Qinna, une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles

Jean-Charles Tournier

N° d'ordre : 2005ISAL0043

THÈSE

QINNA : UNE ARCHITECTURE À BASE DE COMPOSANTS POUR LA GESTION DE LA QUALITÉ DE SERVICE DANS LES SYSTÈMES EMBARQUÉS MOBILES

Présentée devant

L'Institut National des Sciences Appliquées de Lyon

Pour obtenir

Le grade de docteur

par

Jean-Charles Tournier

Laboratoire universitaire : CITI - INSA Lyon
Laboratoire industriel : France Télécom R&D MAPS/AMS

Soutenue le 1^{er} juillet 2005 devant la commission d'examen

Jury MM.

Jean-Bernard STÉFANI	Directeur de recherche à l'INRIA Rhône-Alpes	Laboratoire SARDES
Jean-Marc GEIB	Professeur à l'université Lille 1	Laboratoire LIFL
Jean-Marc JÉZÉQUEL	Professeur à l'université Rennes 1	Laboratoire IRISA
Vincent OLIVE	Responsable de recherche France Télécom R&D	Laboratoire MAPS/AMS
Stéphane UBÉDA	Professeur à l'INSA Lyon	Laboratoire CITI
Jean-Philippe BABAU	Maître de conférences à l'INSA Lyon	Laboratoire CITI

Remerciements

La rédaction de ce manuscrit, ainsi que le travail effectué pendant ces trois dernières années, doit beaucoup à l'ensemble des personnes qui m'ont entourées ou que j'ai pu côtoyer. Je voudrais tout simplement les en remercier à travers cette page.

Tout d'abord, je souhaite remercier Jean-Bernard Stéfani pour m'avoir fait l'honneur de présider mon jury de thèse. Je souhaite également remercier Jean-Marc Jézéquel et Jean-Marc Geib pour leurs remarques et leurs questions constructives en tant que rapporteurs de ce manuscrit.

Je remercie Anne Mignotte, ainsi que Stéphane Ubéda, en tant que directeurs de thèse successifs et pour m'avoir accueilli au sein du laboratoire CITI de l'INSA de Lyon. Je remercie également les membres du CITI, notamment Ahmad, Karen, Julien, Fabrice, Jean-Louis et Bel Gacem.

Je tiens tout particulièrement à remercier Jean-Philippe Babau pour tout ce qu'il a fait durant ces années. Il a été pour moi un véritable guide au monde de la recherche et la réussite de cette thèse lui doit beaucoup.

Je remercie Kathleen Milsted pour m'avoir accueilli au sein du laboratoire MAPS/AMS de France Télécom R&D et pour m'avoir donné les moyens d'effectuer cette thèse dans les meilleurs conditions.

Je tiens à remercier Vincent Olive pour la confiance qu'il m'a accordée pendant ces trois années. Les encouragements de Jacques Pulou, les remarques de Florence Germain et l'aide précieuse de Jean-Philippe Fassino ont sans aucun doute contribué à l'aboutissement de ce travail. Je tiens également à remercier Marc et Naga pour avoir eu la patience de corriger mon anglais laborieux, mais aussi François pour nos sorties escalades, Juraj pour nos longues discussions montagnes et voyages, Tahar, Yoann, Mickaël, Mourad, et j'en oublie…

Je tiens enfin à remercier ma famille, ainsi qu'Audrey, pour leur soutien, leur compréhension et leur patience. Et pour finir, un grand merci à momo, mick et raph pour tous les moments passés ensemble à parler de tout et surtout de n'importe quoi…

Résumé

Les systèmes embarqués communicants sont de plus en plus présents dans notre environnement quotidien sous formes de PDA, téléphones portables, etc. Ces systèmes se doivent d'être ouverts afin de pouvoir accueillir de nouvelles applications tout au long de leur cycle de vie. Ils possèdent alors des contraintes fortes de types qualité de service, sécurité, tolérance aux fautes, etc. La programmation à base de composants apparaît comme une solution prometteuse pour le développement de tels systèmes. Cependant, un des frein à l'adoption de ce type de programmation est que les modèles à composants n'intègrent pas les aspects de gestion de qualité de service.

Ce travail de thèse présente une architecture de gestion de qualité de service pour les systèmes embarqués mobiles à composants. Cette architecture, appelée Qinna, est définie à l'aide de composants Fractal et permet la mise en oeuvre, ainsi que la gestion dynamique, de contrats de qualité de service entre les différents composants d'un système. L'originalité de l'approche proposée permet de prendre en compte la qualité de service quelque soit le niveau considéré du système (niveau applicatif, niveau services, niveau système d'exploitation et niveau ressources).

L'architecture Qinna a été validée par une évaluation qualitative à base de patrons génériques d'architecture, puis par une évaluation quantitative permettant de montrer que le coût de l'architecture reste faible.

Le travail réalisé ouvre de nombreuses perspectives de recherche notamment celle de généraliser l'approche utilisée (définition d'une architecture abstraite de composant pour la prise en charge de la gestion d'une propriété non-fonctionnelle, ici la QdS) à d'autres propriétés non-fonctionnelles (par exemple la sécurité ou la tolérance aux fautes), et d'en tirer des conclusions sur la définition et la génération de conteneurs ouverts.

Abstract

Component-Based Software Engineering is quickly becoming a mainstream approach to software development. At the same time, there is a massive shift from desktop applications to embedded communicating systems (e.g. PDAs or smartphones) : it is especially the case for multimedia applications such as video players, music players, etc. Moreover, embedded communicating systems have to deal with open aspect : applications may come or leave the system on the fly. A key point of these systems is its ability to rigorously manage Quality of Service due to resource constraints. In this thesis, we present a component-based QoS architecture well-suited for open systems, called Qinna. Qinna is defined using Fractal components and takes into consideration the main QoS concepts (specification, provision and management) through the use of contracts between components. The Qinna architecture allows to deal with QoS of each layer of a system (application, services, operating system, resources) in order to have a fine grain management of resources. The architecture has been first validated thanks to a qualitative evaluation using generic architectural pattern. Then, Qinna has been validated from a quantitative point of view in order to demonstrate its low cost. Finally, this work opens many research perspectives such as the generalization of the architecture to other non functional properties (security, fault tolerance, etc.).

Table des matières

Remerciements **i**

Résumé **ii**

Abstract **iv**

Table des matières **vi**

1 Introduction **1**

2 Etat de l'art **5**

2.1 Programmation par composants 6

2.1.1 Introduction 6

2.1.2 Modèles génériques 7

2.1.2.1 UML2.0 7

2.1.2.2 Fractal 7

2.1.2.3 Conclusion 9

2.1.3 Définitions 9

2.1.3.1 Composant 9

2.1.3.2 Modèle et framework de composants 9

2.1.3.3 Type et classe de composant 9

2.1.3.4 Interface 10

2.1.3.5 Mode de coopération 10

2.1.3.6 Propriété configurable 10

2.1.3.7 Contrat 10

2.1.3.8 Implémentation, paquetage et instance 11

2.1.4 Composants pour les applications distribuées 12

2.1.4.1 Introduction 12

2.1.4.2 Entreprise Java Beans 12

2.1.4.3 .NET 14

2.1.4.4 OSGi 15

2.1.4.5 Conclusion 17

2.1.5 Composants pour les systèmes embarqués 17

2.1.5.1 Introduction 17

2.1.5.2 Koala 18

2.1.5.3 PECOS 19

2.1.5.4 Think . . . 21
2.1.6 Conclusion . . . 22
2.2 Architectures de gestion de QdS . . . 23
2.2.1 Définitions . . . 24
2.2.1.1 Architecture de gestion de QdS . . . 24
2.2.1.2 Caractéristiques et contraintes de QdS . . . 25
2.2.1.3 Spectre de QdS . . . 25
2.2.1.4 Contrat de QdS . . . 25
2.2.2 Exemples d'architectures de gestion de QdS . . . 27
2.2.2.1 L'architecture de l'ISO . . . 27
2.2.2.2 Architecture de gestion de QdS de bout en bout . . . 28
2.2.2.3 Architecture de gestion de QdS pour les systèmes à composants . . . 29
2.2.3 Conclusion . . . 30
2.3 Motivations pour une nouvelle architecture de gestion de QdS . . . 30

3 Qinna **33**
3.1 Introduction . . . 33
3.1.1 Objectifs . . . 33
3.1.2 Hypothèses . . . 34
3.1.3 Principes généraux de Qinna . . . 34
3.1.4 Vue globale de Qinna . . . 35
3.1.5 Convention graphique . . . 36
3.2 Définition de l'architecture . . . 37
3.2.1 Interfaces et types de composants . . . 37
3.2.1.1 Interfaces Qinna . . . 37
3.2.1.2 QoSComponent . . . 37
3.2.1.3 QoSComponentBroker . . . 37
3.2.1.4 QoSComponentManager . . . 39
3.2.1.5 QoSComponentObserver . . . 41
3.2.1.6 QoSDomain . . . 41
3.2.2 Types de données . . . 42
3.2.3 Comportement dynamique . . . 43
3.2.3.1 Mise en place de contrat . . . 43
3.2.3.2 Adaptation de contrat . . . 45
3.2.3.3 Mise à jour dynamique des données d'un contrat . . . 46
3.3 Mise en place de Qinna . . . 49
3.3.1 Principes . . . 49
3.3.2 Illustration . . . 52
3.3.2.1 Description du système initial . . . 52
3.3.2.2 Mise en place de Qinna . . . 54
3.4 Conclusion . . . 59

4 Illustrations, expérimentations et évaluations de l'architecture Qinna **61**
4.1 Utilisation de ressources . . . 61
4.1.1 Un composant utilise *n* ressources différentes . . . 61
4.1.1.1 Présentation . . . 61

4.1.1.2 Intégration de Qinna . . . 62
4.1.1.3 Illustration . . . 63
4.1.1.4 Analyse . . . 64
4.1.2 Deux composants partagent les services d'un composant . . . 64
4.1.2.1 Présentation . . . 64
4.1.2.2 Partage d'un composant sécable . . . 65
4.1.2.3 Partage d'un composant insécable . . . 68
4.1.3 Hétérogénéité d'expression de QdS . . . 74
4.1.3.1 Problématique . . . 74
4.1.3.2 Présentation . . . 74
4.1.3.3 Prise en compte par Qinna . . . 74
4.1.3.4 Illustrations . . . 75
4.1.3.5 Analyse . . . 76
4.2 Comportement dynamique . . . 76
4.2.1 Coût de gestion des contrats de QdS . . . 76
4.2.1.1 Problématique . . . 76
4.2.1.2 Présentation . . . 77
4.2.1.3 Actions menées par Qinna . . . 77
4.2.1.4 Illustrations . . . 79
4.2.1.5 Analyse . . . 80
4.2.2 Profil variable de QdS requise . . . 81
4.2.2.1 Problématique . . . 81
4.2.2.2 Présentation . . . 81
4.2.2.3 Prise en compte par Qinna . . . 81
4.2.2.4 Illustration . . . 84
4.2.2.5 Analyse . . . 84
4.2.3 Modification de la capacité d'une ressource . . . 85
4.2.3.1 Problématique . . . 85
4.2.3.2 Présentation . . . 86
4.2.3.3 Prise en compte par Qinna . . . 86
4.2.3.4 Illustration . . . 87
4.2.3.5 Analyse . . . 88
4.2.4 Évolution de la relation d'ordre . . . 89
4.2.4.1 Problématique . . . 89
4.2.4.2 Présentation . . . 89
4.2.4.3 Prise en compte par Qinna . . . 89
4.2.4.4 Illustration . . . 91
4.2.4.5 Analyse . . . 92
4.3 Evaluations quantitatives . . . 92
4.3.1 Évaluations de la mise en place et de l'adaptation de contrats . . . 93
4.3.1.1 Présentation . . . 93
4.3.1.2 Mesures . . . 94
4.3.2 Évaluations quantitatives de la maintenabilité d'un contrat . . . 95
4.3.2.1 Présentation . . . 95
4.3.2.2 Mesures . . . 95

4.3.3 Analyse 96
4.4 Conclusion 97

5 Conclusion et perspectives **103**

Bibliographie **106**

Table des figures **111**

Liste des tableaux **115**

Chapitre 1

Introduction

Cette thèse s'inscrit dans le contexte de l'émergence des systèmes embarqués mobiles (assistants personnels, téléphones portables, lecteurs MP3, etc.) dans notre environnement quotidien. Ce type de système est soumis à de nombreuses contraintes. Il s'agit aussi bien de contraintes liées aux coûts de production (impliquant l'utilisation de matériels spécifiques à bas coûts), que de contraintes ergonomiques (impliquant une taille réduite des systèmes), en passant par des contraintes de mobilité (impliquant une alimentation autonome) et temporelles (liées aux exigences des utilisateurs vis-à-vis des applications multimédia supportées et à l'utilisation d'un médium de communication partagé). L'ensemble de ces contraintes conduisent ces systèmes à disposer de ressources matérielles limitées et spécifiques (CPU, mémoire, réseau, batterie). De plus, l'interface homme machine est limitée et offre des possibilités d'administration réduites, amenant ainsi ces systèmes à posséder des contraintes fortes de fiabilité. En conséquence, le développement de tels systèmes amène à s'assurer d'une gestion efficace des ressources. Ces caractéristiques font qu'à l'heure actuelle les systèmes embarqués mobiles sont construits de façon dédiée aussi bien au niveau matériel (DSP, SOC), que logiciel (systèmes d'exploitation, applications). Cette approche aboutit à des systèmes monolithiques et non flexibles. Or, le modèle économique actuel de développement fait intervenir différents acteurs tout au long du cycle de vie du système. Il apparaît alors un besoin d'ouverture et de flexibilité permettant ainsi leur évolution via l'installation, ou la désinstallation, dynamique d'applications ou de services. Ce besoin d'ouverture, couplé aux contraintes de ce type de systèmes (ressources limitées et spécifiques, besoin de fiabilité), font qu'il est nécessaire de pouvoir disposer d'un support permettant la gestion, à la fois, dynamique et sûre des ressources utilisées par le système.

La programmation à base de composants apparaît comme une solution prometteuse pour le développement de tels systèmes. En effet, ce paradigme de programmation permet de construire et de structurer un système comme un assemblage de briques logicielles dont les dépendances sont clairement identifiées, se différenciant ainsi du paradigme objet. Le système résultant possède alors les propriétés de réutilisabilité (réutilisation de composants), de partitionnement (découpage en composants) et d'adaptation (configuration des composants). De plus, la possibilité d'expliciter les dépendances requises permet de faciliter le changement d'un composant par un autre, notamment à l'exécution. Il existe aujourd'hui de nombreux modèles de composants couvrant un large spectre de domaines d'applications qui va, des applications distribuées (EJB [72], CCM [75], DCOM [32], .NET [68], OSGi [34]), jusqu'aux systèmes embarqués fortement contraints (PECOS [80], VEST [86], Koala [95], Fractal/Think [29], Rubus [48]). Les principaux freins à l'adoption de ce paradigme de programmation dans les systèmes embarqués sont liés soit à l'absence de gestion des ressources du système, soit à une gestion spécifique et pré-déterminée de certaines

propriétés. Ces modèles, mis à part Fractal, sont non-extensibles et ne permettent pas d'identifier les composants lors de l'exécution du système se privant ainsi des apports de composants à l'exécution (évolution, adaptation). D'un autre coté, il existe de nombreuses architectures de gestion de QdS (QoSA [17], TINA [2], OMEGA [74], MASI [56], End-to-End QoS Framework [35], QuA [5], [84], [24], [65], [25], [58], [46] et [3]). Cependant, ces architectures ne permettent pas de tirer profit des avantages de la programmation par composant et ne sont pas adaptées aux caractéristiques des systèmes embarqués.

L'objectif de cette thèse est de contribuer au développement d'un support architectural de gestion des contraintes de qualité de service (QdS) dans les systèmes embarqués ouverts à composants. Pour cela, nous proposons de définir une architecture à base de composants permettant la gestion de la QdS des composants du système. Le type de QdS visé par ce travail est la QdS liée à l'utilisation des ressources matérielles sous-jacentes (CPU, mémoire, réseau et batterie).

Une architecture de gestion de la QdS dans les systèmes embarqués à composants doit considérer un ensemble de propriétés liées à la nature *ouverte* des systèmes visés. Plus précisément, dans ce travail nous considérons les propriétés suivantes :

1. **la généricité :** il existe de nombreuses politiques de gestion de la QdS dans les systèmes embarqués. Il est donc nécessaire de pouvoir disposer d'un support architectural suffisamment générique afin de permettre l'intégration de diverses politiques de gestion de la QdS.
2. **l'hétérogénéité :** les composants d'un système ouvert peuvent avoir été développés par des équipes différentes, ce qui implique que leurs contraintes QdS puissent être spécifiées dans des langages de QdS différents. Il est donc nécessaire de disposer de mécanismes permettant l'utilisation de langages de spécification de QdS hétérogènes.
3. **la dynamicité :** les systèmes visés évoluent dynamiquement tout au long de leur cycle de vie. En effet, les composants peuvent être chargés/déchargés ou activés/désactivés lors de l'exécution du système. Il est donc nécessaire que la gestion de la QdS des différents composants s'effectue dynamiquement et parallèlement à l'évolution structurelle du système.
4. **l'auto-configurabilité :** les besoins en QdS de chacun des composants du système doivent pouvoir être déterminés dynamiquement. En effet, les composants ont pour vocation d'être réutilisés sur différents systèmes. Cela implique qu'un composant peut ne pas connaître le niveau de QdS requis pour une plateforme donnée. Deux cas sont alors possibles : soit le composant ne connaît pas ses besoins en QdS, soit le composant en a une estimation (réalisée par le biais d'expérimentations sur des systèmes précédents ou par le développeur).
5. **la confiance :** le support de gestion de QdS ne doit pas se baser uniquement sur les déclarations d'intention de QdS des différents composants, mais doit mettre en oeuvre des mécanismes permettant de contrôler qu'un composant ne consomme pas une QdS supérieure à celle spécifiée.
6. **la réutilisabilité :** le support de gestion de QdS doit être réutilisable, afin de réduire les temps de développement des systèmes. En effet, étant donné que les composants ont pour vocation d'être réutilisables, il est nécessaire que les aspects de gestion de QdS qui leurs sont liés le soient aussi.

Avant de présenter la définition de l'architecture de gestion de QdS considérant les problématiques énoncées précédemment, le chapitre 2 présente l'état de l'art des modèles de composants et des architectures de gestion de QdS. Il permet de présenter les principaux concepts et définitions de chacun des deux domaines utilisés par l'architecture proposée dans cette thèse. Puis, les motivations pour la définition d'une nouvelle architecture sont

présentées, en soulignant aussi bien les apports que les manques des approches existantes pour chacune des problématiques fixées pour ce travail.

Ensuite, le chapitre 3 présente l'architecture de gestion de QdS proposée par cette thèse, appelée Qinna. Les principes généraux guidant la définition de l'architecture sont tout d'abord précisés, puis les définitions des différents éléments la constituant sont présentées. Le chapitre se poursuit par la présentation d'une méthodologie permettant l'intégration de l'architecture dans un système à composants. Enfin, une dernière partie détaille les réponses apportées par Qinna aux problématiques de gestion de QdS soulevées par les systèmes ouverts.

Enfin, le chapitre 4 est consacré à l'évaluation de l'architecture au travers d'illustrations et d'expérimentations sur des cas types présents dans les systèmes embarqués ouverts. Ce chapitre permet d'évaluer l'architecture proposée afin d'identifier et de quantifier ses apports et ses limites.

Pour finir, le chapitre 5 donne les conclusions, ainsi que des perspectives pour les travaux menés durant cette thèse.

Chapitre 2

Etat de l'art

Cette thèse est liée à deux domaines de recherche que sont la programmation par composants et les architectures de gestion de qualité de service. Ce chapitre présente les principaux concepts, définitions et travaux de chacun de ces deux domaines. Puis nous concluons en présentant les motivations pour la définition d'une nouvelle architecture de gestion de qualité de service pour les systèmes embarqués ouverts à composants.

