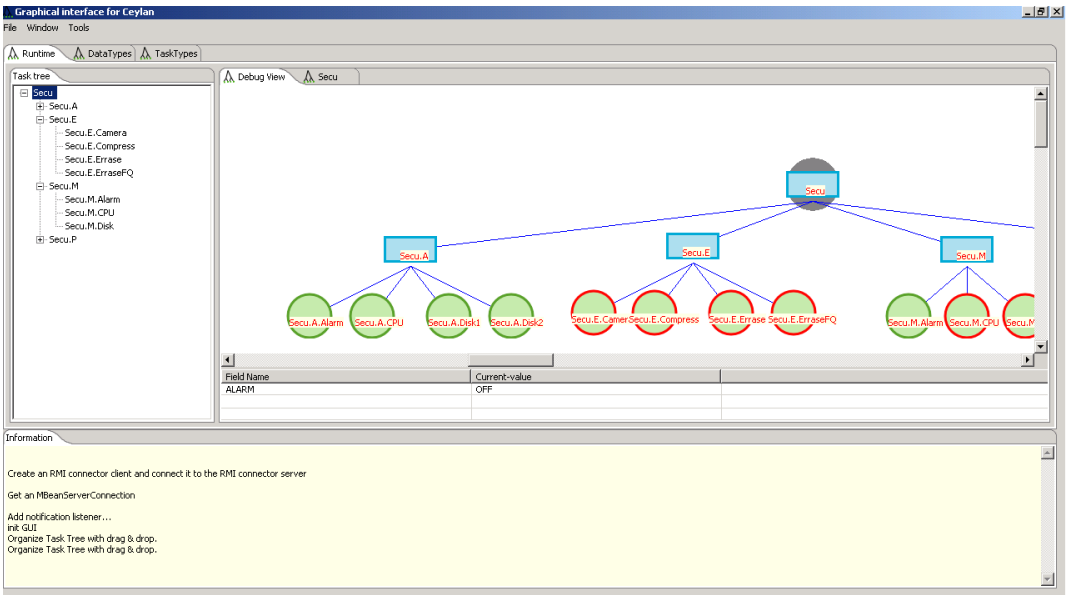

Figure 77 : Capture d'écran de l'interface de gestion des gestionnaires de CEYLAN.

Ces trois catégories sont bâties sur le même principe. A gauche, une barre de navigation qui représente de façon arborescente le modèle en cours, à droite la fenêtre de configuration qui permet l'observation des paramètres de configuration et leur modification si possible (figure 79). L'édition des types de données et de types de tâches offre les mêmes possibilités de création que le XML et ne permet que la suppression, puisque par convention les types ne sont pas modifiables Lors de la création de nouveaux éléments, par exemple un type de tâche (figure 78), l'utilisateur est assisté par l'interface qui fait des propositions parmi les types de données existants.

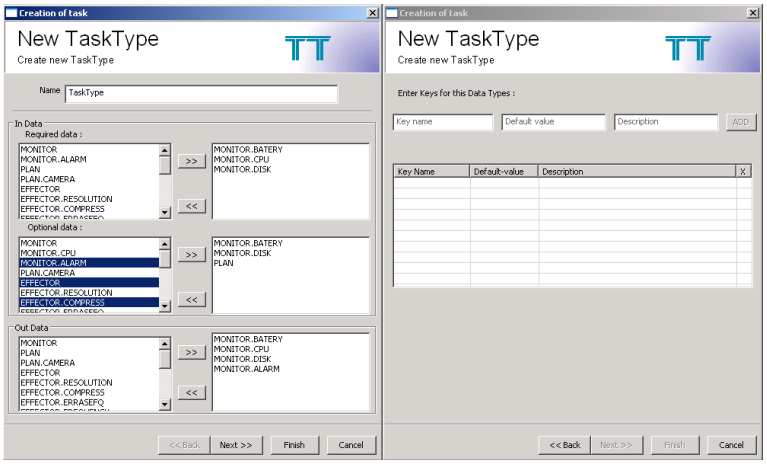

Figure 78 : Création d'un type de tâche.

Pour les tâches, l'interface donne les mêmes possibilités de création que XML mais permet également de modifier dynamiquement la configuration des schedulers, coordinateurs et propriétés de la tâche. Une partie de la configuration doit s'effectuer en XML pour offrir le maximum de flexibilité aux développeurs des algorithmes et schedulers. L'interface permet également de récupérer les statistiques à intervalle de temps régulier. Les panneaux de configuration des tâches composites et atomiques sont différents. La configuration des tâches composites comprend les paramètres de l'algorithme de sélection.

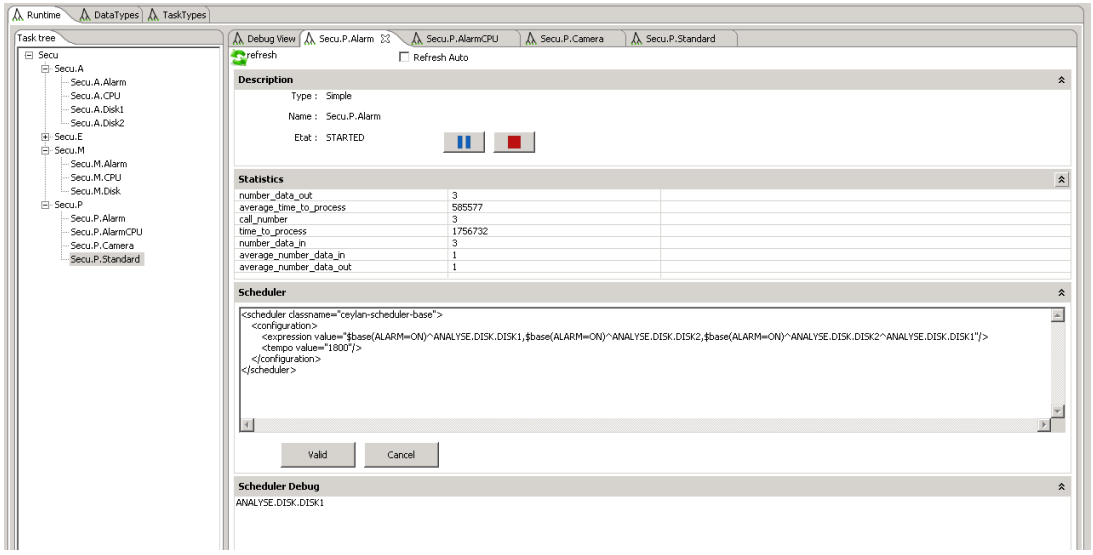

Figure 79 : Configuration d'une tâche.

L'interface offre une vue synthétique de l'état des tâches et de leur organisation, à droite de la figure 77. Les tâches composites sont représentées par un carré et les tâches atomiques par un rond. La couleur de ces éléments change en fonction des états : vert lorsque la tâche est active, rouge lorsque l'instance n'existe pas, orange lorsqu'elle est bloquée et entouré de rouge lorsqu'elle est invalide pour cause de donnée manquante. Cette représentation visuelle permet d'identifier rapidement les problèmes. Sous l'arbre, le panneau permet la configuration rapide de la tâche sélectionnée.

La création des tâches peut se faire directement à partir de cette vue en sélectionnant le composite d'insertion. L'interface propose alors un panneau similaire à la figure 78. Il est possible de choisir parmi les fabriques de service (i.e les fabriques de tâche) ou les types de tâches existant. Le choix d'un type de tâche entraine immédiatement la sélection de la factory associée si bien que la plupart des informations suivantes sont remplies par défaut. La tâche de l'administrateur est donc considérablement réduite en comparaison de l'écriture des formats XML.

Comme nous l'avons expliqué précédemment, l'interface graphique s'appuie également sur le parseur XML, ce qui lui permet de charger des gestionnaires complets très facilement. Il est également possible de sauvegarder une configuration donnée dans un fichier, ce qui fait de l'interface graphique un outil d'édition. C'est une fonctionnalité importante compte tenu de la verbosité du XML qui rend parfois l'écriture de gestionnaire fastidieuse.

En plus de l'interface graphique, nous avons proposé un module de script qui se repose sur l'interface d'administration du module de construction. Les possibilités offertes par les scripts sont pour l'instant réduites à la création, la modification et la suppression des tâches. Il n'est pas encore possible d'exprimer des conditions et de vérifier l'existence de tâches. Il est très simple d'implémenter un module de scénarios au-dessus de ces scripts en construisant un fichier de script par scénarios. Nous n'avons pas encore intégré ces scripts à l'interface. Pour l'instant, cette

possibilité a peu été exploitée, nous nous sommes contenté d'un module qui charge les scripts en fonction de l'heure de la journée (cron). Cette modification automatique pourrait facilement s'étendre à d'autres informations contextuelles.

5 SYNTHESE

Dans ce chapitre nous avons présenté l'implémentation CEYLAN de notre framework. Avec près de 16000 lignes de code l'implémentation est conséquente. Elle met l'accent sur le dynamisme et la réutilisabilité en proposant un modèle à composants orienté service qui permet de faciliter la conception des gestionnaires. Notre prototype réalise l'ensemble des fonctionnalités qui ont été décrites dans le chapitre précédent.

Le cœur de CEYLAN et l'environnement d'exécution s'appuient sur Cilia et iPOJO ; il représente 10000 lignes de code. Le choix de ces technologies et l'orientation service facilitent la gestion du dynamisme et l'implémentation des tâches d'administration. En particulier les implémentations des tâches peuvent être découvertes et utilisées dynamiquement durant l'exécution. C'est iPOJO qui fournit les solutions nécessaires à la gestion de ce dynamisme avec les concepts de factory et de types de composants.

Nous avons respecté l'architecture que nous avons décrite. Chaque module de la tâche est implémenté sous la forme d'un composant ou d'un handler iPOJO. Cela donne une grande flexibilité pour l'assemblage des tâches et en particulier pour l'agencement des schedulers, coordinateurs et ports. Toutefois comme nous l'avons signalé, l'architecture interne d'une tâche atomique ne peut pas être modifiée une fois l'instance créée. Cette limitation devrait rapidement disparaitre dans les implémentations futures.

Les tâches simples et atomiques sont décrites dans un ADL en XML. Il en va de même pour les types de tâches et de données. L'architecture du gestionnaire souhaitée peut être modifiée à tout moment grâce aux interfaces fournies par le module de construction.

A chaque instant, une représentation honnête de l'architecture en exécution est disponible. Nous fournissons un module de statistique qui calcule des informations sur les tâches. Ici, nous ne respectons pas totalement l'architecture que nous avons proposée puisque le modèle et l'administration sont centralisés. Nous sommes capables de détecter les tâches bloquées lorsque leur exécution dépasse un temps fixé par l'expert. Cet ensemble d'informations servira de base à l'administration.

La couche d'administration comprend une interface graphique qui permet d'observer et de modifier le comportement des tâches ainsi qu'un interpréteur de script qui permet de construire des scénarios. L'interface graphique est un client lourd, représentant 6000 lignes de code, qui permet l'administration à distance de la plateforme. Il fournit une représentation honnête du système en exécution et permet la modification de l'ADL et la découverte de tâche. Cette vue synthétique du gestionnaire facilite son administration.

Chapitre 7 - **Validation**

Dans ce chapitre nous donnons un exemple concret d'utilisation de notre approche pour la construction d'un gestionnaire autonomique. Nous avons choisi d'autonomiser une application classique de la domotique : une application de surveillance vidéo. Ce type d'application est généralement bien connu et permet de mettre en œuvre différentes stratégies de gestion.

Dans un premier temps nous présentons l'application, puis nous montrons la construction d'un gestionnaire autonomique étape par étape en expliquant comment les différentes préoccupations de gestion ont été développées indépendamment.

1 APPLICATION DE SURVEILLANCE DE DOMICILE

L'application que nous avons considérée est une application de surveillance de domicile et plus particulièrement la partie responsable de la surveillance des intrusions. Elle est composée d'un ensemble de services permettant la surveillance à distance d'un appartement et de signaler les événements suspects. L'application déclenche une alarme au domicile et informe le service de sécurité auquel les utilisateurs ont souscrit. Pour cela, un ensemble de caméras sont disposées dans les pièces et les images produites et stockées par un enregistreur vidéo.

L'application (figure 80) s'exécute sur une passerelle domestique à services OSGi. Nous disposons d'un ensemble de caméras placées dans les pièces de la maison et connectées à leur pilote, qui s'exécute sur la passerelle. Certaines caméras peuvent être mobiles avec une batterie. Les pilotes capturent régulièrement des images et les envoient de façon asynchrone sur une file de messages en utilisant un bus à événement (Event Admin). Les images sont exploitées par deux composants : un composant de stockage et un détecteur de mouvement. Le composant de stockage réalise une sauvegarde de chaque image dans une base de données ; ces images peuvent êtres visionnées pour identifier les intrus. Le détecteur est un service qui analyse et détecte les mouvements entre chaque image. Lors de la détection d'un changement significatif, ce service envoie un événement qui décrit les caractéristiques (pourcentage de mouvements enregistrés, zone de l'image, pièce). Le service Alarme souscrit à ces événements et prend les décisions nécessaires en fonction de la gravité du message et en particulier de la répétition des signaux. En cas d'intrusion, en fonction de la configuration du mécanisme de communication, un message est envoyé à l'administrateur de la plateforme par mail ou une alarme est déclenchée dans le domicile.