Cependant, avant d'étudier les modèles de composants et les architectures de gestion de QdS, il est nécessaire de définir le terme de *qualité de service*. Il existe de nombreuses définitions de la qualité de service relatives, en particulier, à l'utilisation du réseau. Dans le cadre de cette thèse nous nous appuyons sur les définitions plus générales proposées par l'ISO [46] (International Organization for Standardization) et l'ITU-T [92] (International Telecommunication Union).

L'ISO définit la QdS comme étant :

> *«a set of qualities related to the collective behavior of one or more objects. »*

et l'ITU-T comme :

> *«QoS has been defined as a collective effect of service and performances that determine the degre of satisfaction of the service. »*

Le degré de satisfaction est quantifié à travers des exigences exprimées sur les caractéristiques du système tel que le débit, le temps de réponse, la sûreté, la fiabilité, etc. On distingue généralement la QdS dépendante des ressources utilisées (machines, réseaux, énergie, mémoire), de la QdS liée à l'ajout de fonctions supplémentaires telle que la sécurité, la persistance, la tolérance aux fautes, etc. Dans la suite de cette étude, nous nous focalisons plus particulièrement sur la QdS liée aux ressources.

2.1 Programmation par composants

2.1.1 Introduction

L'origine de la programmation par composants ne fait pas l'unanimité. En effet, certains tels que [41] pensent que l'utilisation de composants logiciels n'est pas une idée nouvelle et que nous avons toujours eu à faire à des composants de différentes sortes. D'autres, tels que [97], situent l'utilisation des composants à la fin des années 1970 comme étant une réponse à la deuxième crise du logiciel provoquée par l'introduction à grande échelle des microprocesseurs. Il a fallu alors produire des logiciels de meilleure qualité et de façon plus rapide et moins coûteuse. A cette époque, une nouvelle discipline du génie logiciel émerge : le *Component-Based Software Engineering* (CBSE). Elle est portée notamment par un centre de recherche créé par le gouvernement fédéral américain et exclusivement dédié au génie logiciel : le *Software Engineering Institute*.

Les bénéfices attendus de l'utilisation des composants sont a priori clairs : il s'agit essentiellement de réduire les coûts de production des logiciels, mais aussi d'en réduire les coûts de maintenance et d'en favoriser la réutilisation [13]. Par exemple, l'analyse effectuée pour le compte du ministère de la défense des États-Unis, rapportée dans [62], laisse arguer une réduction de l'ordre de 47% de travail en moins liée à la réutilisation de composants. De plus, les composants doivent permettre une meilleure structuration des systèmes facilitant ainsi les aspects de vérification et de validation [23].

Il existe aujourd'hui de nombreux modèles à composants couvrant un large spectre de domaine d'applications. Les développements actuels portent essentiellement sur les applications distribuées (COM [67], .NET [68], EJB [72], CCM [75], OSGi [34]). Mais, il existe, aussi, de plus en plus de modèles dédiés aux systèmes embarqués (PECOS [80], VEST [86], Koala [95], Think [29], Rubus [48]).

Afin d'introduire les concepts généraux de la programmation par composants, nous présentons tout d'abord deux modèles de composants génériques (UML2.0 [38] et Fractal [16]). Puis, à l'aide de la terminologie issue de ces deux modèles, ainsi que des synthèses existantes [21, 79, 8, 23], nous fixons les termes relatifs aux composants employés dans ce rapport. Nous présentons ensuite les principales caractéristiques des modèles ayant pour cible les applications distribuées. Bien que ces modèles ne permettent pas de prendre en compte la QdS liée aux ressources, plusieurs raisons motivent leur étude. Premièrement, ils permettent de prendre en compte certaines propriétés de QdS (par exemple la persistance ou la sécurité). Deuxièmement, ils possèdent les caractéristiques des systèmes ouverts en permettant d'ajouter, ou de retirer, dynamiquement les composants. Et troisièmement, ces modèles font l'objet de nombreuses implémentations et sont largement utilisés. Suite à l'étude de ces modèles, les principales caractéristiques des modèles à composants visant les systèmes embarqués sont présentées. Au cours de cette étude, nous nous intéressons plus particulièrement à la façon dont la QdS liée aux ressources est prise en compte par ces modèles.

2.1.2 Modèles génériques

2.1.2.1 UML2.0

Au travers d'UML2.0 [38], L'Object Management Group (OMG) a proposé, en 2003, un premier standard permettant la modélisation de systèmes à base de composants aussi bien logiciels que matériels.

Cette nouvelle version du langage de modélisation UML introduit de nouveaux concepts hérités de différents domaines de recherche, depuis la conception orientée composants de plateformes logicielles à la modélisation d'architectures opérationnelles de systèmes.

Un composant UML possède deux vues complémentaires : une vue de type boîte blanche et une vue de type boîte noire. La vue boîte noire d'un composant est caractérisée par ses interfaces fournies et requises (cf. figure 2.1). Les interfaces fournies permettent d'accéder aux services fournis par le composant, tandis que les interfaces requises modélisent les dépendances externes du composant. Un composant est alors vu comme une unité modulaire, définie par ses interfaces avec son environnement. Un composant peut servir de type, défini par l'ensemble de ses interfaces. Ainsi, deux composants peuvent être interchangeables dans le cas où leurs types sont conformes au même modèle d'interfaces. De plus, les interfaces peuvent être regroupées au sein de ports, permettant ainsi de regrouper les interfaces relatives à une préoccupation donnée (par exemple, les interfaces d'administration, de configuration, etc.). Les ports peuvent posséder à la fois des interfaces requises et fournies. Les composants sont reliés entre eux par des *connecteurs d'assemblage* (appelés *assembly connectors*). Un connecteur d'assemblage relie une interface requise d'un composant à une interface fournie d'un autre composant.

La vue boîte blanche, elle, est cachée à l'utilisateur du composant. Cette vue s'appuie sur un ensemble de *realizations* implémentant le comportement du composant (cf. figure 2.1). La liaison entre une interface et sa *realization* est effectuée par un connecteur de délégation (appelé *delegation connector*).

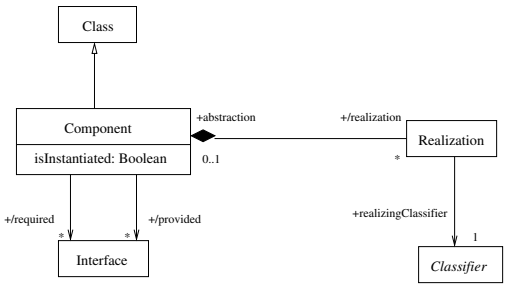

FIG. 2.1 – Méta modèle d'un composant UML2.0.

Enfin, à la phase de déploiement, un composant UML est représenté par un ou plusieurs *artifacts*. Un *artifact* est alors déployable, ou re-déployable, indépendamment d'un environnement d'exécution donné.

2.1.2.2 Fractal

Le modèle de composants Fractal [16], dont la première spécification est parue en 2002, a été conjointement proposé par France Télécom R&D et l'INRIA Rhône-Alpes.

Le modèle de composants Fractal se base sur six concepts clés : le composant, le contrôleur, le contenu, l'interface, le service et la liaison.

Un composant Fractal est formé d'un contrôleur et d'un contenu. Un composant est clairement identifié tout au long de son cycle de vie allant de la phase de développement à l'exécution. A l'exécution, un composant est identifié par son identificateur.

Le contenu d'un composant est composé d'un nombre fini de composants, appelé sous-composants. Ces sous-composants sont sous le contrôle du composant englobant. Le modèle de composants Fractal est récursif grâce à cette notion de sous-composant, mais la récursivité s'arrête au niveau d'un composant primitif. Un composant primitif est défini comme un composant n'étant pas composé de sous composants.

Le contrôleur d'un composant est destiné à contrôler le contenu du composant. Le contrôleur a la capacité d'intercepter les appels entrants et sortants d'un composant, de donner une représentation interne du contenu du composant (un composant est alors réflexif) ou encore de suspendre ou d'arrêter l'activité d'un sous composant.

Un composant Fractal interagit avec son environnement par le biais de services via des points d'accès appelés interfaces. Les interfaces peuvent être de deux types : client ou serveur. Une interface serveur peut recevoir des invocations de services alors qu'une interface client en émet.

Une liaison est une connexion entre deux ou plusieurs composants. Une liaison peut être établie entre une interface client et une interface serveur uniquement si l'interface serveur accepte tous les services que peut émettre l'interface client, et si l'interface client peut accepter tous les résultats retournés par le service. En d'autres termes, l'interface serveur doit être un sous-type de l'interface client.

Parallèlement au modèle de composants, Fractal définit un langage de description d'architecture [14] (ADL Fractal) permettant de définir l'architecture structurelle d'un système. Plus précisément, cet ADL permet de décrire les composants (en termes de contenu, de contrôleur et d'interfaces) ainsi que leurs liaisons. De plus, l'ADL Fractal est modulaire et extensible offrant ainsi la possibilité d'être étendu afin de spécifier, par exemple, des contraintes de déploiement ou de comportement.

La figure 2.2 fournit une représentation graphique d'un composant Fractal `CompX` fournissant une interface `iX` et requérant une interface `iY`. Dans cet exemple, l'interface `iY` est fournie par le composant `CompY`. D'un point de vue fonctionnel, le composant `CompX` fournit `iX` si, et seulement si, un composant lui fournit `iY`.

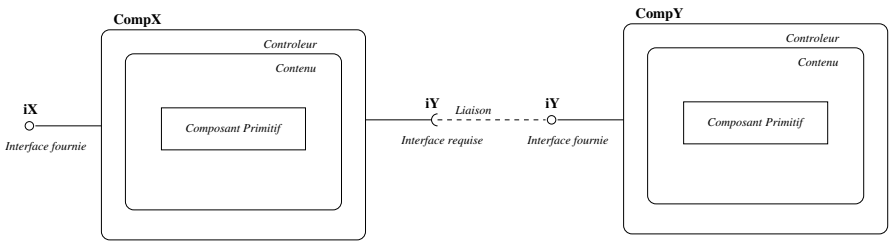

FIG. 2.2 – Représentation graphique de composition de composants Fractal.

2.1.2.3 Conclusion

Les deux modèles présentés dans cette première partie sont des modèles génériques. En effet, aucun des deux modèles ne fait d'hypothèses liées à un domaine d'application particulier. Ces modèles nous permettent d'identifier les notions communes aux modèles de composants et d'en déterminer la terminologie associée que nous présentons dans la suite de cette partie.

2.1.3 Définitions

En se basant sur la terminologie et les définitions des deux modèles présentés précédemment, mais aussi à partir des synthèses existantes sur les modèles à composants [21, 79, 8, 23], nous fixons les termes employés dans ce rapport relatifs aux composants.

2.1.3.1 Composant

Il n'existe pas, à l'heure actuelle, de définition communément admise du terme composant. Cependant, dans le cadre de cette thèse, nous utiliserons la définition largement répandue et proposée par Clemens Szyperski dans [88] :

> *A component is an unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition.*

Plus précisément, nous considérons qu'un composant est une entité logicielle exprimant clairement ses dépendances et les services qu'elle fournit. Idéalement, toutes les dépendances de contexte doivent être capturées par les interfaces du composant. Ainsi, un composant peut être composé et déployé indépendamment dans un système.

2.1.3.2 Modèle et framework de composants

Un modèle de composants permet de spécifier les standards et les conventions suivis par les composants, alors qu'un framework à composants est l'infrastructure logicielle permettant l'exécution des composants conformes à un modèle [8]. Pour un même modèle, il peut exister plusieurs frameworks dépendants par exemple du domaine d'application. A titre d'exemple, Fractal est un modèle alors que Julia [15] (implémentation du modèle Fractal en Java) et Think [29] (implémentation du modèle Fractal en C) en sont des frameworks.

2.1.3.3 Type et classe de composant

Un type de composant [16, 38] est caractérisé par trois éléments : ses interfaces, ses modes de coopération et ses propriétés configurables, alors qu'une classe de composant [16] est définie par son type et son implémentation.

2.1.3.4 Interface

Les interfaces peuvent être de deux types [16, 38] :

- les interfaces *fournies* par le composant sont du même ordre que les interfaces des objets : elles définissent les services fournis par le composant, en listant les signatures des services fournis.
- les interfaces *requises* représentent un progrès par rapport à l'approche objet. Dans le cas des objets, une référence sur un objet utilisé est enfouie au coeur du code. Dans le cas des composants, les interfaces utilisées sont exprimées au niveau du type de composant. Il est ainsi plus aisé d'une part, de substituer un composant par un autre et d'autre part, de gérer les connexions entre les composants, i.e. de connaître les dépendances pour l'installation et le remplacement de composants. Les interfaces requises sont définies par les signatures des services requis.

De plus, la définition d'une interface peut ne pas se limiter aux signatures de services : en effet, il est possible d'enrichir une interface par l'ajout de spécifications de propriétés fonctionnelles ou non-fonctionnelles. L'ajout de ces propriétés permet alors de faciliter la vérification et la prédiction de propriétés d'assemblages de composants.

Il est important qu'une interface soit définie de manière indépendante à toute implémentation. Il est ainsi possible, d'une part, de fournir plusieurs implémentations pour un même type de composant et, d'autre part, de pouvoir substituer une implémentation par une autre implémentation du même type.

2.1.3.5 Mode de coopération

Pour chacune des interfaces d'un composant, il est nécessaire de spécifier le mode de coopération. Il existe trois modes de coopération : le mode synchrone (par exemple l'invocation de méthode), le mode asynchrone (par exemple l'envoi d'un message) et le mode diffusion en continu (par exemple la communication par flots de données).

2.1.3.6 Propriété configurable

Le dernier élément caractérisant un type de composant est l'ensemble de ses propriétés configurables [16]. Ces propriétés permettent d'adapter une instance de composant en configurant son comportement. Le code du composant n'est pas modifié en fonction du besoin, mais paramétré, augmentant ainsi la réutilisabilité potentielle du composant.

2.1.3.7 Contrat

La sémantique de l'assemblage de deux composants est un point clé. Une vision aujourd'hui largement partagée de la composition se fonde sur la notion de contrat entre les deux composants à assembler. On distingue quatre niveaux distincts de contrats, organisés en une hiérarchie suivant leur degré de négociabilité [11] :

- Les contrats de base (niveau 1) spécifient la signature des services offerts ou requis par un composant. Ce sont des contrats fonctionnels, non négociables. Ces contrats sont vérifiés statiquement à la compilation, ou

dynamiquement via l'usage d'un intergiciel. IDL [47] est un exemple de langage de spécification pour ce type de contrat.
– Le deuxième niveau permet de spécifier le comportement des services définis précédemment. Cette spécification se formalise généralement sous la forme d'assertions booléennes appelées pré- et post-conditions. A l'invocation d'un service par un client auprès d'un fournisseur, le client s'engage à respecter la pré-condition. La post-condition engage contractuellement le fournisseur sur le résultat du service invoqué. Ce niveau, connu de la communauté sous le nom de *design by contract*, est présent aussi bien en modélisation (OCL [99], TLA [57] ou Unity [20]) qu'en programmation (Eiffel [66], Java Modeling Language [59]).
– Le troisième niveau concerne la synchronisation entre composants. Les contrats de ce niveau expriment des contraintes d'interactions dans des contextes concurrents. Leur analyse prévient l'apparition d'éventuels problèmes tels que les interblocages. Les contrats de synchronisation ont une importance particulière au sein de la communauté des composants temps réel. Ce niveau doit être spécifié à l'aide de langages possédant des opérateurs de composition tels que SDL [27] ou UML [39], ou à l'aide d'algèbre de processus.
– Enfin, le dernier niveau concerne les aspects dits *extra fonctionnels* des composants. Ils expriment le niveau de qualité de service (QdS) qu'on souhaite observer sur une interface donnée. Ils feront l'objet de l'étude qui suit.

2.1.3.8 Implémentation, paquetage et instance

L'implémentation d'un composant, aussi appelée *arteface* dans UML2.0, est la réalisation exécutable d'un type de composant. L'implémentation d'un composant peut être fournie sous forme de code C ou Java, de byte code, de binaire, etc.

Un paquetage de composant est l'entité diffusable et déployable, bien souvent une archive, contenant au minimum le type de composant et une ou plusieurs implémentations. Ce principe est identifié par l'*artifact* dans UML2.0.

Enfin, une instance de composant est, au même titre qu'une instance d'objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implantation particulière de ce type. De la même manière que pour une instance d'objet, une instance de composant peut recevoir des requêtes de service. De plus, afin de rendre les services fournis, une instance doit être liée à d'autres instances de composants via ses interfaces requises.

Cette première partie a permis de présenter et de définir les principaux concepts de la programmation par composants. Nous allons maintenant étudier les caractéristiques des modèles dédiés à la programmation des applications distribuées, puis des systèmes embarqués, en se focalisant plus particulièrement sur les aspects de gestion de QdS.

2.1.4 Composants pour les applications distribuées

2.1.4.1 Introduction

Les modèles de composants les plus utilisés à l'heure actuelle sont ceux visant la construction d'applications distribuées. Ces modèles (EJB [72], CCM [75], .NET [68], JavaBean [96], COM/DCOM/COM+ [67, 32, 36], OSGi [34]) ciblent principalement le développement d'applications distribuées où la quantité de ressources disponibles sur les plateformes d'exécution n'est pas un critère critique : les développeurs peuvent alors adopter un style de programmation résolument *optimiste* en se souciant peu des ressources nécessaires au fonctionnement des applications. Les principales caractéristiques de ces modèles, à l'exception de CCM, sont leurs liaisons à un langage de programmation (par exemple le langage Java pour les modèles EJB et OSGi) et/ou à un système d'exploitation particulier (par exemple Windows pour les modèles COM, DCOM et COM+). De plus, une autre caractéristique commune à ces modèles est l'utilisation récurrente de méta-données permettant aussi bien de décrire la structure interne d'un composant que l'assemblage de plusieurs composants.

Du point de vue de la gestion de la QdS, il existe deux classes de modèles : soit la QdS n'est pas prise en compte (OSGi [34], JavaBean [96], COM/COM+/DCOM [67, 36, 32]), soit seuls quelques aspects le sont (EJB [72], CCM [75], .NET [68]). Ces aspects sont évidemment ceux inhérents aux applications distribuées tels que la sécurité, la persistance, les transactions et le nommage. Il est intéressant de noter qu'aucun de ces modèles ne se préoccupe de la gestion de la QdS liée à l'utilisation des ressources (mémoire, CPU, réseau, etc.), ce qui s'explique par le fait que les applications visées ne s'exécutent pas sur des systèmes fortement contraints.

Il faut aussi noter que deux principales approches de gestion de QdS sont mises en oeuvre : soit les appels aux mécanismes de QdS sont explicités directement dans le code applicatif (.NET), soit les modèles utilisent une approche par conteneurs (EJB, CCM). Parallèlement, ces modèles ont comme intérêt de se focaliser sur les aspects de déploiement de composants, ainsi que sur les aspects liés à l'évolution dynamique (en cours d'exécution) des applications.

Afin d'étudier ce type de modèle plus en détails, nous en présentons ci-après trois modèles caractéristiques, qui sont les modèles EJB, .NET et OSGi. L'intérêt du modèle EJB réside dans l'architecture proposée (approche par conteneurs) permettant de gérer la QdS d'une application. Le second modèle, .NET, propose une réponse à la problématique de l'évolution des composants (versions des composants), alors que le modèle OSGi propose une approche adaptée au déploiement dynamique des applications. Pour chacun des modèles, les grands principes sont présentés puis une analyse est effectuée.

2.1.4.2 Entreprise Java Beans

2.1.4.2.1 Présentation Le modèle de composants Entreprise Java Beans (EJB) [72], dont la première spécification est parue en 1998, a été développé par Sun Microsystems®. afin de proposer un modèle de composants pour la construction d'applications réparties côté serveur dans le cadre des architectures n-tiers. Une telle application est constituée de composants appelés *Beans* implémentés en Java.

Les Beans peuvent être de trois types : *session*, *entité* et *à messages*. Au sein des applications, les Beans de type session et entité sont accessibles à travers une communication synchrone basée sur Java RMI [70]. Les Beans à

messages, introduits par la dernière version de la spécification, sont un moyen d'intégrer un service d'événements dans les EJB. Ils communiquent de manière asynchrone en utilisant la technologie JMS [71].

Les instances de Beans s'exécutent dans des conteneurs EJB qui sont chargés de leurs aspects de QdS (cf. figure 2.3). Les conteneurs sont inclus, eux-mêmes, dans des environnements d'exécution, des serveurs EJB (J2EE), qui fournissent les services systèmes nécessaires. Le modèle EJB ne prévoit que la gestion des aspects de QdS suivants : persistance, transaction, sécurité et concurrence. Le modèle définit de manière précise les liens entre ces aspects, la manière dont ils sont intégrés dans les conteneurs et l'ordre de leur traitement. Les conteneurs peuvent héberger plusieurs instances de plusieurs types de Beans et gèrent la QdS selon les types des composants.

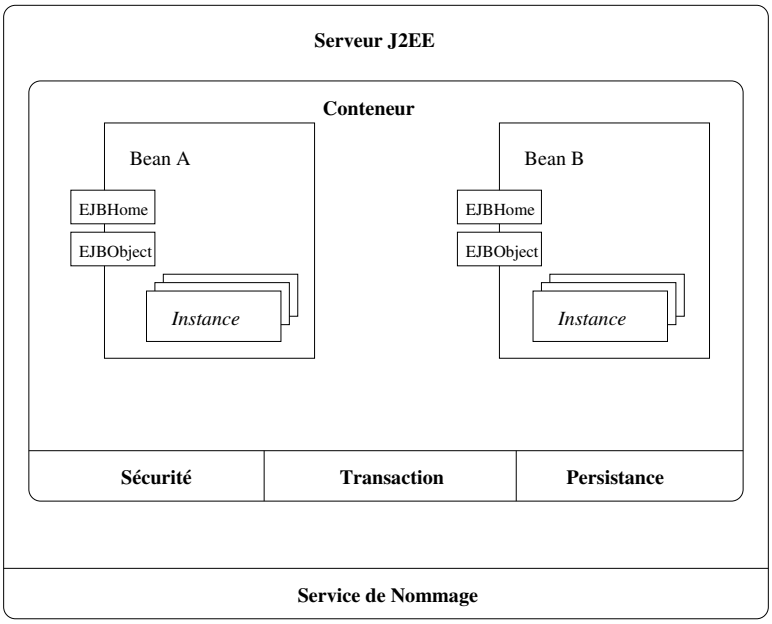

FIG. 2.3 – Architecture d'exécution des EJB.

Pour les Beans session, qui représentent un dialogue avec des clients et qui sont non persistants et non concurrents, les conteneurs ne gèrent que l'état de ce dialogue. Pour les Beans à messages, définis en tant que composant sans état pouvant être partagé entre différents clients, les conteneurs gèrent la concurrence. Pour les Beans entité, représentant des données, les conteneurs gèrent la persistance. Les conteneurs fournissent une gestion de la persistance par défaut, mais offrent aux Beans la possibilité de la gérer eux-mêmes. Pour tous les types de Beans, les conteneurs fournissent la sécurité et les transactions.

Afin de gérer les propriétés de QdS, le conteneur est constitué d'objets d'interposition qui interceptent les invocations sur les Beans et effectuent des traitements avant et après ces invocations. Les pré-traitements peuvent inclure la vérification des droits d'accès qui est propre à la gestion de la sécurité, le démarrage des transactions ou la propagation des contextes transactionnels. Les post-traitements se chargent de l'enregistrement éventuel des Beans sur un support persistant et de la terminaison des transactions démarrées par un pré-traitement.

Dans le cas des composants entité ou session, les conteneurs fournissent deux objets aux beans : un *EJB Home* et un *EJB Object*. Les EJB Homes sont des usines à composants permettant la création, la recherche et la destruction d'instances, et sont rattachés à un type de Beans. Les EJB Objects sont des objets d'interposition rattachés aux instances d'un type de Beans et représentent le seul moyen pour accéder aux composants encapsulés

à l'intérieur des conteneurs. Ce sont eux qui contiennent, en plus des fonctions d'emballage et de déballage des messages distants, les pré et post-traitements nécessaires à la gestion des aspects non fonctionnels.

Dans le cas des composants à messages, dont les instances n'ont pas d'état ni d'identité, les conteneurs fournissent un objet d'interposition qui définit un objet de destination. Les messages des clients sont alors adressés à cet objet qui délègue au conteneur l'envoi d'un accusé de réception et la transmission des messages à une des instances des composants à messages.

Lors de la phase de déploiement un Bean est packagé sous forme d'un fichier archive *.jar*. Ce fichier comprend à la fois le code du Bean et un ou des fichiers de déploiement. Un fichier de déploiement contient deux types d'informations : (a) les informations structurelles décrivant l'implémentation d'un Bean : ces informations sont le nom du Bean, son type, ses interfaces fournies/requises et ses classes d'implémentation ; (b) ses dépendances externes telles que les services de QdS (persistance, transaction et gestion du cycle de vie).

Enfin, il existe de nombreux projets implémentant la spécification EJB. Il s'agit aussi bien de logiciels en source libre (Jonas [52], JBoss [51]), que des logiciels commerciaux (DEA WebLogic [87], IBM WebSphere [100] ou Oracle9iAS [76]).

2.1.4.2.2 Analyse Par rapport aux définitions présentées au paragraphe 2.1.3, le modèle EJB propose la notion de composant déployable et réutilisable. Il propose aussi la notion d'interface : il existe des interfaces prédéfinies et obligatoires (telles que EJBHome et EJBObject) et le développeur peut en définir de nouvelles. De plus, les communications entre les composants peuvent être soit synchrones, soit asynchrones (dépendant du type de Bean). Enfin, il existe la notion de paquetages (fichier .jar) et d'instances de composants s'exécutant sur un serveur.

Ce modèle a l'avantage d'être dynamique (déploiement et création de composants à l'exécution) et réflexif (description du composant à l'exécution). Le modèle EJB propose une approche par conteneurs pour la gestion des aspects de QdS permettant ainsi de respecter le principe de génie logiciel de séparation des préoccupations [78]. Les conteneurs permettent alors d'effectuer un filtrage des appels reçus ou émis par les Beans et de déclencher les services de QdS correspondants. Cependant la liste des services de QdS est figée et non extensible. En effet, il est impossible d'ajouter de nouveaux services aux conteneurs afin d'offrir aux Beans une QdS supérieure ou différente. Enfin, un autre inconvénient de ce modèle est qu'il ne cible qu'une seule plateforme de programmation (Java).

2.1.4.3 .NET

2.1.4.3.1 Présentation Le modèle de composants .NET [68] proposé par Microsoft™ a pour domaine d'application les applications largement réparties réalisées à partir de différents langages de programmation. Ce modèle de composants est étroitement lié au framework .NET et se base sur le principe d'une machine virtuelle appelée *Common Language Runtime* (CLR) et d'un langage intermédiaire le *Microsoft Intermediaire Language* (MSIL) comparable au bytecode Java. Chaque composant .NET est programmé dans un langage de programmation donné (C# [42], VB .NET [9], ASP [61], ADO .NET [81], etc.), puis est compilé une première fois en MSIL qui lui même peut être interprété par la CLR.

Les composants .NET sont appelés *assemblage*. D'un point de vue externe, un assemblage est caractérisé

par un nom, un numéro de version et un ensemble de ressources et de méthodes exportées. D'un point de vue interne, un *assemblage* est composé d'un ensemble de fichiers nécessaire à son exécution : il s'agit de fichiers d'implémentation, de ressources et de description. Les fichiers de ressources peuvent être des pages HTML, des images, des sons ou des vidéos. Le fichier de description, appelé *Manifest*, fournit les informations telles que le numéro de version de l'*assemblage*, les méthodes importées et exportées ainsi que les dépendances de modules. Le déploiement d'*assemblage* se fait par simple copie des fichiers le constituant vers l'emplacement de destination. Ils forment les unités de base pour le déploiement d'une application sur le framework .NET, ainsi que les unités de plus bas niveau à partir duquel les mécanismes d'authentification ne sont plus effectués.

Tout comme les EJB, .NET fournit un ensemble de services de QdS aux *assemblages*. Ces services sont la persistance (via ADO.NET [81]), le nommage (via LDAP/UDDI [19]), la sécurité (via Passport [69]) et la transaction (via MTS [44]). De plus, .NET fournit la possibilité d'intégrer des méta données aux *assemblages* : ceci peut être une voie pour la gestion de nouvelles propriétés de QdS. Par exemple, les contrats d'interfaces à base de pré et post conditions peuvent-être réalisés grâce à l'ajout de méta données.

L'implémentation du framework .NET est fournit par Microsoft® pour les systèmes d'exploitation Windows™ XP, Windows™ 2000 et Windows™ CE. La communauté du logiciel libre a récemment mis à disposition une implémentation du framework appelée Mono [73].

2.1.4.3.2 Analyse Un composant dans l'environnement .NET est un *assemblage* : il est déployable et possède des interfaces importées et exportées. Le framework supporte les communications de types synchrones et asynchrones.

Le principal avantage du modèle .NET est qu'il prend en compte totalement et simplement les aspects de déploiement de composants [21]. De plus, .NET fournit un ensemble de services de gestion de la QdS et il est possible d'en intégrer d'autres via le contrôle des pré et post conditions grâce à l'ajout de méta données d'*assemblages*. Enfin, un autre aspect intéressant de ce modèle est qu'il permet l'intéropérabilité des composants écrits dans des langages différents. Cependant, .NET ne fournit pas de modèle de composant clair tel qu'identifié au paragraphe 2.1.3. De plus, la lourdeur du framework, en termes de place mémoire et de performances, fait que ce modèle ne paraît pas adapté aux systèmes embarqués.

2.1.4.4 OSGi

2.1.4.4.1 Présentation L'Open Service Gateway Initiative (OSGi) [77] est une association d'industriels en charge de définir et de promouvoir un modèle et un framework de composants permettant la livraison de services au travers de tous types de réseau (du réseau local à l'Internet). Une des principales caractéristiques du modèle OSGi est qu'il permet de prendre en compte les évolutions dynamiques des applications.

Le modèle OSGi est basé sur deux concepts principaux pouvant être considérés, d'après nos définitions, comme des composants. Il s'agit des bundles et des services. Tandis que dans les modèles précédents un composant est à la fois une unité de composition et de déploiement, OSGi sépare ces deux concepts. Un service est une unité de composition alors qu'un bundle est une unité de déploiement. Une application est composée d'un ensemble de services, alors qu'un bundle accueille un ensemble de services. De ce fait, une application peut être répartie

sur plusieurs bundles et les bundles sont déployés sur une plateforme d'exécution spécifique, appelée plateforme OSGi.

Les bundles possèdent des interfaces requises et fournies identifiant différents types d'interactions possibles. Un bundle peut fournir ou requérir une librairie de code, des services, ou peut requérir directement les services offerts par la plateforme d'exécution.

La plateforme d'exécution OSGi gère le cycle de vie de chaque bundle dont les différents états sont représentés à la figure 2.4. La plateforme permet d'installer (état *UNRESOLVED*) ou de désinstaller (état *UNISTALLED*) un bundle, mais aussi de résoudre ses dépendances (état *RESOLVED*) et de l'arrêter ou de le démarrer (état *ACTIVE*). Il est à noter qu'un bundle ne peut pas être démarré tant que toutes ses dépendances ne sont pas résolues. De plus, la plateforme peut être enrichie de services supplémentaires afin d'offrir d'autres fonctionnalités aux bundles.

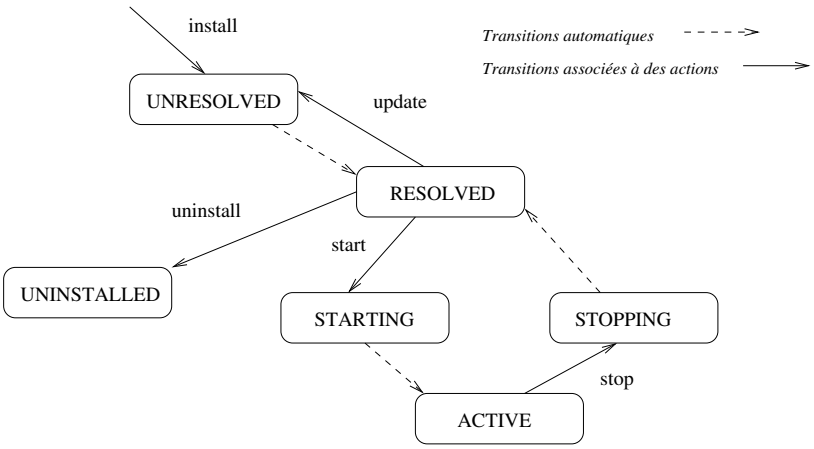

FIG. 2.4 – Cycle de vie d'un bundle OSGi.

Étant donné qu'OSGi cible les applications évoluant dynamiquement, il n'est pas possible de décrire une composition de services statiquement au travers d'un ADL. En revanche, lorsqu'un bundle est installé sur la plateforme d'exécution, ses services fournis sont publiés et la plateforme lui fournit les références pour ses services requis. L'ensemble des connections d'un bundle évolue donc dynamiquement en fonction des services disponibles. Cette dynamicité a un coût : en effet, lorsqu'un bundle est lié à un service ou à package provenant d'un autre bundle, il n'y a aucune garantie pour que le bundle reste disponible. Il faut alors que le bundle interagisse en permanence avec la plateforme d'exécution afin d'être notifié des départs ou arrivées de bundles. De plus, il est à la charge du bundle de mener les actions nécessaires lorsqu'un service requis est désinstallé.

D'un point de vue interne, un bundle est un fichier d'archive java (.jar) contenant un ensemble de packages, de services et de ressources tels que les fichiers de configuration (numéro de version du bundle, services fournis et requis, package fournis et requis), les images ou les sons.

OSGi ne possède pas initialement de mécanismes permettant la gestion de la QdS des bundles ou des services. Cependant, il existe à l'heure actuelle des travaux allant dans ce sens (cf. projet RNRT PISE [4]).

2.1.4.4.2 Analyse OSGi propose la notion de bundle pouvant être exécuté sur une plateforme spécifique et interagissant au travers d'interfaces (de packages ou de services). Il n'existe qu'un mode de communication (syn-

chrone) et l'implémentation d'un bundle est un ensemble de classes java et de fichiers de ressources packagés sous forme de fichiers d'archive. Les originalités du modèle OSGi sont la distinction réalisée entre l'unité de déploiement et l'unité de composition, la gestion du cycle de vie des composants et la gestion dynamique des liaisons.

OSGi se focalise sur les problématiques de déploiement de composants, ainsi que sur l'évolution dynamique des applications. En contrepartie, la prise en compte de la dynamicité implique un coût de gestion des liens entre les bundles. Enfin, bien qu'il n'existe aucun mécanisme de gestion de QdS, il est envisageable d'en intégrer en enrichissant la plateforme d'exécution de services supplémentaires et par l'utilisation des méta-informations sur les bundles.

2.1.4.5 Conclusion

Cette partie a permis de présenter les principales caractéristiques des modèles de composants pour les applications distribuées.

Bien que ces modèles, du fait de leur lourdeur et de leur complexité, ne soient pas adaptés aux contraintes des systèmes embarqués, ils présentent plusieurs caractéristiques intéressantes à retenir pour une gestion de dynamique de la QdS dans les systèmes embarqués ouverts.

Premièrement, l'encapsulation des composants au sein de conteneurs permet la mise en oeuvre du principe de séparation des préoccupations [78] pour la gestion de la QdS afin d'augmenter les possibilités de réutilisation des composants. De plus, l'utilisation de conteneurs permet d'effectuer un filtrage des appels indépendamment des composants. Deuxièmement, la présence d'une plateforme d'exécution mettant en oeuvre des mécanismes permettant le déploiement de composants en cours d'exécution est une caractéristique souhaitée pour les systèmes embarqués ouverts. De même, la possibilité de créer dynamiquement les composants et de gérer leurs liaisons tout au long de leur cycle de vie est primordiale pour ces systèmes. Enfin, troisièmement, l'existence des composants, aussi bien à la phase déploiement qu'à la phase d'exécution, facilite l'évolution dynamique des applications (ajout ou retrait de composants).

Après avoir étudié les principaux concepts des composants pour les applications distribuées, la suite de cette partie se focalise sur les composants dédiés aux systèmes embarqués afin d'en définir les spécificités.

2.1.5 Composants pour les systèmes embarqués

Cette partie s'intéresse aux modèles à composants dans le domaine des systèmes embarqués. Étant donné que ces systèmes ont pour principale caractéristique une limitation des ressources (CPU, mémoire, réseau et batterie), il est intéressant d'étudier les mécanismes mis en oeuvre par ces modèles pour gérer la QdS liée aux ressources.

2.1.5.1 Introduction

Il existe de nombreux modèles à composants dans la littérature ayant pour objectif la construction de systèmes d'exploitations dédiés (OS-Kit [31], Coyote [12], 2K [54], MMlite [43], Pebble [33], Think [29]) ou le développe-

ment d'applications pour les systèmes embarqués (Koala [95], PECOS [80], VEST [86], SEESCOA [93], Rubus [48]). Ces modèles visent les systèmes fortement contraints de contrôle de procédé (PECOS, VEST, SEESCOA, Rubus), les systèmes embarqués grand public (MMlite, Think, OS-Kit, Koala, Pebble) ou les systèmes temps-réel dédiés à la gestion des trafics réseaux (Coyote).

Afin de mieux appréhender ces modèles, nous en présentons plus précisément trois qui sont caractéristiques du domaine. Le premier, Koala, vise les applications ayant des contraintes d'utilisation de la mémoire. Le deuxième, PECOS, vise les applications à contraintes temps-réel et met l'accent sur les aspects temporels. Enfin, le troisième, Think, permet la construction de systèmes d'exploitation dédiés. Son intérêt réside dans ses possibilités d'extension et dans sa structuration en composants à l'exécution.

2.1.5.2 Koala

2.1.5.2.1 Présentation Koala [95] [94] est un modèle de composants, principalement utilisé par Philips depuis la fin des années 1990, pour les périphériques audio et vidéo de types lecteur CD/DVD ou téléviseurs haut de gamme.

Un composant Koala est un morceau de code pouvant interagir avec son environnement uniquement au travers d'interfaces. Un composant peut posséder plusieurs interfaces fournies ou requises. Elles sont spécifiées par un IDL (Interface Description Language) spécifique. De plus, les interfaces peuvent être optionnelles. Une interface fournie optionnelle signifie que celle-ci peut ne pas être implémentée, alors qu'une interface requise optionnelle peut ne pas être liée. Cette approche permet à un composant Koala de s'adapter à son environnement. Pour qu'une interface requise puisse être liée à une interface fournie, il faut que cette dernière implémente, au moins, toutes les fonctions listées par l'interface requise.

Les composants Koala sont décrits à l'aide d'un langage de description de composants spécifique appelé *Koala Language*. Ce langage permet de spécifier les interfaces requises et fournies d'un composant, mais aussi les sous composants qu'il contient ainsi que les liaisons existantes entre ces sous composants. Un composant est représenté sous forme d'un répertoire contenant un ensemble de fichiers d'en tête (pour la description des interfaces), un ensemble de fichiers C (pour l'implémentation des interfaces fournies) et du fichier de description du composant.

La construction du système est réalisée statiquement par une chaîne de compilation adéquate permettant de compiler les différents composants et d'établir les liaisons. De plus, la chaîne de compilation permet d'effectuer des optimisations en termes d'occupation mémoire en éliminant, par exemple, les fonctions non utilisées par les composants. Enfin, à l'exécution, un système Koala se base sur un système d'exploitation temps-réel sous-jacent comprenant un ordonnancement préemptif.