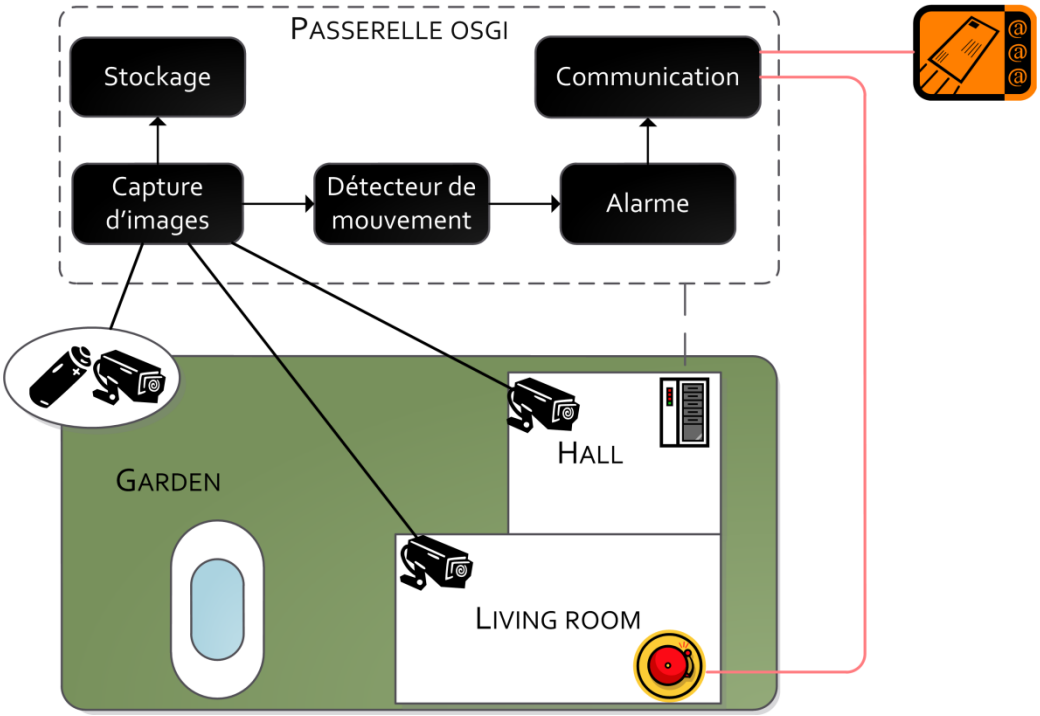

Figure 80 : Application de détection d'intrusion

Chaque composant de l'application est donc un service. Ces services sont réalisés et peuvent être configurés de la façon suivante :

- **le service de capture d'image donne l'accès aux différentes caméras vidéo.** Chaque caméra est identifiée par un nom unique, ce qui permet d'identifier les

pièces et les photos. Ce service peut être configuré pour envoyer des images à différentes fréquences, dans des résolutions différentes, et allumer ou mettre en veille les caméras. Chaque image est diffusée sous forme d'événement sur un canal souscrit par les consommateurs.

- **le service de stockage souscrit à ces images et les sauvegarde dans une base de données de taille limitée.** Il fournit des primitives d'observation et de modification.
- **le service de détection de mouvement reçoit régulièrement les images et les analyse**. Nous utilisons un algorithme naïf de détection de mouvement qui compare chaque image à la précédente et en déduit un pourcentage de pixels modifié. Nous divisons chaque image en portions de taille égale ce qui permet de déterminer les zones de l'image modifiée. La configuration de la sensibilité des caméras s'effectue globalement et non au cas par cas pour chaque caméra. Lorsqu'un mouvement significatif est perçu le service diffuse un message indiquant la pièce concernée, la zone de l'image et la quantité de mouvement perçue.
- **le service d'alarme s'occupe de centraliser les diverses alarmes en provenance des différents capteurs placés dans le domicile et de déterminer l'importance de ces alarmes.** En effet, la détection des intrusions ne constitue qu'une activité particulière pour l'application globale qui supervise le fonctionnement général de la maison. Cette application est notamment responsable de la consommation d'électricité et d'eau, ou encore de la température du réfrigérateur. Le service choisit entre : une communication quotidienne ou hebdomadaire lorsque les alarmes sont d'une sévérité mineure, et une communication immédiate en cas d'intrusion.
- **il existe différents services de communication qui permettent de prévenir l'utilisateur.** Un premier service déclenche une alarme sonore via les hauts parleurs de l'ordinateur, un autre utilise un appareil communicant (de type nabaztag ou iBuddy) et provoque un changement de couleur et enfin le dernier envoie des mails pour informer l'utilisateur lorsqu'il est à distance.

Notre objectif avec cette application est la conception d'un gestionnaire prenant en charge des aspects différents. C'est pourquoi, pour compliquer la gestion, nous avons fait le choix, aujourd'hui de moins en moins réaliste, de limiter fortement l'espace de stockage à disposition des caméras. D'autre part, l'administrateur est prévenu par mail et nous supposons donc que son temps de réaction peut être long. Notre objectif est de conserver le plus grand nombre d'images permettant l'identification des suspects. Notre gestionnaire devra prendre en charge cet espace de stockage, la batterie et les alarmes.

Dans notre cas, les caméras sont de piètre qualité : un téléphone Android et plusieurs webcams. Le téléphone joue ici le rôle de caméra mobile dont la batterie s'épuise et doit être gérée. Nous possédons deux implémentations de détecteur de mouvement. L'une d'elles est précise mais consomme beaucoup de processeur, l'autre est moins précise mais également moins gourmande. Par défaut, l'application utilise l'implémentation la plus précise.

Nous partons du principe que cette application ne peut pas être modifiée directement. La gestion de l'application s'effectue par l'action sur des sondes et effecteurs placés sur l'application. Ces touchpoints proposent des services qui seront utilisés par le gestionnaire. Ils donnent notamment la capacité d'intercepter les événements échangés entre les services et de modifier leur implémentation.

2 GESTION DE L'APPLICATION

Nous objectif est de construire un gestionnaire prenant en charge la configuration et la gestion de cette application. Ce gestionnaire est centralisé et prend en charge l'ensemble des services qui composent l'application.

L'application n'est pas la seule fonctionnant sur la plateforme et il est nécessaire de ménager la consommation des ressources processeur et de stockage. Nous souhaitons construire ce gestionnaire par étapes en intégrant progressivement de nouvelles préoccupations : gestion des caméras, gestion de l'espace disque, gestion du processeur, gestion de la batterie. Chacune de ces préoccupations est implémentée en isolation avec la réutilisation éventuelle de tâches puis fusionnée au sein d'un même gestionnaire.

Dans ce qui suit, nous détaillons chaque scénario et décrivons les tâches développées, les stratégies de sélection, et les résultats obtenus sur l'application. Nous présentons ces derniers sous forme de courbes représentant l'activité du processeur, l'espace disque occupé et lorsque cela est nécessaire les batteries ou la mémoire. Dans de nombreux exemples, pour éviter le chevauchement des deux courbes, nous avons utilisé le service de détection de mouvement économe pour générer la figure.

La figure 81 nous servira de référence pour la suite. Elle illustre le comportement de l'application sans gestionnaire autonomique. L'axe des abscisses représente le temps et celui des ordonnées la consommation mémoire et processeur en pourcentage. Rapidement, l'espace disque sature et ne permet plus le fonctionnement de l'application, le détecteur précis utilisé consomme la moitié du processeur ; la fréquence et la résolution des images ne sont pas modifiées. La présence d'une alarme n'a pas d'influence sur ces aspects. L'application fonctionne donc, mais connaît des limites et a un impact négatif sur les ressources de la machine.

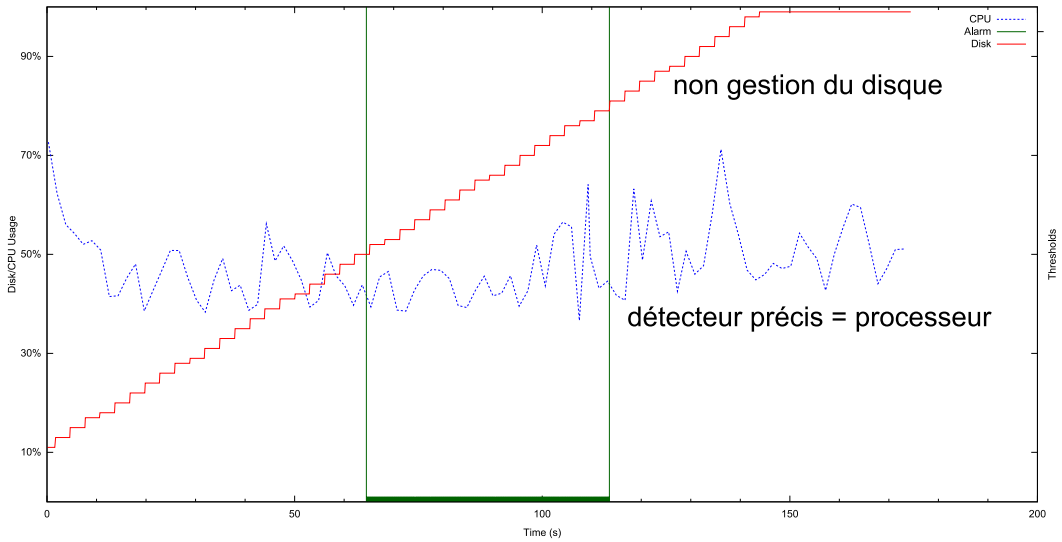

Figure 81 : Comportement de l'application sans gestion autonomique

Pour permettre les tests dans des conditions reproductibles et faciliter la comparaison des courbes, nous simulons les caméras. Nous utilisons trois caméras simulées ; elles diffusent un même lot d'images à intervalle de temps réglable de sorte que les quantités de mouvement enregistrées et les zones réapparaissent régulièrement. Les alarmes résultent soit d'un déclenchement manuel, soit d'un de ces jeux de tests. D'autre part, l'application est exécutée sur une plateforme OSGi ne comportant pas d'autre application. Nous simulons les pics de consommations processeur par l'usage d'un programme tiers qui entraine une consommation

intensive des ressources de la machine. Enfin, le temps est accéléré et les échelles de temps sont fournies essentiellement comme point de référence pour la description des courbes.

Soulignons qu'ici la qualité de la gestion importe peu, le développement d'un gestionnaire autonomique ad-hoc et la conception d'algorithme constituent en effet en eux même un travail de recherche. Nous souhaitons seulement montrer comment la construction d'un gestionnaire est rendue flexible par l'utilisation de notre approche.

2.1 UNE PREMIERE GESTION DE LA MEMOIRE ET DU PROCESSEUR

Nous apportons une première solution simple pour gérer la mémoire et économiser du processeur. Nous souhaitons créer un gestionnaire qui supprime les images les plus anciennes lorsqu'un seuil est dépassé, et qui limite la consommation du processeur lorsqu'il n'y a pas d'alarmes en utilisant la version moins précise du détecteur de mouvement.

Pour la construction du gestionnaire, nous respectons la convention d'IBM et organisons les tâches en quatre grands composites M, A, P et E. Les trois derniers s'abonnent respectivement aux messages de types Monitor, Analyse et Plan. Dans certains cas, ce découpage pourra apparaître excessif mais nous nous y tenons pour donner un point de repère bien connu. La mise en œuvre de ce gestionnaire a réclamé la création des tâches suivantes :

- **une tâche d'observation des alarmes** qui souscrit aux messages envoyés par l'alarme et retransmet le message.
- **une tâche d'observation du processeur** qui utilise une librairie Java pour récupérer la consommation du processeur sur la machine. Cette tâche n'est pas la même selon la plateforme utilisée.
- **une tâche d'analyse alarme** qui doit déterminer si l'alarme est significative. Ici, elle se contente d'utiliser la base de données en stockant un booléen qui indique la présence d'une alarme ou non.
- **une tâche de détection de seuil** qui est instanciée deux fois. Une première fois pour détecter le dépassement de mémoire (disk1 à 80%) et une seconde fois pour le dépassement du processeur (CPU à 95%). Cette tâche envoie un message up en cas de dépassement vers le haut et down en cas de dépassement vers le bas ; elle utilise une moyenne calculée sur une quantité configurable de valeurs précédentes.
- **une tâche de planification qui décide de l'implémentation du détecteur de mouvement à utiliser.** Elle se base sur la présence d'alarme et la présence de seuil. En cas d'alarme, elle active le détecteur performant si le processeur le permet. Si le seuil de processeur est dépassé et lorsqu'il n'y a pas d'alarme, elle utilise l'implémentation économe.
- **une tâche responsable de l'effacement des images du disque**. Elle supprime la quantité d'images demandée en utilisant les primitives de la base.
- **une tâche responsable de l'activation/désactivation des détecteurs**. Ici, nous utilisons deux variantes : suppression/création des instances ou activation/désactivation. Pour la seconde variante, le changement est plus rapide mais consomme plus de ressources.

En plus de ces tâches nécessaires à la gestion de l'application, nous ajoutons des tâches permettant l'observation du comportement de l'application et le déclenchement des tâches d'exécution du gestionnaire. Ces tâches proposent une représentation graphique de l'utilisation des processeurs et CPU en Swing (figure 82) et stockent ces données dans des fichiers csv, ce qui nous permet de générer les courbes à intervalle de temps régulier avec gnuplot.

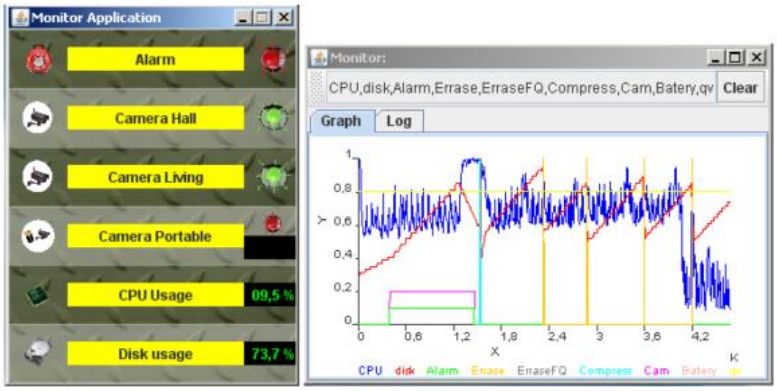

Figure 82 : Visualisation des courbes en direct produite par les tâches de monitoring.

La figure 83 illustre l'utilisation de ces tâches. La courbe en trait discontinu représente la consommation du processeur. Au démarrage de l'application, le détecteur économe est utilisé. Sa faible consommation, 20% en moyenne, ne perturbe ainsi pas le fonctionnement des autres applications ; elle résulte de l'analyse d'une quantité plus faible d'images en comparaison avec le détecteur précis. Lorsque la première alarme est déclenchée vers 64s (signalée par un trait vertical), la tâche de planification des détecteurs est déclenchée (losange sur la courbe) ; elle décide de l'utilisation du détecteur précis. La consommation du processeur augmente rapidement pour atteindre 50% en moyenne. L'application utilise ce gestionnaire jusqu'à la fin de l'alarme qui active de nouveau la tâche consacrée au détecteur, qui prend alors la décision de basculer sur la version économe. On retrouve la même situation entre 148 et 204s avec l'activation et la désactivation d'une alarme. Cependant pendant l'alarme, nous simulons une consommation intensive de processeur jusqu'au dépassement du seuil de 95%, ce qui conduit pendant ce temps à l'utilisation de la version économe. Ce choix explique la brève chute à une consommation de 10% du processeur lorsque la consommation redevient normale. Rapidement, la puissance de calcul est de nouveau disponible et la version précise s'active.

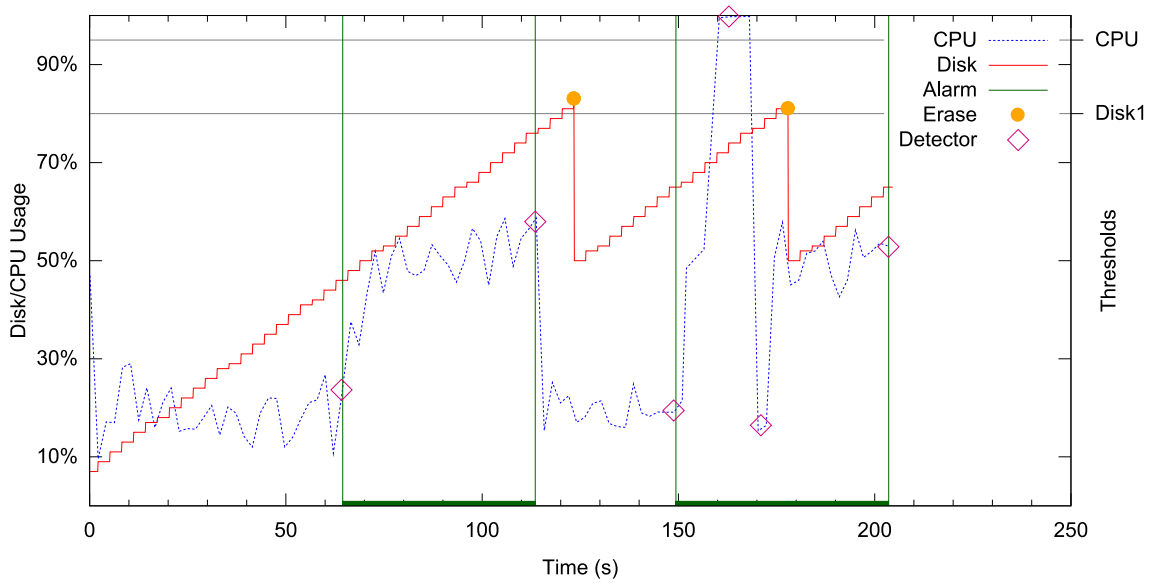

Figure 83 : Suppressions des images anciennes et changement de détecteur.

La courbe en trait continu représente la consommation de l'espace disque alloué à l'application. L'augmentation est très régulière car l'on ne modifie pas la fréquence ni la résolution des images. Dès que la quantité d'images stockées dépasse le seuil disque fixé (Disk1), la tâche responsable de la gestion du disque demande l'effacement d'une quantité fixée (30%) d'anciennes images. L'effet est ici immédiat car l'opération est peu complexe à réaliser.

Ce premier scénario montre une gestion très simple de l'application sans conflit entre tâches. La communication est donc directe et les couches supérieures peu sollicitées. Dans ce cas, la gestion du disque reste indépendante de la présence d'alarme ; des images intéressantes peuvent être effacées. Plus loin, nous prendrons en compte cette information. Les deux préoccupations, gestion du processeur et gestion du disque, ont pu être développées et testées séparément dans des composants indépendants.

2.2 GESTION DU DYNAMISME ET ADAPTATION

Avant de complexifier le gestionnaire, nous configurons le gestionnaire pour prendre en compte des événements internes tels que le blocage d'une tâche ou la présence d'une tâche peu performante.

Nous avons conçu un service qui observe le modèle d'exécution fourni par le système en s'appuyant sur les modules de construction. Lorsqu'une tâche bloquée est détectée, le module recherche une implémentation alternative et, si possible, remplace la tâche.

La figure 84 illustre le remplacement d'une tâche bloquée.

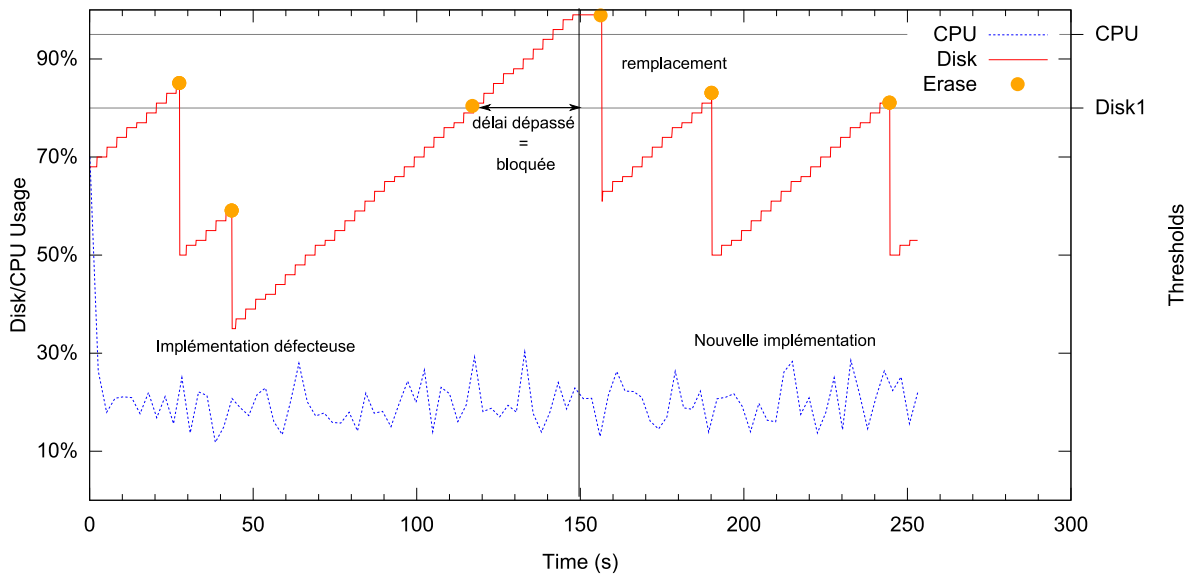

Figure 84 : Remplacement d'une tâche jugée bloquée

La tâche de planification permettant l'effacement est déclarée par son type. Par défaut, l'implémentation choisie est la première disponible. Nous créons une seconde implémentation de cette tâche qui fonctionne de façon aléatoire et qui finit par dépasser le temps maximum autorisé par l'expert pour l'exécution. Dans notre système, ce délai est fixé empiriquement. Pour simuler des erreurs, la tâche se déclenche aléatoirement et son temps de traitement croît au fil du temps. Les premières exécutions se déroulent dans les délais fixés, ainsi la tâche est considérée comme en bonne santé. Ici, nous ne cherchons pas à corriger le comportement incohérent qui apparaît vers 42s (déclenchement en dessous du seuil). En revanche, lorsque le temps d'une exécution dépasse 30s comme c'est le cas vers 118s, le module de statistique considère la tâche comme

bloquée. Ce blocage conduit au remplacement de la tâche par le module de surveillance que nous avons ajouté au gestionnaire. En cherchant parmi les fabriques de tâches disponibles, il trouve l'implémentation correcte qui est alors utilisée après 160s. Après remplacement, le gestionnaire retrouve un comportement adéquat.

Le module de statistique de chaque tâche calcule également le temps moyen entre chaque exécution. Notre module de surveillance se sert de ces statistiques pour supprimer les tâches qui s'exécutent trop fréquemment. Pour la gestion du disque dur, une exécution trop fréquente est en effet synonyme d'une mauvaise gestion du problème. A intervalle de temps régulier, le module de surveillance consulte le module d'exécution et les statistiques des tâches. Lorsqu'il juge la moyenne entre deux exécutions trop faible, il cherche une nouvelle implémentation.

La figure 85 illustre le remplacement d'une tâche jugée inefficace. Après le démarrage du gestionnaire, l'administrateur désinstalle l'implémentation de la tâche d'exécution en cours vers 140s. Automatiquement, le gestionnaire trouve une nouvelle implémentation du même type : ici l'implémentation que nous introduisons est une tâche qui effectue la compression des données au lieu de les effacer. L'administrateur réinstalle ensuite l'implémentation mais le gestionnaire continue d'utiliser l'implémentation en cours car rien ne justifie encore son remplacement.

La tâche de compression est utilisée une première fois vers 180s (le déclenchement est symbolisé par un carré). La compression est efficace lors des premières exécutions car le nombre d'images à compresser est important. A cet instant, on remarque la présence d'un pic de processeur qui marque l'utilisation de l'algorithme. La réponse est également légèrement plus lente que la tâche d'effacement simple : la courbe décroit moins fortement. Par la suite et à partir de 210s, l'efficacité de la compression diminue fortement : les images à compresser sont de moins en moins nombreuses. L'algorithme s'exécute une dizaine de fois consécutivement en ne compressant qu'une image, ce qui ne provoque qu'un léger fléchissement de la courbe.

Finalement vers 290s, le module de surveillance remplace la tâche pour revenir à l'ancienne implémentation qui est de nouveau disponible. Le gestionnaire peut alors reprendre une exécution normale car la tâche qui efface les données est efficace.

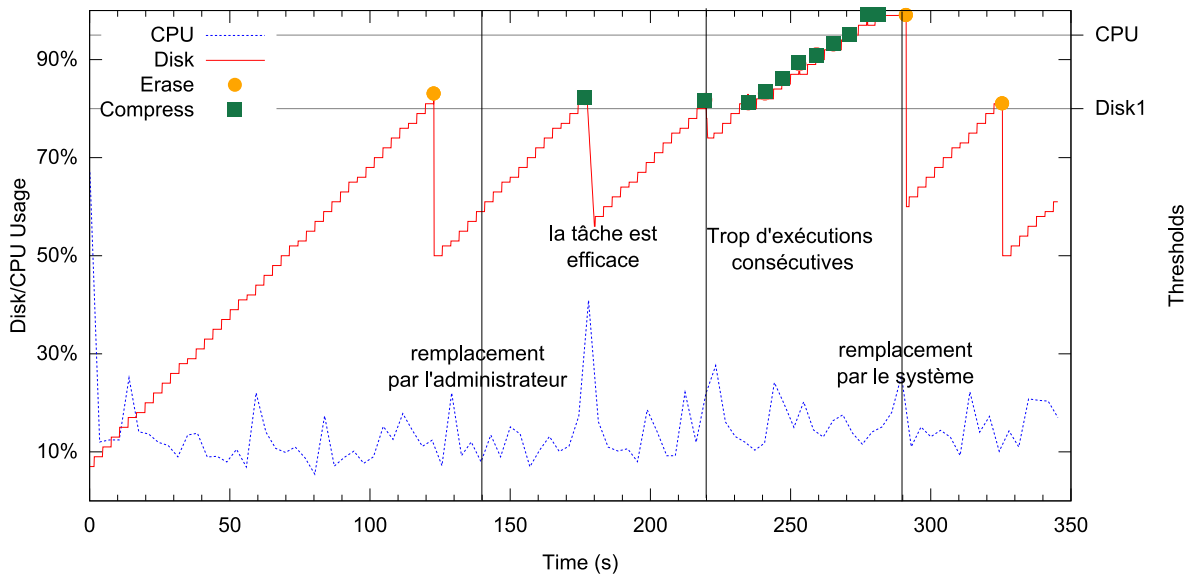

Figure 85 : Remplacement d'une tâche jugée inefficace.

Il n'est donc pas nécessaire de connaître l'implémentation et le comportement précis des tâches pour modifier l'action du gestionnaire. Il suffit de connaître leur rôle. Le choix d'un critère

d'évaluation de l'efficacité d'une stratégie permet de basculer facilement et dynamiquement de l'une à l'autre. Dans ce cas, le nombre moyen d'exécutions sur une période fixée est un bon critère car il est rapide à calculer et relativement fiable. L'adaptation automatique d'un gestionnaire autonomique n'est donc pas nécessairement complexe. Il est même souhaitable qu'elle reste simple, sans quoi le comportement du gestionnaire peut rapidement devenir imprévisible.

2.3 GESTION DES CAMERAS

Nous nous proposons dans ce scénario d'agir sur les caméras pour gérer la fréquence et la résolution des images qui sont capturées pour réduire la quantité d'images stockées lorsqu'il n'y a pas d'alarme. Nous gérons également la caméra pourvue d'une batterie.