Le support des propriétés de QdS est limité dans le modèle Koala. Cependant, il est possible de spécifier les dépendances des composants envers les tâches du système au travers d'interfaces spécifiques. De plus, l'ordre d'exécution des tâches peut être spécifié en utilisant des relations de précédence et des mécanismes d'exclusion mutuelle. Pour les aspects de consommation mémoire, les composants Koala fournissent une interface particulière, nommée *IResource*, permettant de spécifier leur consommation mémoire requise. De plus, il est possible de déterminer certaines propriétés du système à partir des caractéristiques des composants. Ainsi, Koala fournit un outil [30] permettant de calculer la consommation mémoire globale du système à partir des besoins mémoire de

chaque composant (spécifié par l'interface IResource). Par contre, il est impossible d'enrichir les interfaces par de nouveaux attributs et il n'y a pas de support pour la modélisation des aspects temporels ou de performances du système.

2.1.5.2.2 Analyse Le modèle de composants Koala propose la notion de composant lors des phases de développement et de programmation. Lors de ces deux phases, les composants sont réutilisables. De plus, les deux types d'interfaces existent (requises et fournies), mais seul un mode de communication synchrone est possible. Un composant est packagé au sein d'un répertoire, mais il n'existe pas d'instance de composants identifiable à l'exécution. En effet, lors de la phase de compilation, la structure du système en composants est perdue et laisse place à un système monolithique.

Koala a pour avantage de clairement identifier les interactions entre un composant et son environnement car toutes les interactions sont réalisées au travers d'interfaces. L'environnement d'un composant peut aussi bien être logiciel (dépendance vers d'autres composants) que matériel (dépendance vers des services spécifiques du système d'exploitation sous-jacent). De plus, Koala permet de prendre en considération les aspects de consommation mémoire des composants. Cependant, il n'est pas possible d'étendre le modèle afin de considérer d'autres ressources. De plus, Koala ne fournit pas d'éléments afin de disposer d'un contrôle du système d'exploitation sous-jacent.

2.1.5.3 PECOS

2.1.5.3.1 Présentation Le modèle de composants PECOS [80] est issu d'un projet européen IST démarré en 1999 et a pour cible les périphériques industriels tels que les capteurs (température, pression, etc.) ou les actionneurs.

Un composant PECOS peut être composé hiérarchiquement et possède des interfaces définies par des *ports*. Les ports représentent les données partageables entre les différents composants. Un port est caractérisé par une *direction* précisant si il est fourni (la donnée est écrite) ou requis (la donnée est lue). Les communications inter-composants s'effectuent donc par partage de données. De plus, chaque composant possèdent deux opérations permettant de gérer leur cycle de vie : *start* et *stop*. Les composants de plus bas niveau, appelés *composants feuilles*, sont directement implémentés par un langage de programmation (Java ou C++). Les *composants composites* sont construits par la connection de plusieurs composants au travers de leurs ports.

PECOS définit trois types de composants : les composants actifs, les composants passifs et les composants événementiels. Les composants actifs possèdent leur propre thread de contrôle. Un système complet est composé de plusieurs composants et est toujours modélisé comme un composite actif. En effet, les composites actifs ordonnancent les composants les constituant afin de respecter les échéances imposées par les contraintes temps-réel.

Les composants passifs ne possèdent pas de thread de contrôle. Un composant est ordonnancé (appels à *start* et *stop*) par le composant actif le plus proche, hiérarchiquement parlant.

Les composants événementiels sont des composants s'exécutant sur l'occurrence d'un événement. Généralement, ce type de composants est utilisé pour modéliser les éléments matériels du système générant périodiquement des événements.

Enfin, chaque composant actif est caractérisé par des propriétés permettant de spécifier ses besoins mémoire, son WCET (Worst Case Execution Time), sa période et sa priorité.

La figure 2.5 représente le méta-modèle du modèle de composants PECOS.

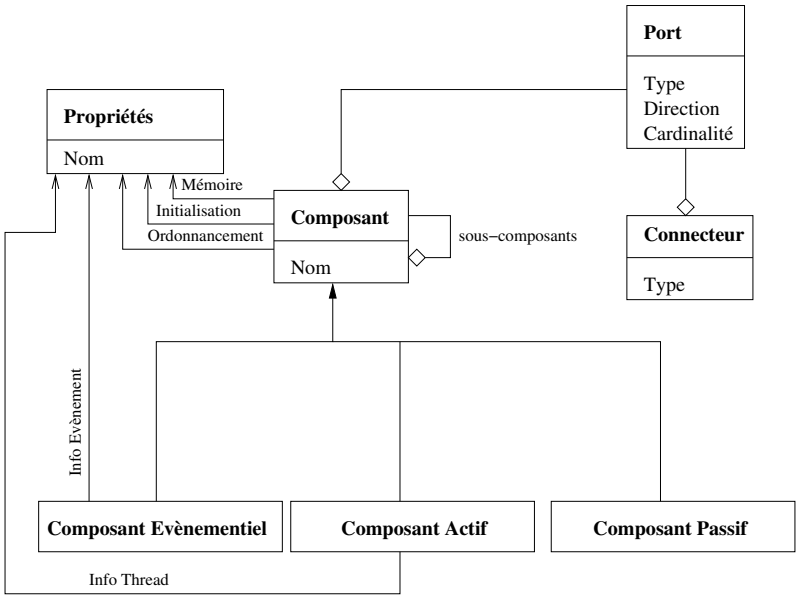

FIG. 2.5 – Méta-modèle du modèle de composants PECOS.

Parallèlement à la description statique du système, PECOS fournit un modèle d'exécution. Ce modèle se base sur les réseaux de Petri [98] afin de pouvoir raisonner sur les contraintes temporelles et générer les séquences d'ordonnancements associés. Le modèle d'exécution intègre les contraintes de synchronisation et les contraintes temps-réel. La synchronisation permet de spécifier les contraintes sur les flux de données entre les composants (principalement pour les composants appartenant à différents threads de contrôle). Les contraintes temps-réel permettent de spécifier que le composant doit être exécuté dans le respect de sa période et de son échéance.

Enfin, PECOS utilise le langage de description d'architecture CoCo [83] pour décrire les types de composants, leur implémentation, ainsi que le graphe de composition des composants constituant le système complet.

Dans le cadre du projet européen, le modèle PECOS a été développé pour le contrôle d'un actionneur pneumatique et des sociétés telles que ABB (Asea Brown Bovery AG, Allemagne) commencent à l'intégrer dans leurs propres plateformes.

2.1.5.3.2 Analyse Le modèle de composants PECOS propose la notion de composant et d'interfaces fournies et requises, appelées ports et caractérisées par une direction. Ici, la notion de composant vis-à-vis de l'exécution, est plus précise que dans les modèles précédents. PECOS fait la distinction entre les composants actifs, passifs et événementiels. De plus, il existe deux modes de coopération entre les composants (par partage de données ou par flot de contrôle via *start* et *stop*) et un ensemble de propriétés configurables est proposé, telles que la mémoire et les niveaux de priorité.

PECOS est un modèle visant les systèmes temps-réel dur. En effet, un certain nombre de mécanismes sont mis en oeuvre afin de capturer les aspects temporels des composants. Cependant, ce modèle possède uniquement la vue composant lors de la phase de développement : une fois compilé, le système résultant devient monolithique et perd sa structuration en composants. Cette approche permet de réduire les coûts dûs à la gestion dynamique des composants et d'avoir un système optimisé. Mais l'évolution dynamique, ainsi que la maintenance du système deviennent, de ce fait, très limitées.

2.1.5.4 Think

2.1.5.4.1 Présentation Le modèle de composants générique Fractal reste une spécification qu'il faut implémenter selon le domaine d'application. Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à l'implémentation Think. Think [29] est une implémentation du modèle Fractal pour la construction de systèmes d'exploitation dédiés. La philosophie de Think est d'utiliser systématiquement le concept de composant afin de construire un système d'exploitation adaptable. Celui-ci est alors vu comme un assemblage de briques ou composants logiciels nécessaires à l'exécution du système. Pour cela, Think définit deux catégories de composants. Une première catégorie réifie la couche matérielle de la plateforme, communément appelée HAL pour Hardware Abstraction Layer. Ces composants réifient les exceptions, la MMU, le cache ou encore les pilotes de périphériques (écran, clavier, disque, port série, etc.). Les composants de la couche HAL ont été portés sur plusieurs plateformes dont l'Intel® StrongARM™ (présent dans les iPaq™ h3600 et h3800), l'Intel® xScale™ (présent dans les iPaq™ h3900 et h2200) ou encore le Portal Player PP5002™ (présent dans l'Apple® iPod™). Une deuxième catégorie de composants fournit les services classiques des systèmes d'exploitation. Il existe, par exemple, des composants mémoire (gestion de mémoire plate ou paginée), des composants pour l'exécution multitâches (threads, sémaphores, ordonnanceurs coopératifs, à priorités ou round robin, etc.) et des protocoles de communication (ethernet, IP, bluetooth, etc.).

Chaque composant Think fournit au minimum quatre interfaces. La première, appelée *LifeCycleController* permet de gérer le cycle de vie du composant en spécifiant les actions à mener lors de son démarrage et de son arrêt. La deuxième, appelée *BindingController*, permet de lier les interfaces requises par le composant. La troisième, appelée *ContentController*, permet d'ajouter, de supprimer et de lister les composants le constituant. Enfin, la dernière, appelée *ComponentIdentity*, permet de retourner l'identificateur du composant, ainsi que la liste de ses interfaces (requises et fournies).

Un composant Think est implémenté en langage C ou directement en assembleur, tandis que ses interfaces sont définies au travers d'un IDL. Bien que le langage C ne fournisse pas le concept d'interfaces, Think met en place le mécanisme nécessaire permettant de les utiliser. Ce mécanisme reprend le principe des tables de fonctions virtuelles (plus communément appelées *vtables*) présent dans le langage C++. L'interface d'un composant est représentée par une structure, appelée *interface descriptor*, contenant un pointeur vers un tableau de fonctions, appelé *methods structure*, et un pointeur vers les données du composant (cf. figure 2.6). Le tableau *methods structure* contient les références vers les fonctions implémentant les services de l'interface. L'appel de ces services se fait alors à partir du pointeur sur l'interface, appelé *interface reference*, référençant la structure *interface descriptor*, qui est ensuite déréférencé vers la fonction correspondante.

La description d'un système est spécifiée grâce à l'ADL Fractal. L'ADL permet de décrire les différents composants du système ainsi que leurs liaisons. Une fois compilé, un composant est stocké sous forme de fichiers binaires. Enfin, Think fournit une chaîne de compilation complète allant des parsers IDL et ADL à la compilation

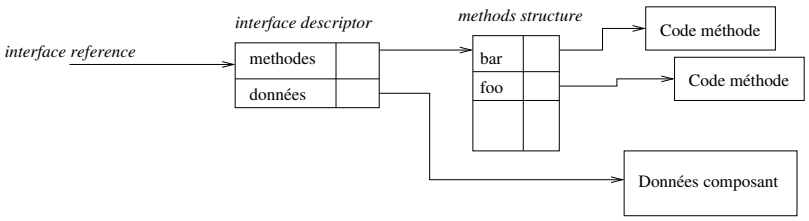

FIG. 2.6 – Représentation graphique du mécanisme mis en oeuvre par Think pour l'utilisation d'interface.

et l'instanciation des composants.

Une fois le système Think compilé, celui-ci garde sa structuration en composants afin de faciliter son évolution dynamique. Le coût engendré par la structuration en composants est estimé en moyenne à 2% de la taille totale du système [28].

Enfin, Think repose sur un modèle de composants suffisamment extensible pour permettre l'intégration de mécanismes de sécurité [50]. Les mécanismes de sécurité portent sur l'authentification de composants afin de vérifier qu'un composant puisse utiliser les services d'un autre composant.

2.1.5.4.2 Analyse Vis-à-vis des définitions du paragraphe 2.1.3, Think fournit le concept de composant deployable et réutilisable aussi bien en phase de développement qu'à l'exécution. Il existe la notion d'interfaces requises et fournies qui matérialisent toutes les dépendances d'un composant. Le mode de communication entre les composants est de type synchrone. Un composant Think est packagé soit sous forme de fichiers binaires, soit sous forme de répertoires contenant les définitions des interfaces et son implémentation. Seule la forme binaire peut être déployée, tandis que la forme répertoire existe lors de la phase de développement. Enfin, il existe la notion d'instance de composants et il alors possible de manipuler un composant lors de l'exécution du système.

Ni le modèle Fractal, ni l'implémentation Think ne prévoit initialement de mécanismes de gestion de QdS. Cependant, Think possède de nombreux atouts permettant d'envisager de tels mécanismes. Premièrement, le modèle de composants utilisés par Think est suffisant ouvert et flexible pour intégrer des aspects de gestion de QdS. Deuxièmement, grâce à la philosophie suivie par Think, où tout est composant (y compris les ressources matérielles), il est possible de disposer d'un contrôle fin des ressources. Et troisièmement, contrairement aux deux modèles précédents, l'exécution d'un système Think est autonome, c'est à dire qu'il ne requiert pas un noyau de système d'exploitation sous-jacent.

2.1.6 Conclusion

Cette partie a permis de présenter et d'illustrer les principales caractéristiques des modèles de composants dans le domaine des systèmes embarqués.

Ces modèles sont caractérisés par plusieurs aspects :

1. la gestion de la QdS est limitée aux problèmes d'ordonnancement temps-réel des tâches et à la consommation mémoire. Ces aspects sont gérés statiquement et a priori.
2. afin d'éviter la mise en oeuvre de plateformes d'exécution complexes, grandes consommatrices de res-

sources, ces modèles ne supportent pas le déploiement de composants à l'exécution. La composition de composants est donc réalisée avant la phase de compilation au moment de la conception.

3. à l'exception de Think, les systèmes, une fois compilés, sont monolithiques. En effet, les composants n'existent qu'à la phase de développement et la structure du système en composants est perdue lors de la compilation.
4. ces modèles, excepté Think, se basent sur l'existence d'un micro-noyau temps-réel sous-jacent.

Ces modèles, ainsi que leurs technologies associées, ont été développés pour des classes de systèmes particuliers : il s'agit généralement de décrire une organisation spécifique du code en s'appuyant sur un système d'exploitation temps-réel sous-jacent. Le concepteur du système peut ainsi exprimer des contraintes sur l'exécution des composants, mais la manière dont celles-ci sont gérées (par exemple pour l'ordonnancement) est implicite aux outils de compilation fournis. Il est à noter le cas particulier du modèle Fractal/Think qui propose une approche homogène de développement des systèmes à l'aide de composants. De plus, Think se distingue des précédents modèles par le fait que la structure en composants du système est réifiée à l'exécution et qu'il ne se repose pas sur un micro-noyau car il est lui-même le résultat d'une composition. Cela permet alors d'avoir un contrôle total des ressources du système ce qui est primordial pour la gestion de la QdS.

A travers les modèles présentés dans cette partie, nous pouvons remarquer que l'efficacité des composants est un critère primordial : en effet, les composants sont implémentés soit en langage C soit directement en assembleur. De plus, ces modèles se focalisent sur la QdS liée à l'utilisation des ressources, et résolvent les problématiques de QdS (temps réel, mémoire) avant la phase de déploiement des systèmes. Ainsi, les politiques de gestion de la QdS sont fixées par le modèle et ne peuvent pas être modifiées ou étendues.

Enfin, deux principaux obstacles ont été identifiés pour l'adoption de la technologie composants dans les systèmes embarqués [79] :

– premièrement, il n'existe pas de modèle standardisé de composants pour ce domaine d'application. Ceci est principalement dû au fait que l'ensemble des industriels concernés ont des priorités différentes concernant les caractéristiques devant être offertes par de tels modèles. Il est donc nécessaire de disposer d'un modèle suffisamment générique pouvant être spécialisé. Cela permettrait de mutualiser les efforts et de disposer d'outils communs.
– deuxièmement, les modèles de composants doivent pouvoir supporter la spécification et la prédiction de contraintes temporelles et de QdS aussi bien statiquement que dynamiquement.

Après avoir identifié les caractéristiques des modèles pour les applications distribuées et les systèmes embarqués, nous nous focalisons maintenant sur les architectures de gestion de QdS.

2.2 Architectures de gestion de QdS

Les architectures de gestion de QdS sont le deuxième domaine de recherche sur lequel porte cette thèse. C'est pourquoi il est nécessaire de définir les principaux termes du domaine, ainsi que d'identifier les principales caractéristiques des architectures existantes. Par la suite, nous distinguons trois types d'architectures : les architectures issues des organismes de normalisation (ISO [46] et l'OMG [3]), les architectures de gestion de QdS de bout en bout pour les applications distribuées (QoSA [17], TINA [2], OMEGA [74], MASI [56], End-to-End QoS Framework [35], [64]) et enfin, les architectures spécifiques aux composants (QuA [5], [84], [24], [65], [25], [58]).

Les architectures issues des organismes de normalisation se focalisent principalement sur la QdS liée au réseau. Les architectures liées à la QdS de bout en bout gèrent la QdS de la partie serveur à la partie cliente pour les applications distribuées. Elles prennent en compte différents types de ressources (CPU, mémoire et réseau) à chaque niveau des systèmes : niveau applicatif, niveau système d'exploitation et niveau ressources. Pour cela, ces architectures se basent sur des systèmes d'exploitation permettant la prise en compte de QdS, tels que RT Mach [89] et Rialto [53], et des réseaux haut débit intégrant des mécanismes de QdS tels que ATM [55]. Enfin, dans le domaine des composants, les architectures existantes ciblent les systèmes embarqués multimédia. Ces architectures se basent sur l'existence d'une machine virtuelle. Il est alors impossible de garantir et de réserver la QdS d'une application étant donné que le contrôle des ressources est limité (pas de connaissance et de contrôle sur la plateforme matérielle d'exécution).

Nous présentons, ci-après, une architecture pour chacune de ces trois familles afin de mieux appréhender leurs caractéristiques. La première architecture présentée est celle définie par l'ISO qui est une des premières architectures normalisées. Puis, nous présentons une des architectures parmi les plus complètes pour la gestion de QdS de bout en bout appelée QoSA. Enfin, nous présentons l'architecture QuA dédiée à la gestion de la QdS pour des applications à composants.

Pour débuter, nous définissons les termes relatifs aux architectures de gestion de QdS et donc à la QdS afin de fixer la terminologie utilisée dans la suite de ce rapport. Pour cela, nous nous basons principalement sur les travaux de synthèses d'architectures de QdS [6, 82] et sur les termes définis par l'ISO [46], l'OMG [3, 40] et l'ITU-T [92].

2.2.1 Définitions

2.2.1.1 Architecture de gestion de QdS

Le premier terme que nous définissons est celui d'architecture de gestion de QdS. Pour cela, nous commençons par définir ce qu'est plus généralement une architecture logicielle, puis, plus spécifiquement, ce qu'est une architecture de gestion de QdS. La définition d'une architecture logicielle, telle qu'elle est entendue dans cette thèse, est celle proposée en 2003 par [10] :

> *«The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships along them.»*

Cette définition signifie qu'une architecture logicielle est un ensemble de contraintes de programmation définissant les éléments logiciels d'un système, ainsi que les relations entre ces éléments. Du point de vue QdS, [18] définit une architecture de gestion de QdS comme étant :

> *«The primary intention of QoS architecture is to define a set of QoS configurable interfaces and services that formalize QoS in the end system, providing a framework for the integration of control, maintenance and management mechanisms to meet application level end-to-end QoS requirements.»*

Une architecture de gestion de QdS fournit donc un cadre architectural permettant l'intégration des différents aspects de gestion de QdS.

Nous donnons maintenant les termes relatifs à la gestion de la QdS.

2.2.1.2 Caractéristiques et contraintes de QdS

La prise en compte de la QdS peut s'effectuer à plusieurs niveaux lors de la mise en oeuvre d'une application ou d'un système. En effet, elle peut être soit directement modélisée lors de la conception de l'application en identifiant plusieurs éléments clés pour l'exécution (quantité mémoire, priorité des processus, etc.), soit être exprimée dans le cahier des charges par des éléments clés perçus par l'utilisateur (comme, par exemple, une échéance). Ces éléments clés, sur lesquels l'utilisateur porte une attention particulière, et qui n'entrent pas directement dans les traitements, sont appelés *caractéristiques de QdS* [40].

A partir d'une caractéristique, l'utilisateur peut spécifier un ensemble de valeurs autorisées appelées *contraintes de QdS* [40].