Une stratégie complémentaire de gestion du disque consiste à prendre en compte la topologie de la maison pour étendre ou limiter la capture des caméras des pièces dans lesquelles une intrusion est peu probable. Par exemple, dans un immeuble les caméras situées aux étages peuvent rester en veille ou dans un mode restreint tant qu'une intrusion n'a pas été détectée dans les étages inférieurs. C'est le risque que nous prenons ici en modifiant la configuration des caméras en fonction de la présence d'alarmes. En outre, nous gérons la batterie des caméras

En plus des tâches de supervisions que nous avons décrites plus haut que nous réutilisons, nous utilisons les tâches suivantes :

- **une tâche de supervision de la batterie des caméras** qui envoie à intervalle de temps régulier un état de la batterie des caméras concernées.
- **une tâche de planification qui vérifie la présence d'une alarme** et détermine quelles caméras doivent être allumées ou éteintes. Cette tâche possède une propriété partagée de configuration qui contient la liste des caméras dites secondaires, c'est-à-dire les caméras dont l'activité peut être réduite lorsqu'il n'y a pas d'alarme. Lorsqu'aucune caméra principale n'est disponible, elle active toutes les caméras secondaires.
- **une tâche responsable de l'activation ou de la mise en veille des caméras** qui fonctionnent sur batterie. Lorsqu'une batterie passe sous un certain seuil la caméra est mise en veille tant qu'il n'y a pas d'alarme, à l'exception des caméras jugées principales. Elle utilise la base de données pour déterminer les caméras secondaires et principales.
- **une tâche qui permet l'activation ou la désactivation des caméras** en agissant sur les touchpoints de l'application.

La figure 86 illustre l'utilisation de ces tâches. Dans notre cas, nous utilisons deux caméras principales : l'une sur batterie et l'autre sur secteur, et deux caméras secondaires. La version simulée de la caméra sur batterie perd très rapidement de l'énergie. Au démarrage, les deux caméras principales fonctionnent, mais lorsque vers 20s la batterie passe sous la barre des 30%, la caméra est éteinte ; la croissance est moins forte et la batterie est économisée. Lorsqu'une alarme est déclenchée vers 55s, toutes les caméras y compris celle sur batterie sont allumées. De fait la batterie est de nouveau consommée. La croissance de la courbe du disque est beaucoup plus forte du fait de l'activation de toutes les caméras. Vers 65s, nous décidons de charger la caméra. A la fin de l'alarme, les caméras secondaires sont éteintes, celle sur batterie est de nouveau mise en veille car elle se trouve sous le seuil des 30%. Cependant, rapidement, les 30% sont atteints et la caméra est de nouveau activée. A la fin, les deux caméras principales sont donc activées.

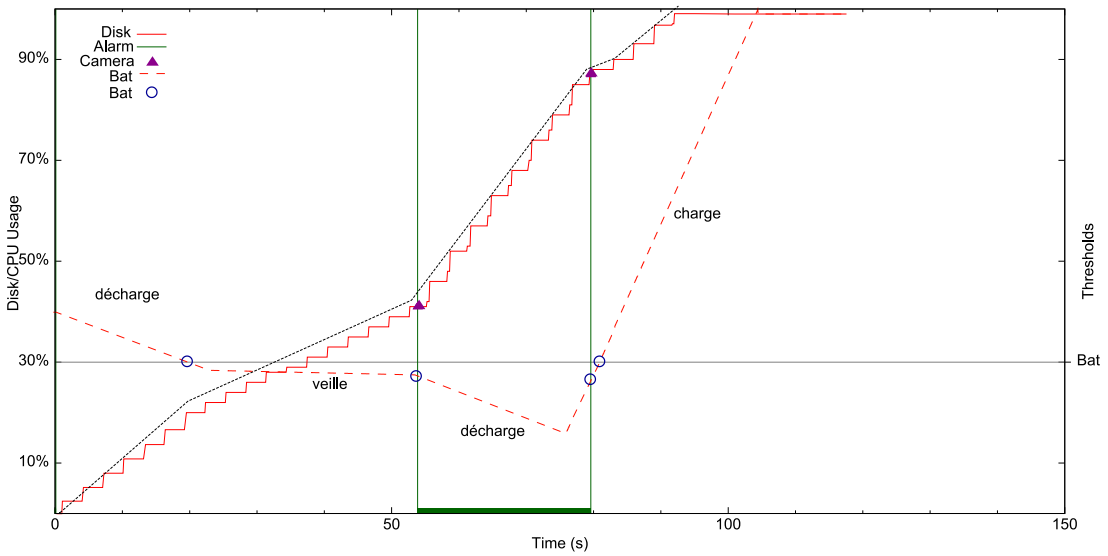

Figure 86 : gestion des caméras.

Dans ce scénario, nous avons pu profiter des tâches déjà développées auparavant (détection de seuil et alarme) et réaliser et tester la gestion des caméras sans nous soucier du développement des autres tâches ni de la gestion du disque. Par la suite, il sera très facile d'intégrer ces tâches au sein du même gestionnaire.

2.4 GESTION DES ALARMES

Nous souhaitons assister l'administrateur dans le tri des alarmes. L'application actuelle déclenche une alarme lorsqu'une certaine quantité de mouvement a été décelée par le détecteur de mouvement. Certaines alarmes peuvent être déclenchées par un mauvais réglage de ce détecteur ou parce que l'utilisateur a oublié de les désactiver. Par exemple, le détecteur ne permet pas de configurer une sensibilité différente pour chaque pièce, ce qui est un problème avec les caméras extérieures qui sont le plus susceptible de détecter une alarme à cause d'un animal.

Nous concevons des tâches pour prendre en compte les informations fournies par le détecteur de mouvement, les zones de la caméra et la présence de l'utilisateur pour choisir d'activer ou de désactiver certains services de communication.

Une première version du gestionnaire est composée de tâches avec une configuration donnée par l'administrateur. Elle réutilise les tâches de monitoring et se compose des tâches suivantes :

- **une tâche d'observation du détecteur qui enregistre pour chaque image la quantité de mouvement calculée et les zones concernées**. Elle enregistre ces informations dans la base de données sous forme d'un dictionnaire dont la clef est le numéro de l'image.
- **une tâche d'observation du réseau** qui communique régulièrement la liste des machines connectées sur le réseau.
- **une tâche d'analyse de détection de présence de l'utilisateur**. Notre implémentation envoie un ping régulier sur l'adresse IP attribuée au smartphone de l'utilisateur (IP fixe attribuée par adresse MAC). Lorsque l'adresse est repérée la tâche considère que l'utilisateur est présent et stocke l'information dans la base.

Le cœur est composée de tâches d'analyse qui, à partir des informations collectées, déterminent si une alarme est présente. Pour chaque alarme, ces tâches indiquent si une alarme est présente ou non. Dans notre cas :

- **une tâche d'analyse de quantité de mouvement.** Pour chaque pièce elle possède une configuration de la quantité de mouvement minimale avant de déclencher une alarme.
- **une tâche d'analyse de zone**. Pour chaque pièce, elle possède une liste des zones qui ne doivent pas déclencher d'alarme – par exemple des zones du jardin telle que la piscine.
- **une tâche de présence** qui décide si l'alarme doit être déclenchée en fonction de la présence de l'utilisateur et du moment de la journée (jour/nuit).

Ces tâches sont indépendantes et il est possible d'en ajouter pour prendre en compte de nouveaux aspects : le type de pièce en fonction du moment de la journée ou la météo par exemple. Pour compléter le gestionnaire, il faut :

- **une tâche de synthèse** qui, sur la base de l'ensemble des rapports d'alarmes reçus, prend la décision de déclencher une alarme ou non. Dans notre version, il faut qu'une seule tâche décide d'une alarme.
- **un ensemble de tâches d'exécution** qui permettent d'agir sur les mécanismes de communication pour empêcher la prise en compte de certaines alarmes : débrancher l'alarme sonore ou empêcher l'envoi de mail entres autres.

Cette première version du gestionnaire fonctionne parfaitement et évite de déclencher des alarmes inutiles. Comme la configuration est précise et provient de l'administrateur, il n'y a pas d'erreurs. En construisant un échantillon simulé de 100 alarmes dont 50 inutiles, les 50 sont détectées. Dans ce cas on voit l'intérêt de la tâche de synthèse qui, sans connaître la nature précise des tâches qui analysent les alarmes ni connaître les informations sur lesquelles elles fondent leurs décisions, parvient à déterminer la présence d'alarme ou non.

Une seconde version, plus complexe, du gestionnaire doit permettre d'éviter l'étape de configuration des différentes tâches. Chaque tâche utilisée en coopération possède un nom unique auquel l'utilisateur peut se référer. Une liste des alarmes est conservée avec la liste des images concernées par la tâche de gestion des alarmes. Lorsqu'une alarme se déclenche, l'utilisateur peut vérifier la liste des alarmes et indique aux gestionnaires via une interface proposée par une tâche de monitoring si l'alarme est correcte ou incorrecte. En cas de faux négatif, il l'indique au gestionnaire ; toutes les tâches d'analyse ajustent leur configuration. En cas de faux positif, il indique en plus les tâches qui sont vraisemblablement responsables, par exemple : false_neg ZONE, MOUV indique que la zone est mauvaise et que la sensibilité est trop élevée. Les alarmes concernées par le faux positif sont notifiées et ajustent leur configuration.

Les deux tâches concernées (sensibilité et zone) possèdent un algorithme d'ajustement qui consiste pour les zones comme pour la sensibilité (classée en six niveaux) à construire une évaluation qui donne une probabilité à chaque choix : par ajout ou retrait de points en cas d'erreur positive ou négative. La sensibilité la plus probable est choisie et les zones dont la probabilité dépasse 75% sont choisies. Ces algorithmes très simples donnent de très mauvais résultats avec 19% de faux négatifs et 23% de faux positifs. La quantité de faux négatif est inquiétante. Il serait souhaitable de faire appel à un expert des applications de vidéosurveillance pour implémenter les algorithmes d'auto-configuration et remplacer nos algorithmes naïfs. Il pourrait alors développer et tester ces tâches séparément et nous la livrer pour que nous la comparions aux nôtres.

Dans ce cas notre intention est de souligner que les tâches peuvent ajuster automatiquement leur configuration dans le temps car leur granularité le permet. Pour cet exemple encore, le développement des tâches a pu se faire en indépendance des autres

préoccupations. L'utilisation d'une tâche de synthèse est une solution intéressante pour faire coopérer des tâches. Il n'y a toujours pas conflit mais coopération, la communication est directe sans intermédiaire ni demande d'autorisation et le gestionnaire est donc réactif.

2.5 ALGORITHME DE COMPRESSION

Nous souhaitons maintenant pouvoir utiliser notre tâche de compression sans qu'elle soit systématiquement désinstallée par le système. Nous l'avons vu, cette tâche n'est efficace que les premières fois ; après quoi, le nombre d'images à compresser n'est plus suffisant pour rendre cette stratégie intéressante. Nous proposons d'utiliser ces deux tâches concurrentes en alternance en contrôlant leur exécution via un arbitre.

Nous déclarons pour cela une tâche de planification composite qui contient deux tâches de planification : l'une préconisant la compression et l'autre l'effacement. Leur activation est contrôlée par un arbitre à jetons. Elle s'appuie sur les deux tâches d'exécution : compression et effacement.

La tâche de planification de compression est la plus prioritaire et possède quatre jetons, tandis que la tâche effacement n'en dispose que d'un seul. Chaque exécution d'une tâche coute un jeton, la tâche la plus prioritaire s'exécute. Le capital est reconstitué avec le temps. Du point de vue extérieur, ce composite s'utilise comme n'importe quelle tâche.

La figure 87 illustre l'utilisation de ce composite.

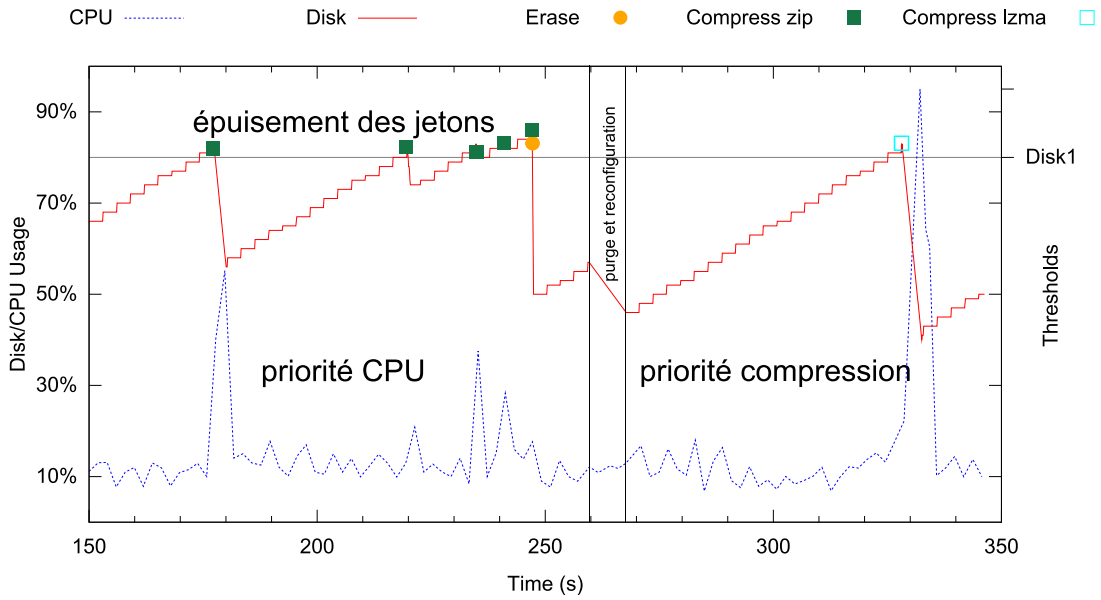

Figure 87 : Les tâches de compression et arbitrage par jetons.

Intéressons nous d'abord uniquement à la première partie de cette figure entre 150s et 260s. Entre 150s et 250s, la stratégie de compression est utilisée cinq fois. La première fois, la stratégie est efficace, la tâche de compression perd un jeton mais le regagne puisque le temps qui sépare la prochaine exécution est suffisant. Les quatre exécutions suivantes sont en revanche beaucoup moins concluantes et la tâche dépense l'intégralité de ses jetons. Alors la tâche d'effacement, qui était en concurrence sur toutes les exécutions précédentes mais qui perdait du fait de sa faible priorité, s'exécute finalement et apporte une solution vers 247s. Le cycle peut ainsi continuer, avant la prochaine exécution, la tâche de compression aura recouvré ses jetons. Cette stratégie simple permet d'introduire une méthode moins destructive de gestion de l'espace de stockage. Les images seront conservées pendant un temps plus long grâce à la compression.