2.2.1.3 Spectre de QdS

Il existe deux manières différentes d'aborder la gestion de la QdS. La première, dite statique, consiste à exprimer une exigence une fois pour toute. L'autre, qualifiée de dynamique, consiste à pouvoir faire évoluer la contrainte au cours de l'utilisation de l'application et ainsi adapter le comportement, à la fois, de l'application et du système en fonction de l'évolution de l'environnement dans lequel ceux-ci s'exécutent. Ces deux manières forment un intervalle de gestion de la QdS, appelé *spectre de QdS* [24]. La conception d'une architecture de gestion de QdS est plus ou moins difficile en fonction du niveau de gestion souhaité par rapport au spectre de QdS.

2.2.1.4 Contrat de QdS

Au sein d'un système, plusieurs applications peuvent résider et être soumises aux exigences des utilisateurs. Afin de répondre aux exigences de l'utilisateur, les applications s'appuient sur l'utilisation de ressources, pouvant elles-mêmes être d'autres applications ou des ressources matérielles. Classiquement une distinction est faite entre les ressources actives (CPU, bus de communication ,etc.) et les ressources passives (mémoires, routeurs, etc.) [82]. Les utilisateurs ou les applications, expriment leurs exigences, en termes de QdS, au travers de *contraintes* sur les ressources utilisées [40]. Ces contraintes ont des valeurs qui constituent ce que l'on nomme les *besoins de QdS*. A l'opposé, chaque ressource peut proposer une *offre de QdS*. Ainsi, deux éléments du système pourront être liés par une *relation de QdS*.

Pour formaliser le lien sélectionné lors de l'utilisation d'un élément par un autre, on parle de *contrat de QdS* identifiant alors le choix d'une relation de QdS parmi celles possibles. Le contrat est au centre de la gestion de la QdS. Il caractérise une mise en relation entre une QdS requise (par exemple par un composant) et une QdS offerte (par exemple par un autre composant). Les *contrats de QdS* possèdent trois propriétés fondamentales devant être prises en compte par une architecture de gestion de QdS [6] : la spécification, l'initialisation et la gestion (voir figure 2.7).

2.2.1.4.1 Spécification de QdS La spécification de la QdS a pour but de formaliser les *contraintes de QdS*. De ce fait, la spécification est, par nature, déclarative. La spécification peut comporter plusieurs aspects :

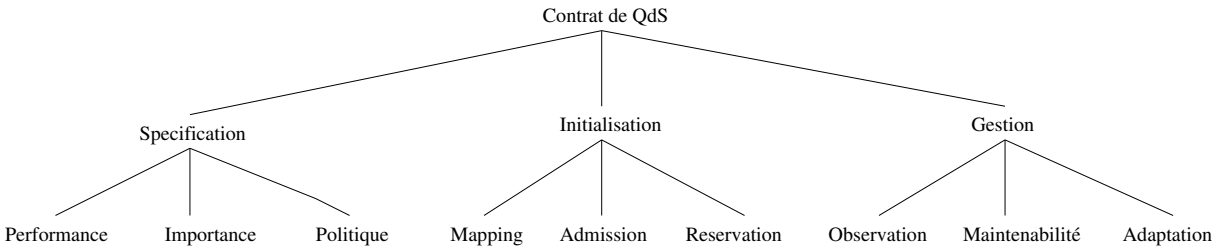

FIG. 2.7 – Propriétés d'un contrat de QdS.

- l'aspect *performance* spécifie le niveau des caractéristiques de QdS sous forme de valeurs fixes, d'intervalles, de fonctions du temps, etc.
- l'*importance* spécifie le niveau d'importance donné pour la performance spécifiée.
- la *politique* spécifie le degré d'adaptation possible et les actions devant être menées lors d'une violation d'un contrat de QdS.

2.2.1.4.2 Initialisation La phase d'initialisation est l'activité permettant de prendre les dispositions nécessaires à l'établissement d'un contrat. L'initialisation regroupe trois mécanismes :

- le *mapping* est la traduction d'un niveau supérieur vers un niveau inférieur. On considère classiquement quatre niveaux : application, services, système d'exploitation et ressources. Par exemple, une spécification de niveau de QdS `BON` pour une application de type `VIDEO` peut être traduite par `25 images/sec` au niveau service, `1 thread` au niveau système d'exploitation et `35% CPU` et `50 Ko MEMOIRE` au niveau ressources.
- Le *test d'admission* a pour objectif de comparer la QdS offerte à celle requise afin de pouvoir valider, ou non, l'établissement d'un contrat. Un contrat ne peut être établi que si les valeurs de la QdS requise sont un sous-ensemble des valeurs de la QdS fournie [40]. L'établissement d'un contrat consiste à fixer ces valeurs.
- La *réservation* s'effectue une fois le test d'admission validé et le contrat établi. Dans ce cas, tous les éléments nécessaires à la réalisation du contrat sont réservés.

2.2.1.4.3 Gestion Afin de maintenir le niveau de QdS défini par les contrats, il est nécessaire de disposer de mécanismes supplémentaires à la simple réservation. La gestion de la QdS regroupe trois activités :

- L'*observation* de la QdS est le mécanisme permettant à chaque couche de surveiller le niveau de QdS effectivement fourni par la couche inférieure.
- La *maintenabilité* de la QdS permet de comparer le niveau de QdS observé à celui spécifié, et d'effectuer des opérations d'ajustement.
- L'*adaptation* de la QdS survient lorsque le mécanisme de maintenabilité n'arrive plus à effectuer l'ajustement nécessaire. A ce moment, l'architecture est obligée d'effectuer des opérations de re-négociation des contrats afin de définir de nouveaux niveaux de QdS.

2.2.2 Exemples d'architectures de gestion de QdS

2.2.2.1 L'architecture de l'ISO

L'ISO a, dès 1995, proposé une architecture de gestion de la QdS [46] pour le modèle de référence en couches OSI (Open Systems Interconnection). Cette architecture est donc orientée tout naturellement vers une gestion de la QdS pour les réseaux.

La définition de l'architecture proposée par l'ISO se base sur quatre concepts clés :
- *les caractéristiques de QdS* définissent le niveau de QdS réel associé à un élément du système.
- *les besoins de QdS* sont l'expression des exigences des utilisateurs sur les *caractéristiques de QdS*. Il s'agit donc de spécifications de QdS.
- *les catégories de QdS* regroupent un ensemble de *besoins de QdS* spécifiques à un domaine d'application particulier (par exemple les besoins de QdS liés aux communications temps-réel).
- *les fonctions de gestion de QdS* sont appliquées aux caractéristiques de QdS afin de satisfaire aux besoins de QdS.

L'architecture proposée par l'ISO est représentée à la figure 2.8. Elle est constituée de deux types d'éléments : les éléments spécifiques à chaque couche identifiée par le modèle OSI et les éléments ayant une vue globale du système. Le rôle de l'élément PCF (Policy Control Function) est d'implémenter la politique de gestion de QdS

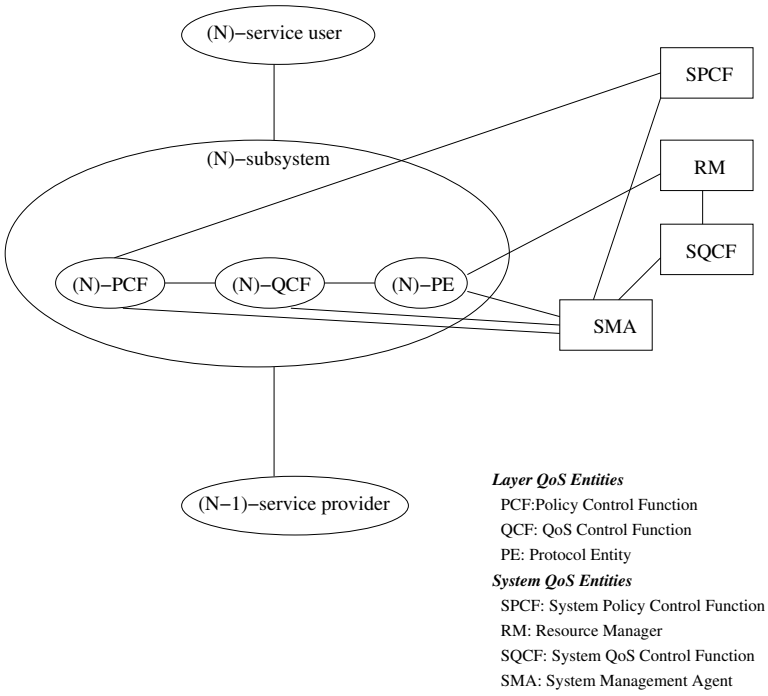

FIG. 2.8 – Architecture de l'ISO de gestion de QdS.

de la couche N du système. Cet élément identifie et définit les actions prioritaires pour une catégorie de QdS devant être réalisées par cette couche. L'implémentation de la politique est spécifique à chaque couche du système, c'est pourquoi elle ne peut être généralisée. Généralement, le PCF inclue les politiques de sécurité et de gestion des ressources. Le PCF s'appuie sur le QCF (QoS Control Function) pour réaliser les actions selon leur niveau

de priorité. Celui-ci a en charge de sélectionner, puis de configurer, l'élément PE (Protocol Entity) qui réalisera réellement l'opération.

D'un autre côté, les éléments ayant une vue globale du système implémentent les fonctions de gestion de QdS. Ils sont composés du SPCF (System Policy Control Function) qui définit les politiques de chaque PCF et s'appuie sur l'élément SMA (System Management Agent) afin d'observer les ressources utilisées par chaque couche. Enfin, l'élément SQCF (System QoS Control Function) définit la politique de maintenabilité devant être appliquée pour chaque PE, laquelle est implémentée par l'élément RM (Ressource Manager).

2.2.2.2 Architecture de gestion de QdS de bout en bout

L'architecture QoS-A (Quality of Service Architecture) [18] a été définie par l'université de Lancaster en 1996. Il s'agit d'une architecture en couches permettant la gestion et le contrôle de la QdS de bout en bout de flux multimédia utilisant des réseaux haut débit de type ATM.

QoS-A définit trois concepts clés : le *flux*, le *contrat de service* et la *gestion de flux*. Un *flux* caractérise un flot de données possédant des caractéristiques de QdS. Un flux possède plusieurs activités tout au long de son cycle de vie, allant de sa production à sa consommation, en passant par sa transmission. Un *contrat de service* est une liaison avec accord sur les valeurs des caractéristiques de QdS entre un utilisateur et un fournisseur de service. Enfin, la *gestion de flux* permet la gestion et la maintenance des niveaux de QoS définis par le contrat de service pour un flux donné. Cela nécessite une gestion active de la QdS et un couplage fort entre les différents éléments du système (réseau, protocole de communication, système d'exploitation, gestion des équipements, etc.).

La figure 2.9 schématise l'architecture globale de QoS-A. Cette architecture est composée de 6 couches hori-

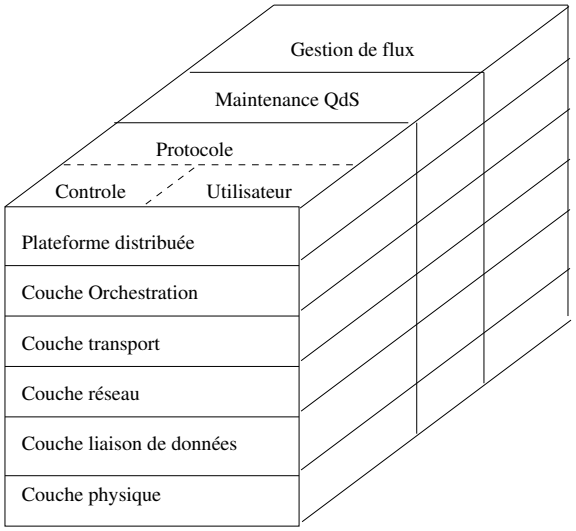

FIG. 2.9 – Architecture QoS-A de gestion de QdS.

zontales et de 3 pans verticaux. Les couches horizontales ont pour principales fonctions d'assurer le transport du *flux* entre les applications réparties. La couche *plateforme distribuée* permet le support des applications réparties en leur fournissant les services nécessaires aux transports des flux multimédia et aux spécifications de leur QdS.

La couche *orchestration* effectue les synchronisations entre les différents flux multimédia interdépendants (par exemple, le son et l'image pour une vidéo). Les couches suivantes sont les couches classiques du modèle OSI de communication.

Les trois pans verticaux sont en charge de la gestion de la QdS des flux, c'est-à-dire de la gestion des contrats de service. Le pan protocole est en charge du transfert du flux d'une couche horizontale à l'autre. Ce pan est décomposé en deux sous pans, l'un réalisant le transport des données du flux (sous pan *utilisateur*) et l'autre en assurant le contrôle (sous pan *contrôle*). Le pan suivant, *maintenance*, est en charge de la maintenance des niveaux de QdS des flux. Pour cela, il s'appuie sur les services offerts par le sous pan contrôle. Enfin, le pan de *gestion de flux* est le pan principal puisqu'il est en charge de l'initialisation (incluant le test d'admission, le mapping d'une couche à l'autre et la réservation) des contrats de services, ainsi que de leur adaptation.

2.2.2.3 Architecture de gestion de QdS pour les systèmes à composants

QuA [5] est une architecture récente (le projet QuA a démarré en 2002) proposée par le laboratoire Simula (Norvège) pour la gestion de la QdS des applications à base de composants. Cette architecture vise principalement les applications distribuées de type multimédia.

L'architecture QuA, représentée à la figure 2.10, poursuit deux objectifs principaux : le premier est de pouvoir dynamiquement composer les composants formant une application, et le deuxième est de pouvoir gérer la QdS de ces composants. Au sein de l'architecture, la gestion de la QdS des composants est gérée par des composants spécifiques. Cette approche permet de réutiliser les composants de gestion de QdS, mais aussi de les modifier sans avoir à reprogrammer toute l'architecture.

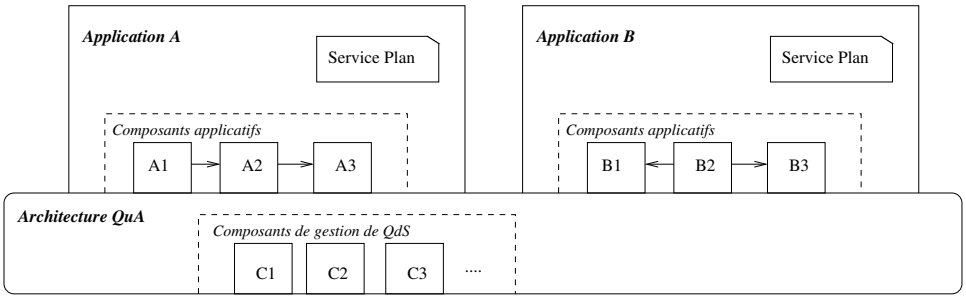

FIG. 2.10 – Architecture QuA de gestion de QdS.

Les composants de gestion de QdS se basent sur les spécifications de QdS utilisateur, réalisées au sein des *utility functions* (non représentées à la figure 2.10), et des spécifications de QdS requises et fournies par les applications, réalisées au sein de *service plan*. Les *service plan* jouent un rôle central dans l'architecture QuA puisqu'ils sont en charge de fournir :

- les identificateurs des composants constituant l'application,
- le graphe de dépendance des composants,
- les paramètres de configuration des composants,
- les dépendances fonctionnelles de l'application à son environnement d'exécution (librairies),
- les caractéristiques de QdS de l'application,
- un modèle de QdS identifiant les valeurs possibles des caractéristiques de QdS,

– et un mapping de QdS, identifiant les niveaux de QdS sur chaque composant.

Les *service plan* sont donc utilisés aussi bien pour construire dynamiquement l'application que pour gérer la QdS des composants la constituant.

Une fois que l'architecture a configuré l'application suivant les besoins de l'utilisateur, les composants de gestion de QdS observent l'évolution des ressources afin d'adapter le niveau de QdS de l'application en cas de fluctuation.

L'implémentation la plus avancée de l'architecture QuA est réalisée en Java et des premières expérimentations dans le domaine des systèmes embarqués multimédia mobiles ont été menées.

2.2.3 Conclusion

Cette partie a permis de présenter les principales caractéristiques des architectures de gestion de QdS. Ces architectures ont pour principal objectif de pouvoir gérer les aspects de QdS des applications de façon dynamique. Pour cela, elles se basent sur le principe de contrat de QdS dont trois grands principes de gestion ont été identifiés. Il s'agit de la spécification (performance, importance et politique), de l'initialisation (mapping, test d'admission et réservation) et de la gestion dynamique (observation, maintenance et adaptation). Avec l'architecture proposée par l'ISO, chaque élément est clairement dissocié dans des entités distinctes (configuration, choix, réservation, action). Cette approche permet de découpler les politiques des mécanismes de gestion de QdS [60] afin d'augmenter la flexibilité de l'architecture. L'architecture QoS-A s'appuie sur une structuration en couches et montre que les trois concepts de gestion de contrat sont suffisants pour une gestion dynamique de la QdS. Enfin, l'architecture QuA démontre l'intérêt, d'un point de vue gestion de QdS dynamique, de pouvoir identifier les composants à l'exécution du système. De plus, la gestion de la QdS s'appuie sur des méta-données regroupées par applications. Cependant, QuA s'appuie sur un système d'exploitation et sur les services d'une machine virtuelle sous-jacente ce qui rend difficile, voire impossible, le contrôle fin des ressources de la plateforme.

Les architectures définissent des concepts génériques qui demandent à être implémentés pour démontrer leur pertinence et leurs performances. Les implémentations actuelles des architectures s'appuient sur des langages de haut niveau tel que Java et sont dédiés à la partie communication.

Les deux premières parties de ce chapitre ont permis d'identifier les principales caractéristiques des modèles de composants, puis des architectures de gestion de QdS. Nous allons présenter maintenant les motivations pour la définition d'une nouvelle architecture de gestion de QdS pour les systèmes embarqués ouverts à composants.

2.3 Motivations pour une nouvelle architecture de gestion de QdS

La partie portant sur les modèles de composants permet maintenant d'analyser les réponses apportées par les approches actuelles aux problématiques soulevées par les systèmes embarqués ouverts à composants en ce qui concerne la gestion de QdS (cf. partie 1 page 2) :

– **Généricité**. Les modèles actuels pré-définissent toujours la façon dont la QdS des composants va être gérée. Cela est vrai aussi bien dans le cadre des applications distribuées (politique de sécurité ou de persistance

pré-définie) que dans le cadre des systèmes embarqués (par exemple, la politique d'ordonnancement des tâches est fixée).
- **Hétérogénéité**. L'hétérogénéité n'est abordée par les approches actuelles qu'au niveau des langages d'implémentation (.NET) et des systèmes d'exploitation (EJB, CCM). Dans ce cas, la gestion de la QdS de niveau ressources n'est pas adressée.
- **Dynamicité**. La dynamicité est uniquement prise en compte par les modèles ne gérant pas la QdS au niveau des ressources (EJB, CCM, .NET, OSGi et Think).
- **Confiance**. Seuls les modèles ne permettant pas la dynamicité assurent que les niveaux de QdS consommés par un composant ne soient pas supérieurs à ceux contractualisés (PECOS, VEST, Koala).
- **Auto-configurabilité**. L'auto-configurabilité n'est pas prise en compte par les modèles de composants actuels. En effet, les modèles à composants présupposent que les niveaux de QdS requis par un composant ou une application soient connus.
- **Réutilisabilité**. La réutilisabilité des mécanismes de gestion de QdS n'est pas, non plus, une problématique adressée par les modèles de composants.

Les principes identifiés lors de l'étude des architectures de gestion de QdS fournissent des pistes intéressantes (séparation des préoccupations, conteneurs de composants, contrats de QdS, gestion des contrats, etc.) pour la gestion dynamique de la QdS dans les systèmes embarqués ouverts à composants. Cependant, l'étude montre que ces architectures ne sont pas adaptées aux systèmes visés par cette thèse. En effet, ces architectures se focalisent principalement sur des systèmes distribués et sur des problématiques liées à la gestion de la QdS réseau. De plus, les architectures existantes pour les applications à composants ne permettent pas de gérer finement les ressources matérielles des plateformes d'exécution, car elles s'appuient sur des machines virtuelles masquant les ressources. De ce fait, les politiques de gestion de QdS pouvant être fournies par l'architecture dépendent de celles fournies par la machine virtuelle et par le système d'exploitation.