Dans un deuxième temps, nous souhaitons mettre en concurrence des algorithmes de compression. Tous ces algorithmes ne sont pas égaux en temps et en consommation processeur et il convient de sélectionner le plus adapté au contexte. La tâche d'exécution de la compression peut donc s'implémenter de différentes façons.

Nous créons un nouveau composite d'exécution et concevons deux tâches qui utilisent les algorithmes de compression zip et lzma. Nous partons du principe que l'implémentation des tâches n'indique aucun élément comparatif, si bien que le choix de l'une ou de l'autre implémentation ne peut se faire sans exécuter ces tâches.

Chaque processeur des tâches de compression possède une méthode permettant l'estimation de la consommation processeur et de la compression utilisée. Occasionnellement, la tâche effectue la compression de quelques images et évalue le niveau de compression obtenu et la quantité de processeur obtenue. La méthode d'estimation est appelée par les coordinateurs et ajoutée aux requêtes des tâches. L'arbitre classe les tâches en fonction de ces informations et autorise l'exécution de la première. Notre arbitre est générique ; lors de sa configuration, l'expert indique dans l'ordre les clefs des champs numériques appartenant aux métadonnées qui doivent être comparées. L'arbitre classe selon la première clef et départage à l'aide des clefs suivantes. En cas d'égalité, le choix s'effectue sur la base du nom unique de la tâche. L'ordre des clefs peut changer. Dans notre scénario nous avons deux ordres : priorité CPU qui classe d'abord sur CPU puis sur compression, et son inverse priorité compression.

Nous avons utilisé ces tâches pour obtenir la figure 87. Du fait de la nature homogène des données, nos implémentations sont programmées pour faire leurs estimations une seule fois au démarrage. L'estimation sur la machine de test et avec l'échantillon de test donne environ pour la compression 35% pour le zip et 51,7% pour lzma, pour le processeur 30% zip et 50% lzma. Dans un premier temps et jusqu'à 260s nous donnons la priorité au processeur, l'algorithme zip est utilisé. Si le pic d'utilisation processeur ne dépasse pas 50% vers 180s, la compression ne dépasse que rarement 30% sur les images que nous utilisons. Les pics suivant sont beaucoup plus faibles puisque moins d'images sont compressées. Entre 260s et 270s, nous avons manuellement effacé les images pour que l'effet de l'algorithme lzma ne soit pas comparable. Vers 265s nous reconfigurons l'arbitre pour mettre l'accent sur la compression, c'est pourquoi l'utilisation de l'algorithme lzma commence vers 327s. Cet algorithme est très efficace mais provoque un pic de processeur important comme prévu par l'estimation. La durée d'exécution est également plus grande. Ce facteur pourrait également être rajouté comme élément de comparaison des algorithmes par l'arbitre.

Nous avons, dans ce cas, introduit les premières tâches en conflit. L'implémentation des arbitres est générique et ces derniers ont une connaissance limitée des tâches. Cependant leur utilisation permet d'apporter de la flexibilité et d'adapter la réponse de notre gestionnaire en fonction du contexte. L'utilisation d'une estimation fournie par les tâches est intéressante pour les comparer dans des situations mal définies. Par exemple sur certaines machines de tests, l'algorithme lzma est trop gourmand et ne peut pas être utilisé. Dans la suite nous introduisons un nouvel arbitre.

2.6 VERS UNE MEILLEURE GESTION DE L'ESPACE DISQUE

Nous souhaitons désormais prendre en considération l'apparition d'alarme pour la gestion du disque. D'autre part, nous voulons également tenir compte plus finement également l'utilisation du processeur. Nous réutilisons les tâches d'exécution permettant la compression et l'effacement que nous avons définis auxquelles nous ajoutons une tâche d'effacement sélectif qui efface en priorité les images avec une quantité de mouvement faible. Elle se sert des informations stockées par la tâche d'observation du détecteur de mouvement que nous avons développée plus haut.

L'idée est d'utiliser la tâche d'effacement lorsqu'il n'y a pas d'alarme et les tâches de compression et de sélection moins destructrices en cas d'alarme. Nous nous servons de la détection de seuil et rajoutons un nouveau seuil pour le disque (Disk1 à 80% et Disk2 à 90%), ainsi qu'un seuil pour le processeur (CPU à 95%)

Il y a de nombreuses façons d'implémenter cette gestion. Nous utilisons un arbitre fonctionnant par priorité et inhibition, ainsi que et les trois tâches de planification suivantes :

- **Plan.Standard** est la stratégie utilisée lorsqu'il n'y a pas d'alarme. Elle a la priorité la plus faible et s'abonne aux événements Disk1. La solution qu'elle propose est systématiquement l'effacement simple.
- **Plan.Alarm** est la stratégie utilisée en cas d'alarme et lorsque le processeur est disponible. Elle souscrit aux deux événements de Disk et vérifie la présence d'alarme. Lorsque disk1 est dépassé, elle propose une compression tant que la consommation ne dépasse pas disk2. Au-delà, elle propose un effacement simple. Son scheduler est configuré pour ne pas déclencher la tâche tant qu'il n'y a pas d'alarme.
- **Plan.AlarmCPU** est la stratégie la plus prioritaire. En cas d'alarme et lorsque la consommation CPU est très forte, la compression n'est plus possible. Elle propose donc d'utiliser la stratégie de sélection plus efficace, puis lorsque le seuil Disk2 est dépassé l'effacement simple. Son scheduler est configuré pour ne pas déclencher la tâche tant qu'il n'y a pas d'alarme ni de dépassement de seuil processeur.

La stratégie la plus prioritaire s'exécute et s'inhibe, ainsi que les autres stratégies, pendant un délai fixe (5s pour l'exemple).

La figure 88 illustre l'utilisation de ces tâches.

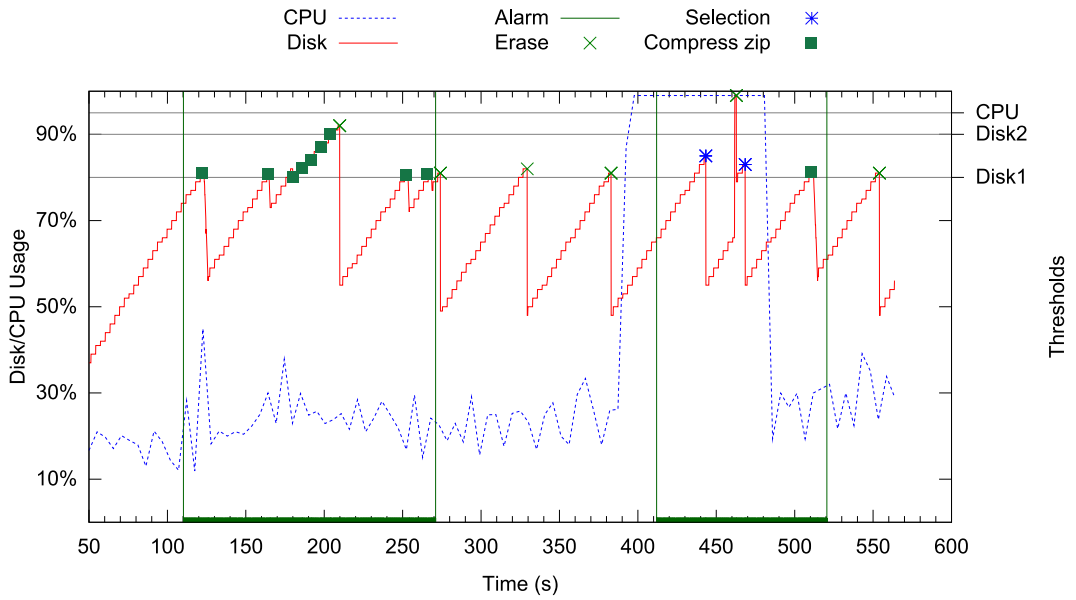

Figure 88 : gestion de l'espace disque en fonction de l'alarme.

Les points indiqués correspondent à l'activation des tâches d'exécution et non des tâches de planification. Entre 110s et 270s, les trois tâches sont en concurrence, mais la présence d'une alarme et la disponibilité du CPU donnent l'avantage à Plan.Alarm. Comme prévu, la compression est utilisée tant que l'usage du disque ne dépasse pas Disk2. C'est ce qui arrive vers 210s, où l'effacement simple est utilisé, après le gestionnaire utilise de nouveau la compression. A la fin de l'alarme entre 270s et 410s, seule la tâche Plan.Standard demande l'exécution et de fait la

stratégie d'effacement simple est utilisée à plusieurs reprises. Par la suite vers 390s et jusqu'à 480s nous simulons une utilisation intensive de processeur. Lorsqu'une alarme se déclenche vers 410s, les conditions sont alors réunies pour l'exécution de la tâche Plan.AlarmCPU qui l'emporte. Logiquement, la stratégie de sélection est utilisée. Nous simulons une arrivé brutale de données vers 450s ce qui provoque l'utilisation de la tâche d'effacement simple. Lorsque le processeur redevient disponible vers 480s, la tâche Plan.Alarm prend le relai puis, lorsque l'alarme disparait, la tâche Plan.Standard lui succéde.

Dans ce scénario avec l'utilisation d'un arbitre simple, nous adaptons la réponse du gestionnaire en fonction du contexte. L'ajout ou le retrait de nouvelles tâches et le développement de nouvelles stratégies est simple, car il suffit de modifier les priorités des tâches. Par exemple, si la tâche Plan.Alarm fait défaut ou se bloque, l'application continuera de fonctionner correctement sans modification du gestionnaire qui utilisera Plan.Standard en alternance avec Plan.AlarmCPU. A partir des statistiques fournies, l'administrateur ou un module d'adaptation pourra modifier le comportement du gestionnaire en changeant les priorités ou en retirant un module.

2.7 UTILISATION CONJOINTE DES TACHES DEVELOPPEES.

Chaque scénario a été développé indépendamment, nous voulons maintenant utiliser l'ensemble des tâches que nous avons développées depuis le début.

Nous remplaçons la tâche Plan.Alarm par la tâche que nous avions utilisée pour la compression. De fait, le nouveau comportement sera d'utiliser des jetons et non plus le seuil disk2. Le reste du gestionnaire ne change pas. Ici nous utilisons le mode priorité à la compression et il n'y a pas de caméra sur batterie.

La figure 89 illustre l'utilisation de l'ensemble des tâches que nous avons définies précédemment. Par rapport à la figure précédente, on constate ici la gestion des caméras et du détecteur de mouvement. Entre 50 et 180s, lorsque l'alarme est déclenchée, l'arbitre à jetons est utilisé (deux jetons pour compression contre un pour effacement). Le seuil de Disk2 n'est plus atteint.

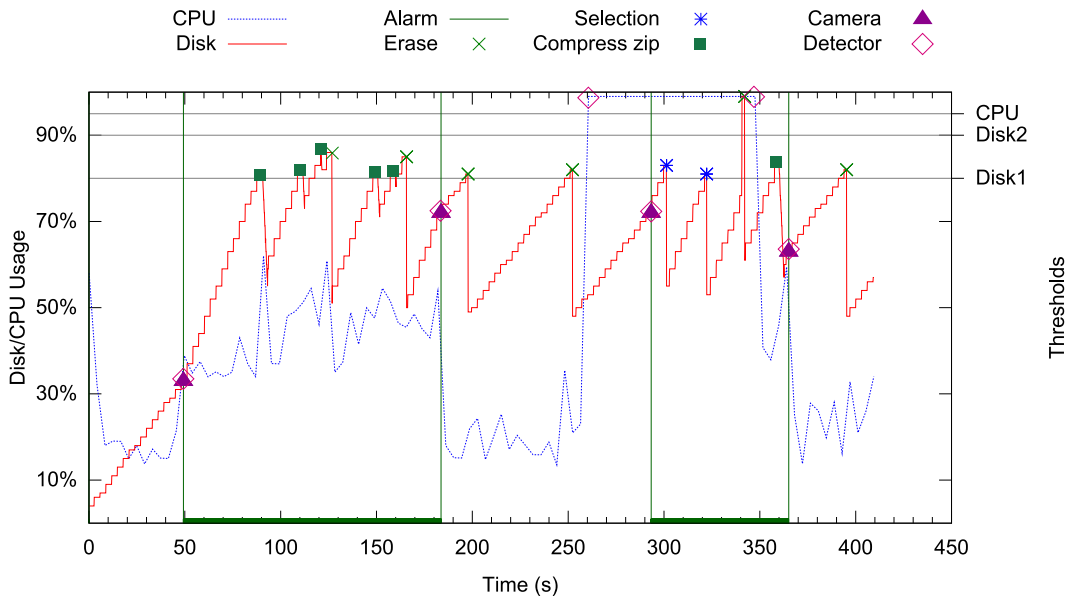

Figure 89 : Utilisation de toutes les tâches.

Dans ce cas, il n'y a pas de conflit entre les différents scénarios qui peuvent être déployés et utilisés séparément. Ainsi il est possible de n'utiliser qu'une version naïve de la gestion du

disque lorsque l'utilisateur se trouve chez lui par exemple. L'utilisation de tâches offre beaucoup de flexibilité à l'exécution et permet de modifier radicalement la façon dont le gestionnaire fonctionne.

2.8 PERFORMANCES

Nous souhaitions vérifier que l'utilisation des arbitres et le découpage n'aient pas un impact trop important sur les performances générales de notre gestionnaire. Dans cet objectif, nous avons développé une version ad-hoc du gestionnaire basée sur iPOJO avec pour seul découpage les blocs MAPE. Nous avons ensuite évalué le temps de réponse de chaque gestionnaire en simulant une application. Cette dernière simule des ensembles de valeurs et vérifie que la réponse du gestionnaire correspond à la réponse attendue tout en évaluant le temps de réaction. En moyenne, l'utilisation du framework entraine une perte de performance d'environ 10 à 20%, ce qui est normal compte tenu des échanges entre les tâches et l'arbitre. Le dynamisme a toujours un prix en performance.