Cette partie montre donc qu'il y a un manque au niveau de la gestion de QdS dans les systèmes embarqués à composants. Ce constat, dont font également part les membres du projet Artist [79], nous amène à proposer une nouvelle architecture de gestion de QdS. Cette architecture, appelée Qinna, a pour objectif de répondre aux problématiques présentées en introduction (cf. page 2). Elle s'appuie sur les caractéristiques des modèles de composants dédiés aux applications distribuées (dynamicité, composants à l'exécution, conteneur), ainsi que celles des modèles dédiés aux systèmes embarqués (efficacité, gestion des ressources) et celles des architectures de gestion de QdS (dynamicité, contrats de QdS, gestion des contrats).

Chapitre 3

Qinna

Ce chapitre présente l'architecture de gestion de QdS, appelée Qinna, proposée dans cette thèse. Une première partie permet d'introduire les principes retenus pour l'architecture. Une deuxième partie définit l'architecture Qinna, tandis qu'une troisième partie en présente les principes de mise en oeuvre dans les systèmes à composants. Enfin, une dernière partie conclue le chapitre en détaillant l'apport de Qinna par rapport aux problématiques des systèmes ouverts.

3.1 Introduction

3.1.1 Objectifs

L'objectif de l'architecture Qinna est de répondre aux problématiques de gestion de QdS des systèmes ouverts à composants identifiées au chapitre d'introduction page 2. L'architecture Qinna doit donc :

- être **générique** : c'est-à-dire être non liée à une gestion particulière de la QdS,
- être **dynamique** : permettre la gestion de la QdS en cours d'exécution,
- intégrer l'**hétérogénéité** : permettre l'utilisation simultanée de différents langages de spécification de QdS,
- assurer la **confiance** : permettre l'intégration de mécanismes assurant que les niveaux de QdS consommés sont bien ceux spécifiés,
- accepter l'**auto-configurabilité** : permettre l'évaluation dynamique des niveaux de QdS requis par un composant,
- permettre la **réutilisation** : permettre la réutilisation des composants de gestion de QdS dans différents systèmes.

Le type de QdS visé par l'architecture Qinna est la QdS liée à l'utilisation de ressources systèmes limitées dans un contexte embarqué (CPU, mémoire, réseau et batterie). Qinna n'a pas pour objectif de permettre la gestion de QdS telle que la sécurité, la tolérance aux fautes, la persistance ou le transactionnel.

3.1.2 Hypothèses

L'hypothèse de base de l'architecture Qinna est que l'ensemble du système devant intégrer Qinna est construit à base de composants. Cela implique une vue homogène en composants quelque soit le niveau du système (applicatif, service, système d'exploitation ou ressources).

De plus, l'architecture Qinna fait l'hypothèse qu'elle possède une connaissance globale des ressources disponibles du système sur lequel elle s'exécute. Cela signifie, en particulier, qu'une requête de service avec des contraintes de QdS ne peut être déléguée à un autre système.

En conséquence, l'architecture Qinna se base sur le constat suivant : le niveau de QdS de l'interface fournie d'un composant dépend du niveau de QdS présent sur ses interfaces requises et de son implémentation :

$$QdS(Itf_{fournie}) = F(QdS(Itf_{requises}), Implementation)$$

Afin de pouvoir gérer la QdS de l'interface fournie d'un composant A, il est nécessaire de pouvoir configurer l'implémentation de A et de gérer la QdS des composants fournissant les interfaces requises de A. Il existe un contrat de QdS implicite sur chacune des interfaces requises de A. Ce sont ces contrats que Qinna a pour objectif d'expliciter et de gérer.

Enfin, dans un premier temps, Qinna fait l'hypothèse d'un modèle de composants simple où une interface fournie par composant nécessite une seule gestion de QdS. Nous considérons alors que ses services ont des profils semblables de QdS requis.

3.1.3 Principes généraux de Qinna

L'architecture Qinna se base sur différents principes identifiés au chapitre 2. Ces principes sont :

1. **la séparation des politiques et mécanismes et la séparation des préoccupations.** Afin d'accroître la réutilisation de l'architecture, Qinna se base sur le principe de séparation des politiques et des mécanismes de gestion de QdS (tel que présent dans l'architecture de gestion de QdS de l'ISO), ainsi que sur le principe de séparation des préoccupations (tel que présent dans des modèles à composants pour les applications distribuées). Dans le cadre de Qinna, les préoccupations concernées sont les préoccupations fonctionnelles et celles de QdS.
2. **les contrats de QdS.** L'étude des architectures de gestion de QdS a permis de montrer que la mise en place de contrats de QdS permet de gérer dynamiquement la QdS d'un système. De plus, afin de pouvoir gérer dynamiquement ces contrats, trois principes de gestion de contrats de QdS ont été identifiés par [6]. Ces principes, présentés à la section 2.2.1.4 page 25, sont la spécification, l'initialisation (test, mapping et réservation) et la gestion (observation, maintenance et adaptation). L'architecture Qinna se base sur cette approche par contrats pour permettre la gestion dynamique de la QdS des composants.
3. **les conteneurs de QdS.** Les conteneurs présents dans les modèles de composants, tels que EJB, permettent d'identifier des points d'accès aux composants afin de gérer leur QdS. L'architecture Qinna se base sur ce principe afin d'identifier un point d'accès unique pour l'utilisation de composants nécessitant une gestion de la QdS.
4. **la maximisation des niveaux de QdS.** L'étude des architectures de gestion de QdS identifie les mécanismes

à mettre en place afin d'adapter dynamiquement les niveaux de QdS des applications. Ces adaptations sont réalisées dans une optique d'établissement de niveaux optimaux par rapport à un critère initial (par exemple, maximisation de l'utilisation de la ressource réseau). L'architecture Qinna, se base alors sur le principe de maximisation des niveaux de QdS des contrats en cours d'exécution par rapport au niveau spécifié initialement.

5. **l'identification des composants à l'exécution.** Au vu des architectures de gestion de QdS pour les applications à composants et afin d'assurer leur gestion dynamique de QdS, Qinna conserve la structuration en composants lors de l'exécution du système.
6. **l'efficacité**. Qinna se base sur le principe d'efficacité identifié dans les modèles de composants pour les systèmes embarqués en permettant l'implémentation de l'architecture par des langages de bas niveaux.
7. **l'indépendance vis-à-vis des politiques**. Le dernier principe guidant la définition de l'architecture est de ne pas présupposer de la politique de gestion de QdS utilisée.

Pour répondre aux trois derniers principes présentés, l'architecture Qinna doit s'appuyer sur un modèle de composants suffisamment générique et flexible. Pour ces raisons, l'architecture Qinna se base sur la définition de composants conformes au modèle Fractal et utilise les propriétés du framework à composants Think. En effet, Think fournit une vue homogène en composants du système, aussi bien au niveau ressources, qu'au niveau applicatif en passant par les niveaux services et système d'exploitation. De plus, il permet l'identification des composants en cours d'exécution du système et il se repose sur un modèle suffisamment flexible permettant d'être étendu. Enfin, Think répond au besoin d'efficacité en étant implémenté en langage C et assembleur.

3.1.4 Vue globale de Qinna

Afin de pouvoir gérer la QdS des composants, Qinna introduit des composants de gestion de contrats de QdS (cf. figure 3.1). Chaque composant, appelé Component, nécessitant une gestion de QdS sur ses interfaces devient un QoSComponent. Les contrats associés aux interfaces fournies de chaque QoSComponent sont initialisés et gérés par un QoSComponentBroker et un QoSComponentManager. Le QoSComponentBroker réalise le test d'admission ainsi que la réservation du QoSComponent pour un niveau de QdS donné sur ses interfaces fournies. Le QoSComponentManager implémente l'opération de mapping ainsi que les mécanismes d'adaptation et de maintenance liés aux contrats de QdS. Les appels à ces mécanismes sont réalisés suivant les politiques d'adaptation et de maintenance implémentées par le QoSDomain. De plus, le QoSDomain définit les politiques d'observation de contrats de QdS implémentées par les QoSComponentObservers.

Cette structuration de l'architecture permet de respecter les principes généraux énoncés par Qinna. En effet, la séparation des préoccupations fonctionnelles et de QdS est respectée : les QoSComponents implémentent les préoccupations fonctionnelles du système, alors que les autres composants de l'architecture (QoSComponentBrokers, QoSComponentManagers, QoSDomain et QoSComponentObservers) en implémentent les préoccupations de QdS. Le QoSDomain joue alors le rôle de conteneur en définissant un point d'entrée unique pour l'utilisation des QoSComponents. L'implémentation du QoSDomain, mais aussi les QoSComponentManagers, les QoSComponentBrokers et les QoSComponentObservers, implémentent les services de gestion de QdS fournis par le conteneur aux QoSComponents. Au sein de la préoccupation de QdS, le principe de séparation des politiques et des mécanismes est respecté : le QoSDomain implémente les politiques d'adaptation et de maintenance des contrats de QdS, tandis que les mécanismes correspondants sont implémentés par les QoSComponentManagers. Enfin, les

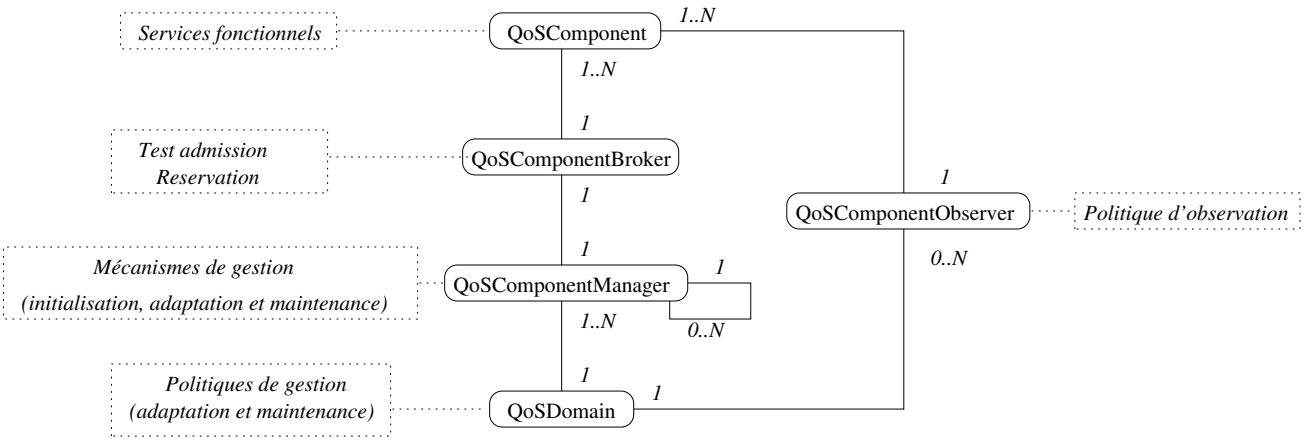

FIG. 3.1 – Vue globale de l'architecture Qinna.

trois principes de gestion de contrat de QdS sont repris : la spécification est capturée par les interfaces du QoSDomain (cf. tableau 3.1), le mapping est réalisé par les QoSComponentManagers et le test d'admission, ainsi que la réservation, sont réalisés par les QoSComponentBrokers. Comme précisé ci-avant, les aspects d'adaptation et de maintenance sont réalisés par les QoSComponentManagers et le QoSDomain, tandis l'aspect d'observation est traité par les QoSComponentObservers. L'opération de mapping est réalisée par les QoSComponentManagers car les mécanismes d'adaptation et de maintenance se basent sur cette opération. De plus, l'opération d'observation est réalisée par des composants dédiés car cette opération est optionnelle lors de la gestion de contrat. Il est alors aisé de la rajouter, de la supprimer ou de la reconfigurer.

3.1.5 Convention graphique

Les conventions graphiques utilisées dans la suite de ce rapport sont les suivantes (cf. figure 3.2) :

- un composant est représenté par :
  - un rectangle
  - un nom (par exemple, *Nom_Composant1* et *Nom_Composant2*)
  - des propriétés typées (par exemple, *a* de type *A* et *b* de type *B*)
- une interface est représentée par un nom (par exemple *i1* ou *i2*) et une arité (par exemple, *1* à *N* pour *i1*).
- une interface requise non liée est représentée par un demi cercle (par exemple *i2*),
- une interface fournie non liée est représentée par un cercle plein (par exemple *i1*),
- une liaison entre deux interfaces est représentée par une flèche (par exemple *L*). La base de la flèche représente l'interface requise (*i3*), tandis que la pointe représente l'interface fournie (*i4*).

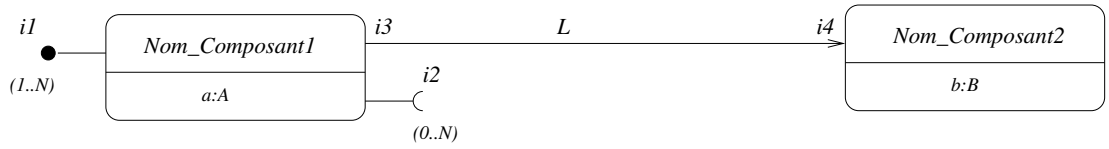

FIG. 3.2 – Représentation graphique du composant *Nom_Composant1* lié au composant *Nom_Composant2* par la liaison *L* et fournissant une interface non liée (*i1*) et requérant une interface non liée (*i2*).

3.2 Définition de l'architecture

La suite de cette partie présente plus précisément chaque élément constituant l'architecture Qinna. Qinna définit un ensemble de types de composants (cf. figure 3.1 page 36), un ensemble d'interfaces et un ensemble de types de données. De plus, Qinna définit un comportement dynamique décrivant les interactions entre les composants afin d'assurer la gestion des contrats de QdS. La suite de cette partie présente chacun de ces aspects.

3.2.1 Interfaces et types de composants

3.2.1.1 Interfaces Qinna

La définition d'un type de composant est réalisée au travers des interfaces requises et fournies portées par le composant. Cinq types de composants sont ainsi définis par Qinna : les QoSComponents, les QoSComponentBrokers, les QoSComponentManagers, les QoSComponentObservers, et les QoSDomains. L'ensemble des interfaces définies par Qinna est reporté dans le tableau 3.1 page 38.

3.2.1.2 QoSComponent

Un composant de type QoSComponent fournit, au moins, une interface fonctionnelle pour laquelle on souhaite gérer la QdS et peut en requérir plusieurs (cf. figure 3.3).

Afin de fournir un niveau de QdS donné, appelé QdS objectif (*qos_objective*) et de type `T_QoS`, un QoSComponent requiert un certain niveau de QdS sur ses interfaces fonctionnelles requises et doit être configuré via une contrainte locale (*local_constraint*) de type `T_LC`. De plus, un QoSComponent identifie un niveau de QdS courant (*qos_current*) de type `T_QoS`, identifiant le niveau de QdS réel qu'il fournit. Le QoSComponent est en charge de vérifier que le niveau de QdS courant soit inférieur au niveau de QdS objectif.

Un composant de type QoSComponent fournit l'interface `iQoSLevel`, permettant de manipuler la contrainte locale du QoSComponent, ainsi que le niveau de QdS objectif. Afin de pouvoir notifier du niveau réel de QdS fournit, identifié par le niveau de QdS courant, le QoSComponent requiert l'interface `iQoSInform` ou fournit l'interface `iQoSObserver`. Ces deux interfaces permettent de mettre en place deux types de notifications différentes : la première est initiée par le QoSComponent (mode *push*) lorsque l'interface requise `iQoSInform` est liée et que le QoSComponent est démarré, alors que la seconde est initiée par un autre composant (mode *pull*).

3.2.1.3 QoSComponentBroker

Le QoSComponentBroker est responsable du test d'admission et de la réservation des QoSComponents. Pour le test d'admission, il s'appuie sur une demande de niveau de QdS (niveau de QdS objectif) et une contrainte globale courante (de type `T_CG`) qu'il compare à une contrainte globale (de type `T_CG`). Si le test d'admission est validé la contrainte globale courante est mise à jour et la réservation du QoSComponent est effectuée. Celle-ci consiste à fixer la contrainte locale et le niveau de QdS objectif du QoSComponent. Une fois la réservation

Interfaces	Services
`iQoSLevel`	T_STATUS set(T_LC lc, T_QoS q_obj) T_QoS get()
`iQoSObserver`	T_QoS get()
`iQoSInform`	T_STATUS push(T_QoS q_current)
`iQoSBroker`	T_CID reserve(T_LC lc, T_QoS q_obj) T_STATUS free(T_CID cid) T_STATUS modify(T_CID cid, T_LC lc, T_QoS q_obj)
`iQoSBrokerAdministration`	T_STATUS set(T_GC gc) T_GC get()
`iQoSManager`	T_CT_MANAGER reserve(T_QoS q) T_STATUS free(T_CT_MANAGER ct) T_STATUS modify(T_CT_MANAGER ct,T_QoS q)
`iQoSAdapter`	T_STATUS degradeQoSLevel(T_CT_MANAGER ct) T_STATUS upgradeQoSLevel(T_CT_MANAGER ct)
`iQoSMaintener`	T_STATUS degrade(T_CT_MANAGER ct) T_STATUS upgrade(T_CT_MANAGER ct)
`iQoSManagerAdministration`	T_STATUS set(T_MT mt) T_MT get() T_STATUS set(T_S step)
`iQoSObserverLifeCycle`	T_STATUS start(T_CID cid, T_POL pol, T_QoS q_obj) T_STATUS pause(T_CID cid) T_STATUS stop(T_CID cid)
`iQoSObserverConfigure`	T_STATUS set(T_CID cid, T_QoS q_objective)
`iQoSDomain`	T_CID reserve(T_QoS qu, T_IMP imp) T_STATUS free(T_CID cid) T_STATUS modify(T_CID cid, T_QoS qu, T_IMP imp)
`iQoSObserverException`	T_STATUS exception(T_CID cid, T_QoS observed_q)
`iQoSDomainAdministration`	T_STATUS set(T_OR f_Order_Relation)

T_STATUS : Type abstrait pour une information de retour d'état d'un service (succès ou échec)
T_LC : Type abstrait pour une information *contraite locale*
T_GC : Type abstrait pour une information *contraite globale*
T_QoS : Type abstrait pour une information *niveau de QoS*
T_CID : Type abstrait pour une information *identifiant de composant*
T_CT_MANAGER : Type abstrait d'identification *d'un contrat des QoSComponentManagers*
T_MT : Type abstrait pour une information *table de mapping*
T_POL : Type abstrait pour une information *politique d'observation*
T_IMP : Type abstrait pour une information *importance*
T_S : Type abstrait pour une information *pas de maintenabilité*
T_OR : Type abstrait pour une information *relation d'ordre des niveaux d'importance*

TAB. 3.1 – Interfaces définies par l'architecture Qinna.

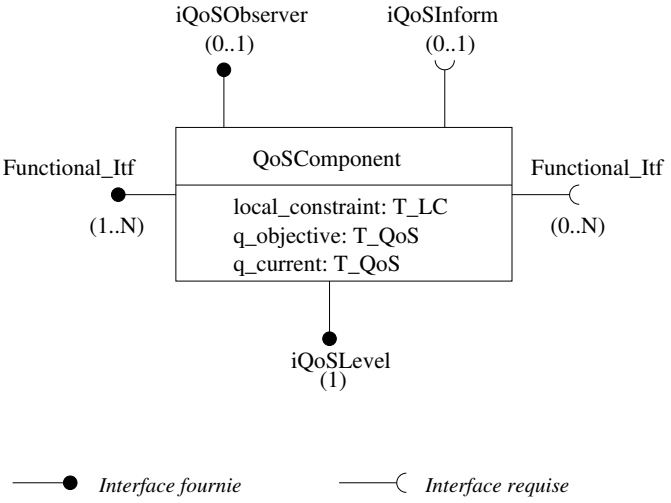

FIG. 3.3 – Représentation d'un QoSComponent.

réalisée, le QoSComponentBroker retourne une référence vers un QoSComponent. Étant donné que l'architecture Qinna se base sur le framework de composants Think, la référence d'un composant est donné par son identificateur au travers de l'interface *ComponentIdentity* (cf. section 2.1.5.4 page 21). Il existe un QoSComponentBroker par classe de QoSComponent.