3 SYNTHESE

Dans ce chapitre nous avons donné un exemple de développement de gestionnaire autonomique par l'utilisation de tâches. Ce cas d'application nous a permis de mettre en évidence les situations pour lesquelles l'emploi de tâches se révèle utile. Progressivement, nous avons construit le gestionnaire en ajoutant de nouveaux aspects gérés par les tâches.

L'utilisation de notre approche a permis de développer chaque préoccupation en séparation. En définitive, c'est plusieurs sous-gestionnaires que nous avons développés. Nous avons pu réutiliser un certain nombre de tâches de supervision et d'exécution, ainsi que les tâches de détection de seuil, pour construire chacun de ces aspects. Ainsi nous avons pu tester et évaluer séparément des parties de la boucle de contrôle. Nous avons montré que cet ensemble de tâches pouvait être utilisées simultanément pour obtenir un comportement du gestionnaire globalement complexe. Grâce aux mécanismes de sélection, il est possible d'obtenir un comportement globalement cohérent.

Les tâches d'administration permettent d'étendre facilement le comportement du gestionnaire. C'est ce que nous avons réalisé avec la tâche de synthèse lors de la gestion des alarmes. Il est très facile de modifier le comportement du gestionnaire pour intégrer de nouvelles informations contextuelles.

A cette occasion, nous avons également montré que les tâches pouvaient, à la différence de simples règles par exemple, modifier leur configuration au cours du temps. C'est le cas des tâches qui apprennent automatiquement à éliminer les fausses alarmes.

D'autre part, nous avons montré des cas d'adaptation de l'architecture du gestionnaire à son environnement. D'abord à son état interne, lorsque des tâches n'étaient pas assez performantes ou se bloquaient ; un module simple d'auto-adaptation du gestionnaire est alors capable de les remplacer. Ensuite, lorsque les priorités de l'administrateur changent avec l'utilisation d'algorithmes de compression différents en fonction des indications fournies par l'administrateur.

Dans ce chapitre, nous nous sommes essentiellement concentrés sur les techniques d'arbitrage. Nous avons pu obtenir des résultats similaires par filtrage, voir à ce sujet [146] et [145].

Bien sûr ce dynamisme a un léger coût en performance qui peut atteindre 20% par rapport à un gestionnaire ad-hoc, mais ce dernier n'est pas aussi adaptable. Le développement est également plus long. Ce coût de développement initial, lié principalement à la verbosité de notre ADL, est compensé par la réutilisation des tâches et par la facilitation de la maintenance. Le développement d'un environnement spécialisé pourrait atténuer cette différence.

Chapitre 8 - **Conclusion**

Ce dernier chapitre synthétise les idées principales de notre proposition et ouvre des perspectives à partir de notre contribution. Les recherches présentées ont définies un framework permettant la construction de gestionnaires dynamiques et extensibles. Elles offrent une base à des travaux ultérieurs.

1 SYNTHESE

Dans cette thèse, nous nous sommes focalisés sur l'architecture et le développement de gestionnaires autonomiques. Nous avons présenté dans une première partie le domaine de l'informatique autonomique en insistant sur les architectures proposés dans la littérature ; puis nous nous sommes intéressés aux approches à services qui nous semblent prometteuses pour apporter du dynamisme et de l'extensibilité aux gestionnaires. Nous avons décrit ensuite notre approche, d'abord de façon globale en expliquant les principes puis en détaillant son architecture et une implémentation particulière. Nous avons enfin illustré son utilisation par la construction d'un gestionnaire sur un cas d'application.

1.1 CONTEXTE

Les bénéfices attendus de l'informatique autonomique sont nombreux : diminution des couts de maintenances, réduction des erreurs dues aux opérateurs humains, optimisation et personnalisation des services proposées aux clients. Il ne fait aucun doute que l'autonomisation des applications et des systèmes va représenter un enjeu de taille pour les entreprises dans le futur, conférant un avantage concurrentiel important à celles qui s'y attèlent.

L'idée est simple : déporter les tâches administratives de la maintenance aux systèmes, conférer un degré d'autonomie aux systèmes. L'objectif est ambitieux mais pas irréaliste : il ne s'agit pas de remplacer l'humain. Comme nous l'avons vu au travers de la présentation du domaine, il existe une progression des systèmes traditionnels vers les systèmes autonomiques. De plus le domaine s'appuie sur de nombreux travaux réalisés depuis longtemps dans d'autres domaines : l'intelligence artificielle, la robotique, les approches multi-agents et même la biologie qui lui donne son nom. Mais construire un système autonomique n'est pas pour autant facile et il ne suffit pas d'assembler empiriquement des algorithmes provenant de tel ou tel domaine pour avoir un système autonomique : il faut une méthode, des techniques, un cadre de développement.

Nous avons vu qu'il existe de nombreux travaux permettant de concevoir des applications autonomiques. Souvent ad-hoc, parfois généralistes, ces plateformes se focalisent cependant essentiellement sur la communication entre les gestionnaires et assez peu sur l'architecture interne de ces derniers. D'autres plateformes se focalisent sur une technique très particulière de gestion : par exemple Rainbow se focalise sur l'observation de l'architecture. Si ce projet est généraliste, il n'en reste pas moins difficile à appliquer à tous les types de systèmes et notamment ceux ou l'architecture n'est pas apparente ou peu dynamique.

Souvent les gestionnaires sont conçus à base de règles ou de composants inamovibles, avec une architecture qui, lorsqu'elle n'est pas monolithique, se réclame du modèle de référence d'IBM ou y ressemble. Ce modèle n'est pas mauvais en soi – c'est une excellente architecture logique qui identifie les activités nécessaires à la réalisation d'une boucle autonomique – mais nous considérons qu'il ne s'agit pas d'une architecture d'implémentation et qu'il ne devrait pas être utilisé de façon littérale et systématique.

Il ne faut pas oublier que les applications évoluent et sont dynamiques. L'utilisation de règles permet la modification dynamique du comportement du gestionnaire mais au prix d'une maintenabilité complexe. Il est difficile de comprendre le comportement du gestionnaire. Les composants, quand à eux, ne sont généralement utilisés que pour permettre le développement séparé des grandes activités d'IBM et pratiquement jamais pour des questions de dynamisme. Actuellement, pour modifier le comportement d'un gestionnaire il faut se contenter de jouer sur les propriétés de configuration exposées, ce qui offre peu de flexibilité. Une modification profonde exige son arrêt, recompilation ou réassemblage, réinstallation et redémarrage. Cette méthode de construction n'est clairement pas adaptée aux évolutions actuelles des systèmes ni au besoin de dynamisme que nous avons beaucoup évoqué dans la description de l'approche à services.

A notre sens, pour accélérer le développement des applications autonomiques il faut un modèle de développement homogène favorisant la réutilisation des fonctionnalités. Nous considérons que de par la trop forte granularité de ces blocs architecturaux, l'architecture de référence MAPE-K d'IBM ne satisfait que trop partiellement à cette exigence. Les différentes préoccupations de gestion, les différents auto-* que nous avons évoqués, sont mêlés les uns aux autres ce qui non seulement complique la compréhension du gestionnaire, mais complique également la réutilisation.

Ce sont ces différentes lacunes qui ont motivé nos travaux. Nous nous sommes notamment intéressés à l'application des concepts de l'approche orientée service au domaine de l'informatique autonomique pour améliorer le dynamisme.

1.2 NOTRE CONTRIBUTION

Notre objectif était le développement d'un modèle de programmation qui :

- identifie et sépare clairement les concepts appartenant de façon générique à l'informatique autonomique des concepts métiers propres à l'application ou au système considérée ;
- permette la réutilisation des fonctionnalités autonomiques récurrentes pour accélérer et simplifier le développement ;
- permette la modification en profondeur du comportement du gestionnaire en apportant dynamisme et extensibilité. ;
- soit compatible avec les outils existant permettant la réalisation des différentes étapes de la boucle autonomique : algorithmes d'inférence, outils de supervision entre autres.

Pour atteindre nos objectifs, nous avons défini le concept de tâche d'administration, nouveau niveau d'abstraction intermédiaire et complémentaire au modèle MAPE-K. Les tâches d'administration sont des unités fortement spécialisées dans une activité de gestion particulière : récupérer la valeur du processeur, trouver une solution à un problème particulier ou identifier la présence d'un utilisateur par exemple. Leur forte cohérence et leur granularité plus réduite, en comparaison au bloc MAPE, simplifient leur développement et facilitent la compréhension du gestionnaire. La tâche constitue également une unité d'encapsulation réutilisable destinée à être utilisée d'un gestionnaire à l'autre.

La granularité plus forte que celle d'une simple règle – une tâche peut tout à fait être réalisée par un ensemble de règles – évite un morcellement trop fort qui compliquerait la maintenance et l'évolution. Ainsi les tâches ont une action directement mesurable sur l'activité du gestionnaire. Leur fonctionnalité est décrite par un type qui n'induit pas une implémentation particulière. L'utilisateur de la tâche peut, comme pour un service, l'employer sans se soucier des contingences liées à la plateforme d'exécution.

Le comportement global du gestionnaire provient de la coopération de cet ensemble de tâches. Au fil du temps, l'ensemble des tâches actives évolue en fonction des besoins, des objectifs et de l'historique des exécutions. Chaque tâche expose les informations suffisantes à l'évaluation de son action : il est ainsi possible de sélectionner les tâches les plus appropriées au besoin du moment.

Nous avons présenté une architecture générale de notre framework. Au cœur, l'architecture de la tâche est conçue de telle façon que les différentes préoccupations liées à sa conception sont séparées : fonctionnalité, implémentation de l'algorithme métier, condition dans lesquelles la tâche s'active, relation des tâches. L'architecture est ainsi fortement modulaire et offre de nombreuses possibilités d'extension. Au-dessus, nous avons décrit l'architecture de contrôle et les différents mécanismes de gestion de conflits. Ici, encore, notre objectif était d'offrir

le plus de souplesse possible aux développeurs en leur permettant de concevoir leur propre algorithmes.

Notre approche permet la conception des gestionnaires dans un modèle indépendant de la plateforme. L'expert décrit l'architecture du gestionnaire souhaité en termes de fonctionnalités (types de tâche) ; le framework se charge de trouver et d'instancier les implémentations appropriées. De cette façon, les gestionnaires peuvent être portées d'une machine à l'autre et concepteur peut se focaliser complètement sur le comportement du gestionnaire.

Nous n'avons pas voulu orienter la présentation de notre plateforme vers un modèle de programmation particulier. C'est ce que nous avons fait dans la suite en proposant dans notre réalisation un modèle à composants. Ce dernier s'appuie sur l'approche orientée service pour supporter le dynamisme proposé par notre approche. Nous fournissons un framework pleinement fonctionnel et un certain nombre d'implémentations alternatives de mécanismes pouvant ainsi servir de modèles aux développeurs. Au-dessus de ce framework, nous fournissons des outils d'administration permettant l'observation et la modification dynamique du comportement du gestionnaire.

Notre approche permet donc la conception de gestionnaires :

- **modulaires** en permettant de définir chacun des aspects de la gestion dans un bloc architectural dédié ;
- **homogènes** en se reposant sur le concept unique de tâche d'administration pour la réalisation de l'ensemble des activités de gestion ;
- **flexibles** en permettant l'extension et la redéfinition d'un grand nombre de fonctionnalités : les développeurs peuvent ainsi personnaliser le framework en fonction de leurs besoins ;
- **évolutifs** en permettant la modification de la composition de l'ensemble des tâches et la façon dont elles coopèrent ;
- **dynamiques** car ces adaptations du comportement peuvent intervenir à tout moment de l'exécution en fonction des besoins et des changements d'objectif ;
- **administrables** en fournissant des interfaces d'administration, une interface graphique, en permettant la conception du gestionnaire avec un modèle abstrait et en remontant l'ensemble des informations nécessaires à son adaptation dans un modèle du gestionnaire en exécution.

Nous avons mis en pratique notre approche sur un cas d'utilisation lié à la domotique. Nous avons en particulier montré que les tâches permettaient le développement indépendant des différents objectifs de la gestion ; qu'elles pouvaient, contrairement à des règles, adapter leur comportement en tirant les leçons des exécutions précédentes ; et que les mécanismes de sélections, lorsqu'ils sont utilisées, permettent d'assurer un comportement globalement cohérent au gestionnaire. L'impact sur les performances est modéré en regard de la flexibilité et du dynamisme gagné. La mise en commun de toutes les tâches a permis de créer un gestionnaire au comportement globalement complexe.

2 PERSPECTIVES

2.1 AUTO-ADAPTATION DU GESTIONNAIRE

L'idée d'auto-adapter un gestionnaire peut sembler curieuse de prime abord, car dès lors quand s'arrêter ? Elle nous semble cependant une piste de recherche intéressante et une suite logique à notre travail. D'ailleurs n'avons-nous pas commencé à automatiser certains remplacements de tâche dans notre exemple ?

Nous avons conçu notre framework pour qu'il fournisse toutes les informations nécessaires à l'évaluation des tâches. Nous pensons qu'il est possible d'automatiser l'utilisation de ces informations pour adapter sans l'intervention de l'administrateur ou d'un expert le comportement du gestionnaire.

Comme nous l'avons dit dans l'état de l'art, l'une des difficultés de l'informatique autonomique est la transformation des objectifs de haut-niveau en actions. Notre framework n'apporte pour l'instant pas de solution à ce problème particulier : tout au plus augmente-t-il le niveau d'abstraction des briques logicielles utilisées. Nous pensons que c'est le rôle d'un gestionnaire d'auto-adaptation de transformer ces objectifs abstraits en actions concrètes et pour ce faire de modifier la topologie du gestionnaire.