Un composant de type QoSComponentBroker requiert une, ou plusieurs, interfaces `iQoSLevel` et fournit l'interface `iQoSBroker` (cf. figure 3.4). Cette interface permet de réserver, et de modifier, un QoSComponent avec un niveau de QdS objectif et une contrainte locale donnée. De plus, l'interface `iQoSBrokerAdministration` est fournie afin de fixer la contrainte globale du QoSComponentBroker.

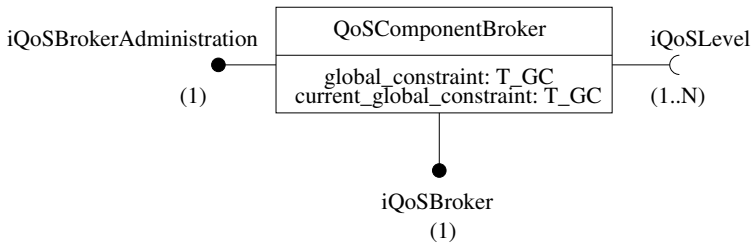

FIG. 3.4 – Représentation d'un QoSComponentBroker.

3.2.1.4 QoSComponentManager

Un composant de type QoSComponentManager est responsable du niveau de QdS fourni par l'interface fonctionnelle d'un QoSComponent. Du point de vue de la gestion de la QdS, l'accès à un QoSComponent se fait toujours par l'intermédiaire du QoSComponentBroker associé. Il existe donc un QoSComponentManager par classe de QoSComponent. Il possède trois rôles :

1. Initialiser, d'un point de vue QdS, l'exécution d'un QoSComponent. C'est-à-dire, qu'à partir d'une spécification précisant le niveau de QdS désiré, de type `T_QoS`, le QoSComponentManager la traduit à l'aide d'une table de mapping, de type `T_MT`, en une contrainte locale et un niveau de QdS objectif sur le QoSComponent

et en niveaux de QdS requis sur les interfaces requises. La table de mapping possède un attribut permettant de spécifier si les valeurs contenues dans la table sont fiables (attribut fixé à *reliable*) ou non fiables (attribut fixé à *unreliable*). Ensuite, le QoSComponentManager se base sur les QoSComponentManagers concernés afin d'obtenir les références des QoSComponents fournissant les services requis. Le QoSComponentManager établit alors un contrat de type `T_CT_MANAGER` comprenant :

(a) un identifiant de contrat unique ;

(b) l'identifiant du QoSComponent fourni par le QoSComponentBroker ;

(c) l'identifiant du QoSComponentObserver applicable pour le QoSComponent fourni par le QoSComponentBroker ;

(d) le niveau de QdS fourni identifié par la table de mapping et correspondant au niveau de QdS objectif du QoSComponent ;

(e) la liste des identifiants des contrats établis par les autres QoSComponentManagers. Cette liste représente les contrats établis pour les interfaces requises du QoSComponent.

De plus, le composant QoSComponentManager possède un ensemble d'opérateurs *T_QoSX translate(T_QoS q)*. Ces opérateurs permettent de traduire une spécification donnée, de type *T_QoS*, en une spécification compréhensible par la table de mapping, notée ici *T_QoSX*.

2. Implémenter les mécanismes permettant l'adaptation dynamique de la QdS d'un QoSComponent. L'adaptation de la QdS consiste à passer d'un niveau de QdS à un autre tels qu'ils sont définis et ordonnés par la table de mapping.

3. Implémenter les mécanismes de maintenabilité d'un QoSComponent. La maintenabilité consiste à modifier les données de la table de mapping : c'est-à-dire que pour un niveau de QdS donné, les valeurs de QdS requises sont ajustées suivant un pas, appelé *pas de maintenabilité* de type `T_S`. Il existe donc un pas de maintenabilité pour chaque interface requise identifiée par la table de mapping.

Afin de remplir ces rôles, un composant de type QoSComponentManager requiert une interface `iQoSBroker`, pour accéder aux QoSComponents, et zéro ou plusieurs interfaces `iQoSManager` (liées aux interfaces requises des QoSComponents gérées). Ces interfaces `iQoSManager` permettent d'obtenir les références des QoSComponents fournissant les interfaces requises fonctionnelles à un niveau de QoS donné. Un composant de type QoSComponentManager fournit une interface `iQoSManager` permettant de réserver un QoSComponent avec un niveau de QoS donné, une interface `iQoSAdapter` mettant en oeuvre les mécanismes d'adaptation, et une interface `iQoSMaintener` mettant en oeuvre les mécanismes de maintenabilité. De plus, il fournit une interface `iQoSManagerAdministration` permettant d'initialiser la table de mapping et de fixer le pas de maintenabilité. La représentation d'un QoSComponentManager est donnée à la figure 3.5.

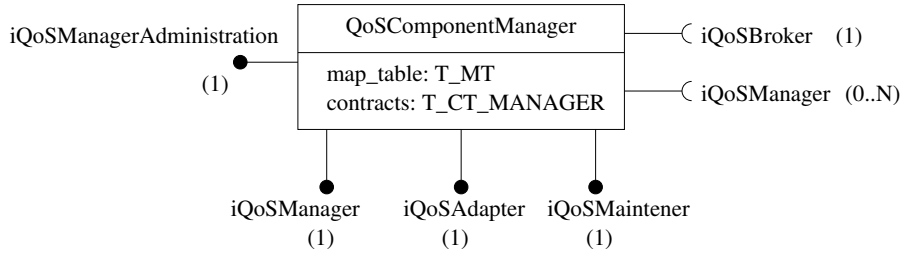

FIG. 3.5 – Représentation d'un QoSComponentManager.

3.2.1.5 QoSComponentObserver

Un composant de type QoSComponentObserver a pour rôle d'observer et de notifier des niveaux de QdS fournis par les interfaces fonctionnelles des QoSComponent lors de leur exécution. Il implémente donc les politiques d'observation de type `T_POL`. L'observation peut être soit initiée par le QoSComponentObserver (mode *pull*), en effectuant une demande au QoSComponent du niveau de QdS courant, soit initiée par le QoSComponent lui-même (mode *push*). Il existe un QoSComponentObserver par classe de QoSComponent.

Un composant de type QoSComponentObserver, (cf. figure 3.6), requiert l'interface `iQoSObserver` afin d'être informé du niveau courant de QdS lorsqu'il initie une observation. Dans le cas d'une observation initiée par le QoSComponent, le QoSComponentObserver fournit l'interface `iQoSInform` afin de pouvoir être appelé. De plus, il requiert l'interface `iQoSObserverException` afin de notifier une exception lorsque le niveau de QdS courant est différent de celui contractualisé. Afin d'être informé du niveau de QdS contractualisé du QoSComponent observé, il fournit l'interface `iQoSObserverConfigure`. Enfin, il fournit l'interface `iQoSObserverLifeCycle` permettant de gérer le cycle de vie de l'observation et de la notification (start, pause, stop). Cela signifie que même dans le cas d'une observation initiée par le QoSComponent, le QoSDomain ne peut être notifié que si le QoSComponentObserver a été démarré. Au démarrage de l'observation (start), la politique d'observation est passée au QoSComponentObserver. Si la politique est `NULL`, l'observation n'est pas initiée par le QoSComponentObserver, mais par le QoSComponent.

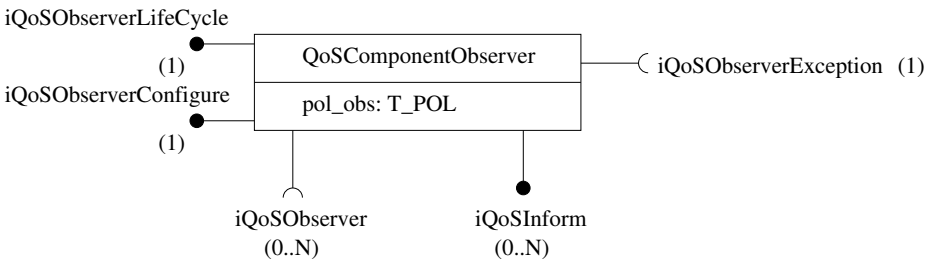

FIG. 3.6 – Représentation d'un QoSComponentObserver.

3.2.1.6 QoSDomain

Un composant de type QoSDomain est le composant de plus haut niveau de l'architecture. Il encapsule les composants QoSComponents, QoSComponentBrokers, QoSComponentObservers et QoSComponentManagers. Du point de vue utilisateur, il s'agit du point d'entrée unique pour toutes demandes de gestion de QoS.

Un composant de type QoSDomain définit les politiques d'adaptation et de maintenabilité. Ces politiques se basent sur une liste de contrats de type `T_CT_DOMAIN`. Chaque contrat lie un niveau d'importance, de type `T_IMP`, à un identifiant de contrat établi par un QoSComponentManager.

Afin d'établir les contrats, le QoSDomain fournit l'interface `iQoSDomain` permettant de réserver un QoSComponent avec un niveau de QdS et un niveau d'importance donné. Pour la gestion des contrats, le QoSDomain requiert les interfaces des QoSComponentManagers (`iQoSManager`, `iQoSAdapter`, `iQoSMaintener`). Les politiques d'observation sont initiées via l'interface requise `iQoSObserverLifeCycle` et les niveaux de QdS

contractualisés sont passer aux QoSComponentObservers via l'interface requise `iQoSObserverConfigure`. De plus, il fournit l'interface `iQoSObserverException` afin d'être informé des violations de contrats de QoS. Enfin, un composant QoSDomain fournit l'interface `iQoSDomainAdministration` permettant de configurer la relation d'ordre liée au type T_IMP. La figure 3.7 représente un composant de type QoSDomain.

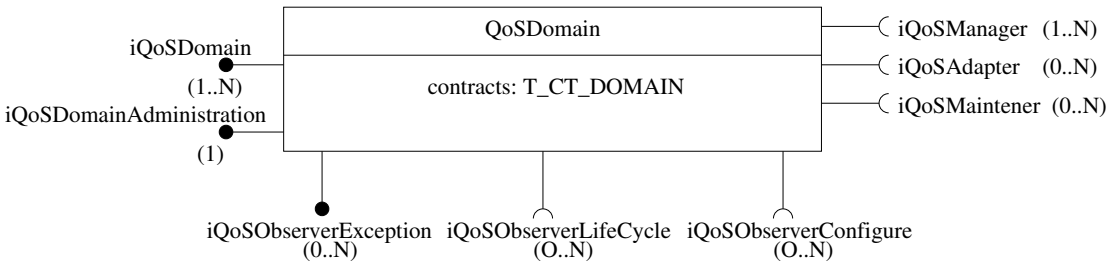

FIG. 3.7 – Représentation d'un QoSDomain.

3.2.2 Types de données

Lors des définitions des interfaces Qinna, ainsi que des différents types de composants, plusieurs types abstraits de données ont été utilisés. Nous définissons ici les contraintes portant sur ces types.

En premier lieu, le type `T_STATUS` possède les éléments `OK` et `KO` permettant d'informer, respectivement, du succès ou de l'échec d'un appel à un service.

Deuxièmement, il existe un élément par défaut pour les initialisations de données inconnues. A cet effet, les types `T_LC` (contrainte locale), `T_GC` (contrainte globale), `T_QoS` (niveau de QdS) et `T_IMP` (niveau d'importance) possèdent un élément `default`. De leur côté, les types `T_CT_ID` (identifiant de contrat) et `T_CID` (identifiant de QoSComponent) possèdent l'élément `NULL` signifiant l'absence de référence. De même, le type `T_POL` possède l'élément `NULL` signifiant l'absence de politique d'observation. De plus, le type `T_MT` (table de mapping) possède un élément `realiability` pouvant être fixé soit à `reliable` (table de mapping sûre), soit `unreliable` (table de mapping non sûre).

Pour réaliser des opérations sur les contraintes de QdS, il existe des opérateurs de :
- calcul pour la manipulation des niveaux de QdS :
  - pour la soustraction d'un niveau de QdS à une contrainte globale courante : `T_GC sub(T_GC gc, T_QoS q2)`
  - pour l'addition d'un niveau de QdS à une contrainte globale courante : `T_GC add(T_GC gc, T_QoS q2)`
- comparaison pour :
  - l'acceptation d'une contrainte globale courante : `bool compare( T_GC cg, T_GC cg_courant)`
  - la comparaison des niveaux de QdS objectif et courant : `bool compare( T_QoS q_objectif, T_QoS q_courant)`
  - la comparaison de deux niveaux d'importance : `bool compare (T_IMP i1, T_IMP i2)`
- mapping pour déterminer les niveaux de QdS requis pour un niveau de QdS fourni donné : `T_QoS2 map(T_QoS1 q)`

Pour la gestion des contrats, une donnée de type `T_CT_MANAGER` (contrats au niveau des QoSComponentManagers) est composée d'une donnée de type `T_CT_ID` (identifiant du contrat), de deux données de types `T_CID` (identifiant du QoSComponent et du QoSComponentObserver), d'une donnée de type `T_QoS` (niveau de QoS devant être fourni) et de zéro ou plusieurs données de types `T_CT_ID` (identifiant les contrats pour les QoSComponents fournissant les interfaces requises).

Enfin, une donnée de type `T_CT_DOMAIN` (contrats au niveau du QoSDomain) est composée d'une donnée de type `T_IMP` (niveau d'importance) et d'une donnée de type `T_CT_MANAGER` (identifiant du contrat de QoS).

3.2.3 Comportement dynamique

Après avoir présenté la structure de l'architecture Qinna, nous présentons maintenant son comportement dynamique. A cette fin, nous utilisons les diagrammes de séquence afin d'illustrer les interactions entre les différents composants lors des principales opérations de gestion des contrats. Les principales opérations à considérer sont la mise en place d'un contrat, le suivi des contrats et la mise à jour des données relatives aux contrats. La première opération est liée à un changement dans la structure par l'activation d'un nouveau composant. La deuxième opération correspond aux adaptations dynamiques des contrats en cours suite à l'arrêt ou à l'activation d'un nouveau composant, ou encore à un changement de l'environnement. Enfin, la troisième opération est réalisée lorsqu'un contrat n'est pas respecté. Il est à noter que toutes ces opérations doivent être menées de manière atomique afin de préserver la cohérence globale du système. Les scénarios présentés sont donc exclusifs les uns aux autres. De plus, les exécutions des services définis par l'architecture sont considérées comme plus prioritaires que les composants dont Qinna gère la QdS. Les paragraphes suivants détaillent les trois opérations de gestion de contrat.

3.2.3.1 Mise en place de contrat

Une demande d'activation d'un QoSComponent aboutit, ou non, à la mise en place d'un contrat de QdS. La mise en place de ce contrat peut être :

- soit une réussite totale lorsque tous les tests d'admission impliqués par le contrat sont validés,
- soit une réussite après adaptation (dégradation) des contrats en cours d'exécution, afin de permettre la validation des tests d'admission pour le nouveau composant,
- soit un échec lorsque l'un des tests d'admission n'est pas validé et qu'il n'y a plus de contrat en cours d'exécution à adapter.

La figure 3.8 schématise l'enchaînement des opérations réalisées lors de la réussite totale de la mise en place d'un contrat lors de l'activation d'un composant QoSX utilisant un composant QoSY. La demande d'activation d'un composant QoSX au niveau de QdS *q* et d'importance *imp*, passe d'abord par une demande au QoSDomain (service *reserve* de l'interface *iQoSDomain*), qui fait lui-même appel au QoSXManager chargé de la gestion de la QdS de QoSX (service *reserve* de l'interface *iQoSManager*). Ce dernier, après examen de la table de mapping, réserve le composant QoSY requis par QoSX avec le niveau de QdS nécessaire, noté *qy* (via le QoSYManager). Au retour du contrat établi par le QoSYManager, noté *id_contrat_QoSY*, le QoSXManager effectue une demande auprès du QoSXBroker d'un composant QoSX avec une contrainte locale donnée par la table de mapping, notée *cl_x*, et le niveau de QdS objectif *q_x*. Si le QoSXBroker valide le test d'admission, la référence de QoSX est retournée à QoSXManager après avoir fixé sa contrainte locale et le niveau de QdS objectif. Le QoSXManager est alors en charge de lier le composant QoSX au composant QoSY. Enfin, le QoSXManager établit le contrat de QdS

identifiant le composant QoSX et le sous contrat requis (*id_contrat_QoSY*), puis le QoSDomain enregistre à son tour le contrat en l'enrichissant du niveau d'importance (*imp*).

Le scénario générique d'établissement d'un contrat correspond au cas où QoSX utilise N QoSComponents, notés QoSY1 à QoSYN. Le QoSXManager effectue alors un appel au service *reserve* pour chaque QoSYiManager avec $i \in [1; N]$. L'ordre des appels aux services *reserve* des QoSYiManagers et du QoSXBroker n'est pas fixé par Qinna.

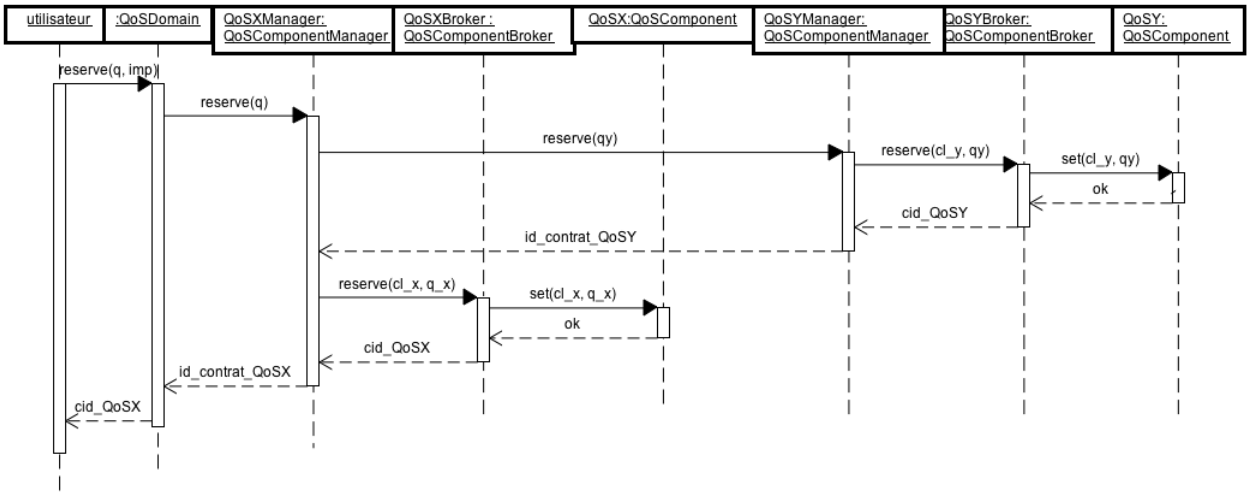

FIG. 3.8 – Réussite lors de la mise en place du contrat lié à l'activation du composant QoSX utilisant un composant QoSY.

Lorsqu'un des tests d'admission au niveau des QoSComponentBrokers (QoSXBroker ou QoSYiBroker) n'est pas validé, les sous-contrats précédemment enregistrés pour QoSX doivent être annulés via le service *free* des QoSYiManagers. Ensuite, le QoSDomain en est informé et adapte les contrats en cours d'exécution (cf. adaptation de contrat page 45), puis re-émet la demande d'activation du composant QoSX au QoSXManager.

Lorsqu'il n'y a plus de contrat à adapter et que les tests d'admission ne sont pas validés, il y a donc échec de la mise en place du contrat. Lorsqu'un échec intervient, il est à la charge de l'utilisateur de re-émettre une demande d'activation avec un niveau de QdS moins élevé. En effet, Qinna adapte uniquement les niveaux de QdS des contrats en cours d'exécution, mais pas celui du contrat en cours de mise en place. Cela signifie que lorsqu'un contrat est mis en place, son niveau de QdS est égal à celui spécifié lors de la demande d'activation (jusqu'à ce qu'une autre demande d'activation de QoSComponent intervienne).