Cette adaptation n'a pas nécessairement besoin d'être complexe. Une approche naïve que nous avons déjà évoquée est de fonctionner par règles ou scénarios – chaque scénario étant déclenché par l'apparition d'un contexte particulier. Une autre solution est de s'appuyer sur les travaux réalisés par les systèmes spécialisés dans l'analyse et la modification de l'architecture tels que Rainbow.

L'un des premiers usages de l'auto-adaptation devrait être la vérification de la cohérence du comportement du gestionnaire. Nous pensons qu'il est possible, à partir des informations décrites dans les modèles de types et de données, de calculer l'ensemble des boucles autonomiques réalisables – en tout cas pour les gestionnaires de petites tailles. Nous souhaiterions utiliser ses informations pour vérifier que le comportement du gestionnaire sera stable et cohérent, et éventuellement pouvoir prouver que le gestionnaire ne se bloquera pas. Il sera cependant très probablement nécessaire d'ajouter des informations au modèle pour mieux contraindre les gestionnaires.

L'auto-adaptation des gestionnaires fait également sens pour les grilles de calculs. Dans un parc comportant des systèmes autonomiques identiques, chacun géré par un gestionnaire conçu selon notre approche, un module d'auto-adaptation pourrait tirer parti des informations provenant de chacun des gestionnaires. Il est ainsi possible de mettre en commun les informations pour déterminer les problèmes récurrents, identifier les solutions efficaces : la meilleure topologie de l'application, les meilleurs ensembles de tâches en fonction du contexte. Le dynamisme et le caractère opportuniste de nos gestionnaires rend l'exploration de plusieurs solutions possible à moindre coût. Le module d'auto-adaptation peut par exemple tester ses solutions sur un sous-ensemble de gestionnaires avant de mettre en œuvre une solution globale.

Mais dans ce cas pourquoi ne pas continuer l'adaptation, et auto-adapter l'auto-adaptation ? Nous pensons que ce ne sera pas nécessaire. Nous croyons qu'il est possible de trouver un ensemble de politiques d'adaptation génériques et simples employables sur la plupart des gestionnaires. Pour cela, il sera peut-être nécessaire de monter encore un peu le niveau d'abstraction de notre framework et de la description du gestionnaire.

2.2 DISTRIBUTION

La possibilité d'utiliser des tâches à distance et de distribuer la configuration du gestionnaire nous paraît intéressante. Dans la plupart des systèmes la distribution a plusieurs raisons : le passage à l'échelle, la séparation des préoccupations ou la confidentialité.

C'est avant tout pour le premier point que nous considérons que la distribution est utile. Il existe en effet de nombreuses machines aux capacités réduites qui doivent cependant être autonomisées. Certaines opérations, comme trouver une solution particulière à un problème ardu, peuvent demander des ressources telles qu'il faudra externaliser certaines tâches pour éviter de surcharger le processeur de ces tâches. Les tâches les plus simples continueront quant à elles à s'exécuter sur la plateforme pour garantir une réactivité optimale : l'exécution des boucles autonomiques les plus simples se poursuivra donc localement. Cette solution nous semble bien plus adaptée et performante qu'une externalisation pure et simple du gestionnaire.

Nous considérons que la séparation des préoccupations est déjà suffisante dans notre approche. Dans ce cadre, distribuer encore plus les tâches et le contrôle a un coût qui nous semble élevé pour un gain qui nous paraît faible.

La confidentialité est un facteur qu'il ne faut pas négliger. Certaines entreprises peuvent vouloir contrôler l'utilisation de leurs algorithmes et ne pas souhaiter distribuer certaines tâches. A long terme, on peut envisager, par exemple, la monétisation de l'utilisation de certaines tâches possédant un algorithme particulièrement optimisé et efficace – son emploi nécessitant par exemple un abonnement. Les Web Service ont notamment ouvert la voix à de telles pratiques.

Bien sûr la distribution a un coût et nécessite une étude particulière pour optimiser les communications entre les tâches. La résolution de conflit telle que nous la concevons pour l'instant avec l'échange de messages pourrait en pâtir : il sera sans doute nécessaire de repenser ces mécanismes. Il en va de même pour la base de données, bien qu'elle ne joue qu'un rôle secondaire dans notre gestion puisque chaque tâche possède un espace de stockage pour accumuler les données. La mise en place de proxys localisés pour accéder à la base de données pourrait éventuellement constituer une solution intéressante.

2.3 ATELIER ET OUTILS DE DEVELOPPEMENT

Dans cette thèse nous nous sommes focalisés sur le développement du middleware et l'implémentation du support d'exécution. Pour l'heure, si l'on exclut la réutilisation de tâches, l'écriture d'un gestionnaire avec notre approche est notablement plus longue qu'un développement ad-hoc, ce qui est normal puisqu'elle nécessite la description d'un grand nombre d'éléments. En particulier, le choix du XML pour la description des modèles est un parti pris qui a un coût en développement : l'écriture des descriptions est fastidieuse.

Cette situation n'est pas une fatalité, le développement d'outils spécialisés pour le développement de tâches devrait permettre de diminuer significativement le temps nécessaire à la réalisation d'un gestionnaire. La mise en œuvre d'un atelier de développement [147] offrant un environnement de développement spécialisé pour la création de tâches nous semble indispensable à long terme. Le niveau d'abstraction fourni par le framework, notamment avec les types de tâches et de données, devrait faciliter la réalisation d'un tel outil.

Chapitre 9 - **Bibliographie**

[1] J. Gray, “What Next? A dozen remaining IT problems,” *Turing Award 1999 Lecture: www. research. microsoft. com/gray*, 1999.

[2] M. Weiser, “The computer for the 21st century,” *Scientific American*, p. 94-104, 1991.

[3] B. Randell, “The 1968/69 nato software engineering reports,” *History of Software Engineering*, 1996.

[4] A. Weiss, “Computing in the clouds,” *COMPUTING*, vol. 16, 2007.

[5] S. Lightstone, “Foundations of Autonomic Computing Development,” in *Engineering of Autonomic and Autonomous Systems, 2007. EASe '07. Fourth IEEE International Workshop on*, p. 163-171, 2007.

[6] D. A. Patterson, “A simple way to estimate the cost of downtime,” in *Proc. 16th Systems Administration Conf.| LISA*, p. 185–8, 2002.

[7] H. W. Lawson, “Rebirth of the computer industry,” 2002.

[8] R. Sterritt, “Why Computer-Based Systems Should be Autonomic,” *IN PROC. 12 TH IEEE INTERNATIONAL CONFERENCE ON ENGINEERING COMPUTER-BASED SYSTEMS (ECBS 2005*, vol. 4, p. 406--412, 2005.

[9] A. G. Ganek, C. P. Hilkner, J. W. Sweitzer, B. Miller, J. L. Hellerstein, et I. B. M. Coporation, “The response to IT complexity: autonomic computing,” in *Third IEEE International Symposium on Network Computing and Applications, 2004.(NCA 2004). Proceedings*, p. 151–157, 2004.

[10] D. A. Patterson, “Availability and maintainability≫ performance: New focus for a new century,” *Key Note at FAST*, vol. 2, 2002.

[11] J. Bourcier, “Auto-Home : une plate-forme pour la gestion autonomique d'applications pervasives,” University Joseph Fourier, 2008.

[12] P. Horn, “Autonomic Computing: IBM's Perspective on the State of Information Technology,” *IBM*, 2001.

[13] J. O. Kephart, “Research challenges of autonomic computing,” in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, p. 15–22, 2005.

[14] M. G. Hinchey et R. Sterritt, “99% (Biological) Inspiration...,” in *Proceedings of the Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, p. 187-195, 2007.

[15] D. Ghosh, R. Sharman, H. Raghav Rao, et S. Upadhyaya, “Self-healing systems—survey and synthesis,” *Decision support systems*, vol. 42, n°. 4, p. 2164–2185, 2007.

[16] P. P. Grassé, “La reconstruction du nid et les coordinations interindividuelles chezBellicositermes natalensis etCubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs,” *Insectes sociaux*, vol. 6, n°. 1, p. 41–80, 1959.

[17] A. Colorni, M. Dorigo, V. Maniezzo, et others, “Distributed optimization by ant colonies,” in *Proceedings of the First European Conference on Artificial Life*, vol. 142, p. 134–142, 1991.

[18] J. G. Hansen, E. Christiansen, et E. Jul, “The Laundromat Model for Autonomic Cluster Computing,” in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, p. 114-123, 2006.

[19] M. Zhang, P. Martin, W. Powley, et P. Bird, “Using economic models to allocate resources in database management systems,” in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008.

[20] G. Tesauro et al., “A Multi-Agent Systems Approach to Autonomic Computing,” in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, p. 464–471, 2004.

[21] T. De Wolf et T. Holvoet, “Towards Autonomic Computing: agent-based modelling, dynamical systems analysis, and decentralised control,” in *Proceedings of the First International Workshop on Autonomic Computing Principles and Architectures*, p. 10, 2003.

[22] A. Moreno et J. Pavn, *Issues in Multi-Agent Systems: The AgentCities.ES Experience (Whitestein Series in Software Agent Technologies and Autonomic Computing)*. 2007.

[23] B. Srivastava, “The Case for Automated Planning in Autonomic Computing,” in *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, p. 331–332, 2005.

[24] J. O. Kephart et D. M. Chess, “The vision of autonomic computing,” *Computer*, p. 41–50, 2003.

[25] R. A. Brooks, “Intelligence without representation,” *Artif. Intell.*, vol. 47, n°. 1, p. 139-159, 1991.

[26] J. O. Kephart et W. E. Walsh, “An artificial intelligence perspective on autonomic computing policies,” in *Policies for Distributed Systems and Networks, 2004. POLICY 2004 Proceedings.*, p. 3–12, 2004.

[27] R. Want, T. Pering, et D. Tennenhouse, “Comparing autonomic and proactive computing,” *IBM Systems Journal*, vol. 42, n°. 1, p. 129–135, 2003.

[28] J. Ahola, “Ambient intelligence,” *ERCIM News*, vol. 47, 2001.

[29] M. Parashar et S. Hariri, “Autonomic Computing: An Overview.,” in *UPP*, p. 257-269, 2004.

[30] R. Murch, *Autonomic computing*. IBM Press, 2004.

[31] R. Sterritt, “Autonomic computing,” *Innovations in systems and software engineering*, vol. 1, n°. 1, p. 79–88, 2005.

[32] R. Sterritt et M. Hinchey, “Autonomic computing - panacea or poppycock?,” in *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, p. 535-539, 2005.

[33] H. A. Mueller, L. O'Brien, M. Klein, et B. Wood, *Autonomic Computing*. 2006.

[34] J. O. Kephart et W. E. Walsh, “An artificial intelligence perspective on autonomic computing policies,” in *Policies for Distributed Systems and Networks, 2004. POLICY 2004 Proceedings.*, p. 3–12, 2004.

[35] M. C. Huebscher et J. A. McCann, “A survey of autonomic computing—degrees,

models, and applications," 2008.
[36] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, et E. Gjorven, "Using Architecture Models for Runtime Adaptability," *IEEE Software*, vol. 13, p. 62, 2006.
[37] T. Strang et C. Linnhoff-Popien, "A context modeling survey," in *Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp*, 2004.
[38] K. Henricksen et J. Indulska, "Developing context-aware pervasive computing applications: Models and approach," *Pervasive and Mobile Computing*, vol. 2, n°. 1, p. 37–64, 2006.
[39] A. K. Dey et G. D. Abowd, "Towards a better understanding of context and context-awareness," in *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, p. 304–307, 2000.
[40] B. Schilit, N. Adams, R. Want, et others, "Context-aware computing applications," in *Proceedings of the workshop on mobile computing systems and applications*, p. 85–90, 1994.
[41] T. G. Xiao, X. H. Wang, H. K. Pung, et D. Q. Zhang, "An Ontology-based Context Model in Intelligent Environments," *IN PROCEEDINGS OF COMMUNICATION NETWORKS AND DISTRIBUTED SYSTEMS MODELING AND SIMULATION CONFERENCE*, p. 270--275, 2004.
[42] V. Tsetsos et S. Hadjiefthymiades, "An innovative architecture for context foraging," in *Proceedings of the Eighth ACM International Workshop on Data Engineering for Wireless and Mobile Access*, p. 41–48, 2009.
[43] B. Miller, *Can you CHOP up autonomic computing?* 2005.
[44] M. Wooldridge et N. R. Jennings, "Intelligent agents: Theory and practice," *Knowledge engineering review*, vol. 10, n°. 2, p. 115–152, 1995.
[45] M. Salehie et L. Tahvildari, "Autonomic computing: emerging trends and open problems," in *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, p. 1–7, 2005.
[46] T. De Wolf et T. Holvoet, *Autonomic Computing: Concepts, Infrastructure, and Applications, chapter A Taxonomy for Self-* Properties in Decentralised Autonomic Computing*. CRC Press, to appear in, 2006.
[47] *An Architectural Blueprint for Autonomic Computing*. IBM, 2006.
[48] N. Damianou, N. Dulay, E. Lupu, et M. Sloman, "A Language for Specifying Security and Management Policies for Distributed Systems," *Imperial College Research Report DoC*, vol. 1, 2000.
[49] E. Lupu et al., "AMUSE: autonomic management of ubiquitous e-Health systems," *Concurrency and Computation*, vol. 20, n°. 3, p. 277, 2008.
[50] L. Lymberopoulos, E. Lupu, et M. Sloman, "An adaptive policy-based framework for network services management," *Journal of Network and Systems Management*, vol. 11, n°. 3, p. 277–303, 2003.
[51] D. Garlan, S. Cheng, A. Huang, B. Schmerl, et P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, n°. 10, p. 46–54, 2004.
[52] W. E. Walsh, G. Tesauro, J. O. Kephart, et R. Das, "Utility functions in autonomic systems," in *Proceedings of the International Conference on Autonomic Computing*, p.

70–77, 2004.
[53] D. M. Chess, A. Segal, I. Whalley, et S. R. White, “Unity: Experiences with a prototype autonomic computing system,” in *Proceedings of the First International Conference on Autonomic Computing*, p. 140–147, 2004.
[54] S. J. Russell et P. Norvig, *Artificial intelligence: a modern approach*. Prentice hall, 2009.
[55] B. D. Ziebart, D. Roth, R. H. Campbell, et A. K. Dey, “Learning automation policies for pervasive computing environments,” in *Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings*, p. 193–203, 2005.
[56] A. Ranganathan et R. H. Campbell, “Autonomic pervasive computing based on planning,” in *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, p. 80–87, 2004.
[57] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, et E. Brewer, “Failure diagnosis using decision trees,” in *International Conference on Autonomic Computing*, 2004.
[58] S. Sicard, F. Boyer, et N. D. Palma, “Using components for architecture-based management: the self-repair case,” in *Proceedings of the 30th international conference on Software engineering*, p. 101-110, 2008.
[59] X. Dong et al., “Autonomia: an autonomic computing environment,” in *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, p. 61–68, 2003.
[60] B. Miller, *The "Standard" way of autonomic computing*. 2005.
[61] D. Garlan, J. Kramer, et A. Wolf, Éd., “Proceedings of the first workshop on Self-healing systems,” Charleston, South Carolina, p. 120, 2002.
[62] N. De Palma, S. Bouchenak, F. Boyer, D. Hagimont, S. Sicard, et C. Taton, “Jade: un environnement d'administration autonome,” presented at the Technique et Science Informatique, 2007.
[63] R. E. Fikes et N. J. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” *Artificial intelligence*, vol. 2, n°. 3, p. 189–208, 1971.
[64] A. L. Blum et M. L. Furst, “Fast planning through planning graph analysis* 1,” *Artificial intelligence*, vol. 90, n°. 1, p. 281–300, 1997.
[65] T. John W. Sweitzer et T. Christine Draper, *Autonomic Computing: Concepts, Infrastructure, and Applications, chapter Architecture Overview for Autonomic Computing*. CRC Press, to appear in, 2006.
[66] S. A. Gurguis et A. Zeid, “Towards autonomic web services: Achieving self-healing using web services,” in *Proceedings of DEAS’05*, 2005.
[67] G. Tesauro et al., “A Multi-Agent Systems Approach to Autonomic Computing,” in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, p. 464–471, 2004.
[68] M. Agarwal et al., “Automate: Enabling autonomic applications on the grid,” in *Proceedings of the Autonomic Computing Workshop*, vol. 2003, p. 48–57, 2003.
[69] X. Chen et M. Simons, “A component framework for dynamic reconfiguration of distributed systems,” *Component Deployment*, p. 131–155, 2002.
[70] F. Baude, L. Henrio, et P. Naoumenko, “A component platform for experimenting with autonomic composition,” in *Proceedings of the 1st international conference on*

Autonomic computing and communication systems, p. 1–9, 2007.

[71] IBM Redbooks, *A Practical Guide to the IBM Autonomic Computing Toolkit (IBM Redbooks)*. .

[72] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, et Y. Diao, "ABLE: A toolkit for building multiagent autonomic systems," *IBM Systems Journal*, vol. 41, n°. 3, p. 350–371, 2002.

[73] G. Kaiser, G. Valetto, J. Parekh, et P. Gross, *Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems*. Citeseer, 2005.

[74] S. Cheng, "Rainbow: Cost-effective, Software Architecture-based Self-adaptation," Carnegie Mellon University, 2008.

[75] D. Patterson et al., *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. EECS Department, University of California, Berkeley, 2002.

[76] S. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitnagel, et P. Steenkiste, "Using Architectural Style as a Basis for System Self-repair," in *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, p. 45-59, 2002.

[77] D. Garlan, S. W. Cheng, et B. Schmerl, "Increasing system dependability through architecture-based self-repair," *Lecture Notes in Computer Science*, p. 61–89, 2003.

[78] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, et J. B. Stefani, "The fractal component model and its support in java," *Software-Practice and Experience*, vol. 36, n°. 11, p. 1257–1284, 2006.

[79] S. Bouchenak, N. De Palma, D. Hagimont, et C. Taton, "Autonomic management of clustered applications," in *IEEE International Conference on Cluster Computing*, 2006.

[80] Sicard, Taton, Bouchenak, Boyer, et N. de Palma, "Coordination of Multiple Autonomic Managers.," presented at the The Second EuroSys Authoring Workshop, Lisbon, Portuga, 2007.

[81] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, A. Segal, et J. O. Kephart, "Autonomic computing: Architectural approach and prototype," *Integr. Comput.-Aided Eng.*, vol. 13, n°. 2, p. 173–188, 2006.

[82] G. Tesauro et al., "A multi-agent systems approach to autonomic computing," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, p. 464–471, 2004.

[83] M. P. Papazoglou, "Service -Oriented Computing: Concepts, Characteristics and Directions," in *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, p. 3, 2003.

[84] J. Yang, H. Chen, S. Hariri, et M. Parashar, "Autonomic Runtime Manager for Large Scale Adaptive Distributed Applications," in *EEE International Symposium on High Performance Distributed Computing*, 2005.

[85] H. Liu et M. Parashar, "Accord: A programming framework for autonomic applications," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 36, n°. 3, p. 341–352, 2006.

[86] Z. Li et M. Parashar, "Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications," in *Proceedings of The International Conference on Autonomic Computing*, p. 278–279, 2004.

[87] N. Jiang, A. Quiroz, C. Schmidt, et M. Parashar, “Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments,” *Concurr. Comput. : Pract. Exper.*, vol. 20, n°. 12, p. 1455-1484, 2008.
[88] J. A. McCann et M. C. Huebscher, “Evaluation issues in autonomic computing,” *Lecture notes in computer science*, p. 597–608, 2004.
[89] P. C. David et T. Ledoux, “WildCAT: a generic framework for context-aware applications,” in *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, p. 7, 2005.
[90] A. Diaconescu, *Composite probes, a monitoring framework for organising data into configurable hierarchies.* France Telecom RD MAPS/AMS, 2007.
[91] “Java Management Extensions (JMX).” [Online]. Available: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/. [Accessed: 11-Sep-2010].
[92] “Eclipse Test & Performance Tools Platform Project.” [Online]. Available: http://www.eclipse.org/tptp/. [Accessed: 11-Sep-2010].
[93] X. Fournier-Morel, P. Grojean, G. Plouin, et C. Rognon, *SOA, le guide de l'architecte*. DUNOD, 2006.
[94] K. Channabasavaiah, K. Holley, et E. Tuggle, *Migrating to a service-oriented architecture*. 2003.
[95] C. Longépé, *Le projet d'urbanisation du système d'information*. DUNOD, 2001.
[96] R. Sessions, “Fuzzy boundaries: objects, components, and web services,” *Queue*, vol. 2, n°. 9, p. 40–47, 2005.
[97] G. Bieber et J. Carpenter, *Introduction to Service-Oriented Programming*. Motorola ISD, 2001.
[98] A. Arsanjani, “Service-oriented modeling and architecture,” 09-Nov-2004. [Online]. Available: http://www.ibm.com/developerworks/library/ws-soa-design1/. [Accessed: 31-Aoû-2010].
[99] S. Chollet, “Orchestration de services hétérogènes et sécurisés.,” Université Joseph Fourier, Grenoble, 2009.
[100] M. Aiello, G. Frankova, et D. Malfatti, “What's in an Agreement? An Analysis and an Extension of WS-Agreement,” *Lecture notes in computer science*, vol. 3826, p. 424, 2005.
[101] H. Cervantes, “Vers un modèle à composants orienté services pour supporter la disponibilité dynamique,” Laboratoire LSR, Université Joseph Fourier, 2004.
[102] J. M. Myerson, “Web service architectures,” *Web Service Business Strategies and Architectures*, 2002.
[103] C. Marin, “Une approche orientée domaine pour la composition de services,” Université Joseph Fourier, Grenoble, 2008.
[104] OASIS, “Security Assertion Markup Language (SAML) V2.0,” 15-Mar-2005. [Online]. Available: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
[105] OASIS, “eXtensible Access Control Markup Language (XACML) Version 2.0,” 01-Fév-2005. .
[106] OASIS, “Security Assertion Markup Language

(SAML) V2.0," 15-Mar-2005. [Online]. Available: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
[107] A. Bottaro et al., "A multi-protocol service-oriented platform for home control applications," in *Demonstration at IEEE Consumer Communications and Networking Conference (CCNC 2007), Las Vegas*, 2007.
[108] J. Estublier et E. Simon, "Universal and Extensible Service-Oriented platform Feasibility and experience: The Service Abstract Machine," 2009.
[109] M. P. Papazoglou, "Extending the Service Oriented Architecture," *Business Integration Journal*, vol. 7, p. 18-21, Fév. 2005.
[110] M. Stal, "Using Architectural Patterns and Blueprints for Service-Oriented Architecture," *IEEE SOFTWARE*, vol. 13, p. 54, 2006.
[111] OASIS, "Web Services Business Process Execution Language." [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
[112] S. Burbeck, *The Tao of e-business services*. 2000.
[113] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, et C. Marin, "A Dynamic-SOA Home Control Gateway," in *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, p. 463–470, 2006.
[114] C. Escoffier, "iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques," Université Joseph Fourier, 2008.
[115] "Main Page - Jini.org." [Online]. Available: http://www.jini.org/wiki/Main_Page. [Accessed: 11-Sep-2010].
[116] UPnP Forum, "UPnP™ Device Architecture 1.1," 15-Oct-2008. [Online]. Available: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.
[117] Microsoft, "Devices Profile for Web Services," Fév-2006. [Online]. Available: http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.
[118] G. Alonso, F. Casati, H. Kuno, et V. Machiraju, *Web services: concepts, architectures and applications*. Springer Verlag, 2004.
[119] W3C, "Canonical XML," 15-Mar-2001. [Online]. Available: http://www.w3.org/TR/2001/REC-xml-c14n-20010315.
[120] W3C, "SOAP Specifications," 27-Avr-2007. [Online]. Available: http://www.w3.org/TR/soap/.
[121] W3C, "Web Service Definition Language (WSDL)," 15-Mar-2001. [Online]. Available: http://www.w3.org/TR/wsdl.
[122] W3C, "UDDI Version 3.0.2," 2004. [Online]. Available: http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm. [Accessed: 11-Sep-2010].
[123] D. Winer, "XML-RPC Specification," 15-Jun-1999. [Online]. Available: http://www.xmlrpc.com/spec. [Accessed: 11-Sep-2010].
[124] W3C, "SOAP Message Transmission Optimization Mechanism," 25-Jan-2005. [Online]. Available: http://www.w3.org/TR/soap12-mtom/.
[125] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Citeseer, 2000.
[126] Open Grid Forum, "Web Services Agreement Specification (WS-Agreement)," 14-Mar-2007. [Online]. Available:

http://www.ogf.org/documents/GFD.107.pdf.
[127] "Web Service Modeling Ontology." [Online]. Available: http://www.wsmo.org/. [Accessed: 11-Sep-2010].
[128] "Web Services Policy 1.2 - Framework (WS-Policy)." [Online]. Available: http://www.w3.org/Submission/WS-Policy/. [Accessed: 11-Sep-2010].
[129] O. Alliance, "OSGi Service Platform Core Specification (Release 4)," 2005.
[130] H. Cervantes et R. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," in *ICSE*, 2004.
[131] C. Escoffier et R. Hall, "Dynamically Adaptable Applications with iPOJO Service Components," in *SC2007*, 2007.
[132] K. K. Lau et Z. Wang, "A taxonomy of software component models," in *Proceedings of the 31st EUROMICRO Conference*, p. 88–95, 2005.
[133] B. Meyer, "The grand challenge of trusted components," in *International Conference on Software Engineering*, vol. 25, p. 660–667, 2003.
[134] G. T. Heineman et W. T. Councill, *Component-based software engineering: putting the pieces together*. Addison-Wesley USA, 2001.
[135] N. Medvidovic et R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Softw. Eng.*, vol. 26, n°. 1, p. 70-93, 2000.
[136] K. Lau et Z. Wang, "A Survey of Software Component Models," *IN SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS. 2005. 31 ST EUROMICRO CONFERENCE: IEEE COMPUTER SOCITY*, 2005.
[137] E. Bruneton, T. Coupaye, et J. B. Stefani, "Recursive and dynamic software composition with sharing," in *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.
[138] R. Monson-Haefel et A. K. Weissinger, *Enterprise JavaBeans*. O'Reilly \& Associates, Inc., 2003.
[139] D. Box, *Essential com*. Addison-Wesley, 1998.
[140] C. Escoffier, R. S. Hall, et P. Lalanda, "iPOJO: An extensible service-oriented component framework," in *IEEE International Conference on Services Computing, 2007. SCC 2007*, p. 474–481, 2007.
[141] "Apache Tuscany." [Online]. Available: http://tuscany.apache.org/. [Accessed: 11-Sep-2010].
[142] BEA, "BEA Aqualogic Service Bus 3.0,," Mar-2008. [Online]. Available: http://www.bea.com/content/news_events/white_papers/BEA_AquaLogic_Service_Bus_ds.pdf.
[143] R. Farha et A. Leon-Garcia, "Blueprint for an autonomic service architecture," in *Proc. of the Intl. Conf. on Autonomic and Autonomous Systems, 2006. ICAS'06*, 2006.
[144] Y. Maurel, A. Diaconescu, et P. Lalanda, "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," in *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, p. 3–11, 2010.
[145] Y. Maurel, A. Diaconescu, et P. Lalanda, "Creating Complex, Adaptable Management Strategies via the Opportunistic Integration of Decentralised Management Resources," in *2009 International Conference on Adaptive and Intelligent Systems*, p. 86–91, 2009.

[146] A. Diaconescu, Y. Maurel, et P. Lalanda, “Autonomic management via dynamic combinations of reusable strategies,” in *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*, p. 1–10, 2008.

[147] P. Lalanda et C. Marin, “A domain-configurable development environment for service-oriented applications,” *IEEE SOFTWARE*, p. 31–38, 2007.