L'opération duale à la mise en place d'un contrat est l'annulation d'un contrat. Il n'existe qu'un seul scénario d'annulation car cette dernière est toujours possible. L'annulation d'un contrat aboutit à la libération des QoSComponents impliqués par le contrat et à une adaptation des contrats actifs restants afin de maximiser les niveaux de QdS. Une annulation de contrat se produit lorsqu'une demande de désactivation de QoSComponent est effectuée : une demande de désactivation est effectuée au QoSDomain (service *free* de l'interface *iQoSDomain*), puis transmise au QoSComponentManager concerné (service *free* de l'interface *iQoSManager*). Celui-ci libère le QoSComponent grâce à son QoSComponentBroker (service *free* de l'interface *iQoSBroker*), puis transmet les demandes de libération aux QoSComponentManagers identifiés par le contrat. Le contrat au niveau des QoSCom-

ponentManagers et du QoSDomain est alors annulé. Puis, le QoSDomain améliore les contrats restant en cours d'exécution, ce qui revient à une adaptation de contrat. Les scénarios d'adaptation sont maintenant présentés.

3.2.3.2 Adaptation de contrat

Lorsqu'un contrat ne peut être mis en place ou lorsqu'un contrat est annulé (cf. paragraphe précédent), l'architecture Qinna a la possibilité d'adapter les niveaux de QdS des contrats en cours d'exécution. L'adaptation de contrats peut amener soit :

- à une dégradation de contrats (baisse du niveau de QdS),
- à une amélioration de contrats (augmentation du niveau de QdS).

Lorsqu'un test d'admission ne peut être validé, le QoSDomain dégrade les contrats en cours d'exécution selon les niveaux d'importance jusqu'à trouver une réponse acceptable. La figure 3.9 représente les opérations menées successivement lors de la dégradation du contrat lié au composant QoSX utilisant le composant QoSY. Le QoSDomain recherche, tout d'abord, le contrat à dégrader suivant sa politique d'adaptation, puis en effectue la demande au QoSComponentManager concerné (service *degradeQoSLevel* de l'interface *iQoSAdapter*). Le QoSXManager détermine alors le niveau de QdS directement inférieur au niveau courant, noté *qinf*, grâce à la table de mapping, puis effectue les demandes de modification auprès du QoSYManager (service *modify* de *iQoSManager*) et de QoSXBroker (service *modify* de *iQoSBroker*). Enfin, les contrats au niveau de QoSXManager et QoSYManager sont mis à jour afin d'enregistrer les nouveaux niveaux de QdS fournis par QoSX et QoSY.

Le scénario générique de dégradation d'un contrat correspond au cas où QoSX requiert N QoSComponents (notés de QoS1 à QoSN). Lors d'une demande de dégradation, le QoSXManager effectue un appel de dégradation à chaque QoSiManager avec $i \in [1; N]$. L'ordre des appels aux QoSiManagers et au QoSXBroker n'est pas fixé par Qinna. De plus, il est à noter que lors des dégradations des contrats les moins prioritaires, l'opération de dégradation peut aboutir à l'annulation des contrats.

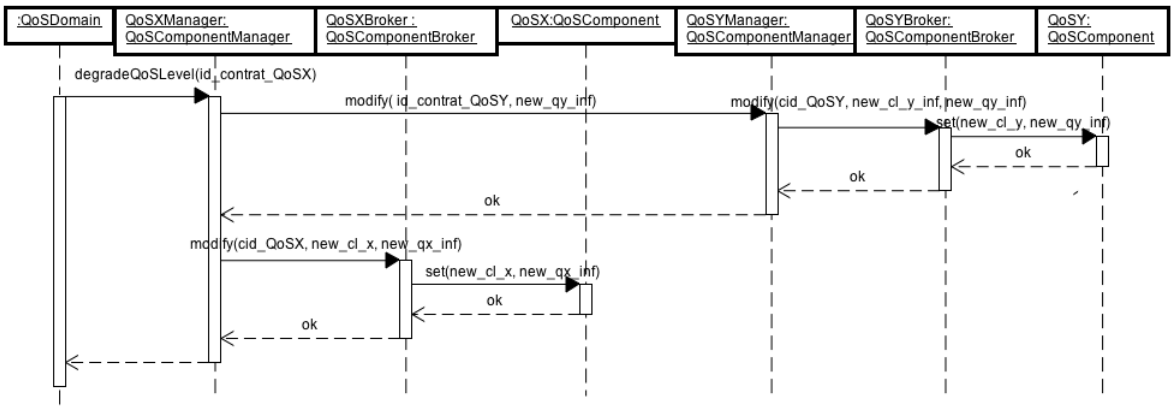

FIG. 3.9 – Dégradation du contrat lié au composant QoSX utilisant le composant QoSY.

Lorsqu'un composant est désactivé, le QoSDomain tente d'augmenter la QdS des contrats en cours d'exécution afin de maximiser les niveaux des contrats restants. Le choix, du ou des contrats augmentés, dépend de la politique d'adaptation du QoSDomain et se base sur les niveaux d'importance des contrats. L'amélioration d'un contrat de QdS peut aboutir soit à une réussite (le nouveau niveau de QdS est supérieur à l'ancien), soit à un échec (le niveau

de QdS reste inchangé). La figure 3.10 représente l'enchaînement des opérations lors de l'amélioration réussie du contrat lié au composant QoSX utilisant QoSY. La demande d'amélioration du contrat de QoSX est initiée par le

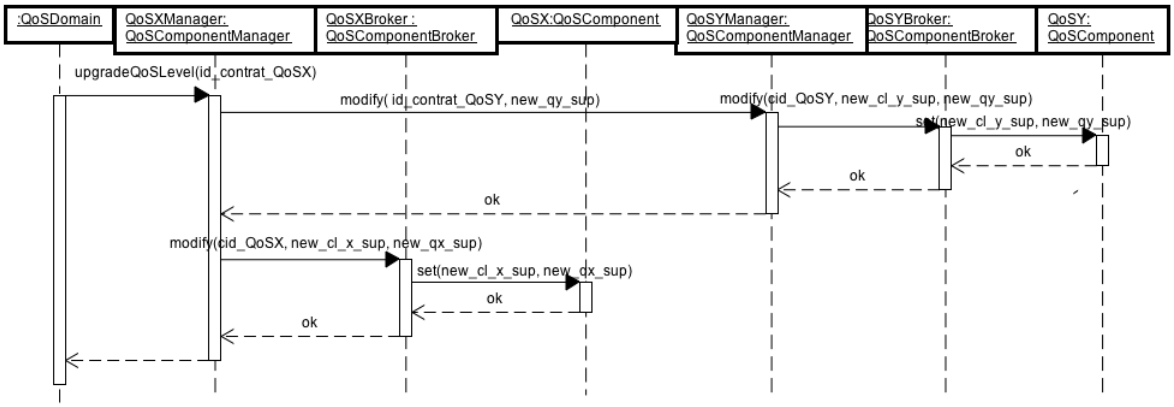

FIG. 3.10 – Amélioration réussie du contrat lié au composant QoSX utilisant le composant QoSY.

QoSDomain (service *upgradeQoSLevel* de l'interface *iQoSAdapter*) d'après sa politique d'adaptation. La demande est transmise au QoSXManager qui évalue, grâce à la table de mapping, le niveau de QdS immédiatement supérieur au niveau courant, noté *qsup*. Puis, il effectue les demandes de modifications correspondantes au QoSYManager (service *modify* de l'interface *iQoSManager*) et au QoSXBroker (service *modify* de l'interface *iQoSBroker*). Enfin, les contrats aux niveaux de QoSXManager et QoSYManager sont mis à jour afin de prendre en compte le nouveau niveau de QdS.

Lors de l'amélioration d'un contrat, si un des tests d'admission réalisé par le QoSXBroker ou le QoSYBroker n'est pas validé, l'amélioration est un échec. Le QoSDomain tente alors d'améliorer un autre contrat. Il est cependant nécessaire d'annuler les modifications des sous-contrats déjà enregistrées. La figure 3.11 représente les opérations réalisées lors de l'échec de l'amélioration du contrat d'un composant QoSX utilisant un composant QoSY. Nous considérons, dans cet exemple, que l'échec est dû à la non validation du test d'admission par le QoSXBroker. Avant d'informer le QoSDomain de l'échec, il est donc nécessaire de rétablir le niveau de QdS initial sur le contrat de QoSY, noté *qy*.

Dans le cas générique où QoSX requiert N QoSComponents (notés de QoS1 à QoSN), le QoSXManager effectue les demandes d'amélioration auprès de chaque QoSiManager avec $i \in [1; N]$. Comme pour les scénarios précédents, l'ordre des appels n'est pas fixé par Qinna. De plus, lors d'un échec d'amélioration, l'ordre des appels d'annulation des modifications n'est pas non plus fixé.

3.2.3.3 Mise à jour dynamique des données d'un contrat

La mise à jour des données d'un contrat peut intervenir dans deux cas : soit la table de mapping possède au moins une de ses données fixée à *default*, soit la table de mapping est spécifiée comme *unreliable*. Dans les deux cas, un QoSComponentObserver est mis en place afin d'observer le niveau de QdS fournit par le QoSComponent lors de son exécution et d'en notifier le QoSDomain. L'observation réalisée par le QoSComponentObserver peut être initiée soit par le QoSComponentObserver, soit par le QoSComponent lui-même.

La figure 3.12 schématise l'enchaînement des opérations lors de la notification d'une violation du contrat de

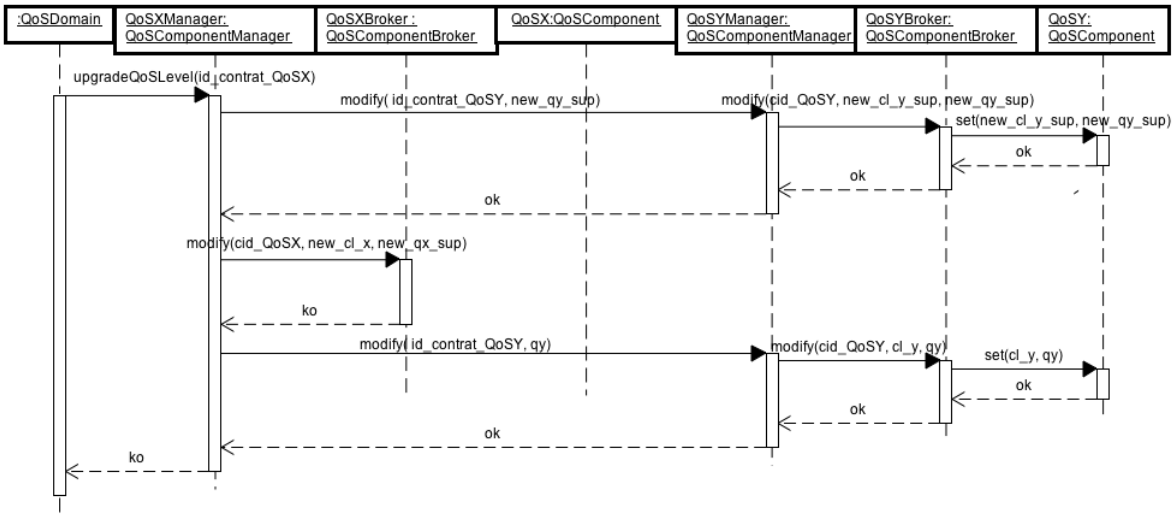

FIG. 3.11 – Échec de l'amélioration du contrat lié au composant QoSX utilisant le composant QoSY.

QoSX lorsque le QosXObserver initie l'observation à travers une politique, notée *pol_obs*, différente de *NULL*. Le QoSDomain informe tout d'abord le QoSXObserver du niveau de QoS contractualisé par le composant QoSX

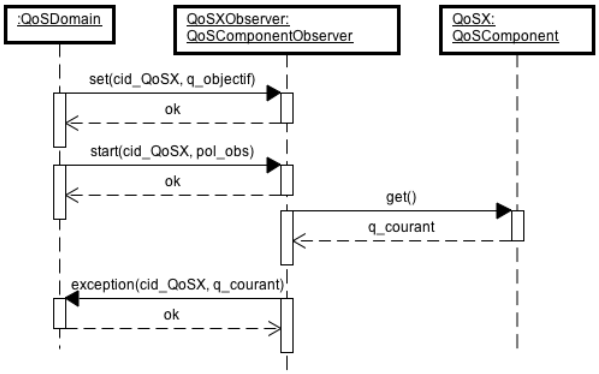

FIG. 3.12 – Notification au QoSDomain de la violation du contrat lié à QoSX, dans le cas d'une observation initiée par QoSXObserver.

(service *set* de l'interface *iQoSObserverConfigure*). Puis, lors de l'activation de QoSX, il démarre l'observation de QoSX par QoSXObserver (service *start* de l'interface *iQoSObserverLifeCycle*). Celui-ci observe le niveau de QdS fournit par QoSX (service *get* de l'interface *iQoSObserver*) suivant sa politique d'observation, puis notifie le QoSDomain de la violation de contrat (service *exception* de l'interface *iQoSObserverException*).

La figure 3.13 schématise l'enchaînement des opérations lors de la notification d'une violation du contrat de QoSX lorsque l'observation est initiée par QoSX (*pol_obs=NULL*). Dans le cas où la politique d'observation est égale à *NULL*, le QoSXObserver, après avoir été configuré (service *set* de l'interface *iQoSObserverConfigure*) et démarré (service *start* de l'interface *iQoSObserverLifeCycle*) par le QoSDomain , est en attente d'un appel de QoSX sur l'interface *iQoSInform* afin d'être informé du niveau réel de QdS fourni. Lors de cet appel, si il y a violation du contrat de QoSX, le QoSXObserver peut alors en informer le QoSDomain (service *exception* de l'interface *iQoSObserverException*).

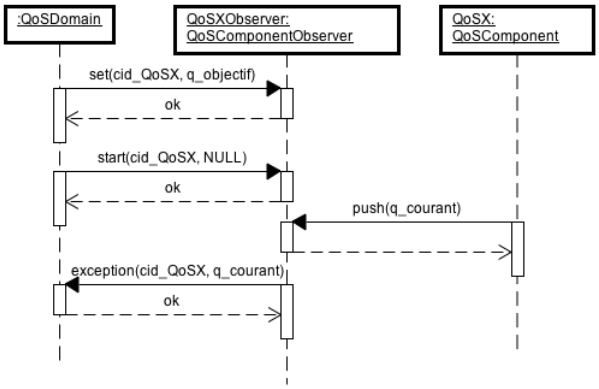

FIG. 3.13 – Notification au QoSDomain de la violation du contrat lié à QoSX, dans le cas d'une observation initiée par QoSX.

A la suite d'une notification de violation de contrat, le QoSDomain déclenche sa politique de maintenabilité afin d'adapter les niveaux de QdS requis par QoSX. La politique de maintenabilité peut aboutir soit :

– à une augmentation des niveaux de QdS requis,
– à une diminution des niveaux de QdS requis.

La figure 3.14 représente la suite d'opérations réalisées lorsque la politique de maintenabilité aboutit à une augmentation réussie des niveaux de QdS requis pour le composant QoSX. La décision est alors transmise au QoSXManager (service *upgrade* de *iQoSMaintener*) qui augmente d'un pas de maintenance les niveaux de QdS requis identifiés par le contrat. Il est à la charge du service *upgrade* d'identifier les niveaux de QdS à augmenter. Ensuite, le nouveau niveau de QdS requis sur QoSY est modifié par le biais du service *modify* du composant QoSYManager. Par rapport à la mise en oeuvre des mécanismes d'adaptation étudiés précédemment, le niveau de QdS fourni par QoSX n'est donc pas modifié : en effet, l'objectif ici est uniquement d'augmenter les niveaux de QdS requis par QoSX. Il est à noter que l'amélioration des niveaux de QdS peut aboutir à un échec dans le cas où

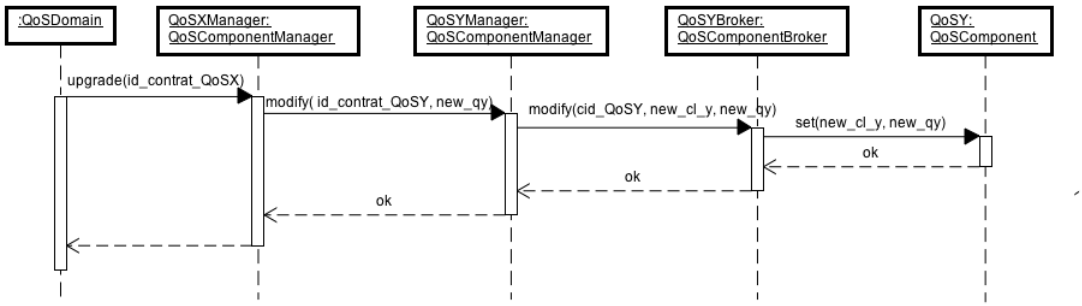

FIG. 3.14 – Mise à jour des données du contrat lié au composant QoSX utilisant le composant QoSY.

le test d'admission réalisé par le QoSYBroker n'est pas validé lors de la demande de modification du niveau fourni par QoSY. Cela signifie alors que le niveau de QdS contractualisé par QoSX ne peut pas être fourni. Dans ce cas, le QoSDomain dégrade le niveau de QdS fourni par QoSX.

Enfin, dans le cas où la politique de maintenabilité aboutit à une diminution des niveaux de QdS requis par QoSX, l'enchaînement des opérations est le même. Le QoSDomain demande au QoSXManager de diminuer les niveaux de QdS requis par QoSX (service *degrade* de l'interface *iQoSMaintener*), qui diminue alors les niveaux de QdS de la valeur du pas de maintenabilité. Il est à noter que la diminution des niveaux de QdS requis ne peut

pas aboutir à un échec.

3.3 Mise en place de Qinna

Après avoir défini l'architecture Qinna du point de vue statique et dynamique, nous présentons, dans cette partie, une méthodologie de mise en place dans un système à composants. Nous détaillons, dans un premier temps, les principes de mise en place, puis nous les illustrons au travers d'un exemple.

3.3.1 Principes

Les principes de mise en place de Qinna ont été établis suite aux expériences de mise en oeuvre dans un contexte de systèmes embarqués mobiles de type assistant personnel. Ces principes s'appuient sur une structuration classique des systèmes en quatre couches : la couche *ressources*, la couche *système d'exploitation*, la couche *services* et la couche *applications*. La couche *ressources* réifie le matériel de la plateforme (CPU, mémoire, réseau, batterie, etc.). La couche *système d'exploitation* fournit les services permettant l'utilisation de ces ressources (par exemple, les processus, les threads ou les sockets). Enfin, la couche *services* identifie les bibliothèques de fonctions utilisées par les applications (par exemple, les fonctions de décodage du son et de la vidéo), tandis que la couche *applications* contient les applications manipulées directement par l'utilisateur du système (par exemple, un lecteur vidéo ou de musique). A partir d'un tel système, la mise en place de Qinna se décompose en cinq étapes successives (cf. figure 3.15) :

1. D'une part, il faut identifier les QoSComponents du système. Un QoSComponent est défini comme un composant fournissant une interface dont on souhaite gérer la QdS. Il existe différentes sortes de QoSComponents. Par principe, dans un système embarqué, tous les composants réifiant les accès aux ressources du système sont des QoSComponents. Un composant gestionnaire de CPU (ou communément appelé ordonnanceur), gestionnaire de mémoire, gestionnaire de réseau et gestionnaire de batterie est donc un QoSComponent. D'autre part, les composants d'exécution, généralement fournis par les systèmes d'exploitation, tels que les threads ou les sockets, sont eux aussi des QoSComponents. De plus, les composants de services utilisant les composants d'exécution sont eux-mêmes des QoSComponents. Dans le cadre des systèmes embarqués ouverts, les composants de services sont des composants d'encodage et de décodage audio et vidéo ou d'afficheurs vidéo.

   Une fois les QoSComponents identifiés, il est nécessaire d'expliciter leur contrainte locale et leur niveau de QdS objectif. Une contrainte locale est la propriété configurable du QoSComponent permettant de le configurer pour un niveau de QdS objectif donné. Cette configuration permet de fixer les paramètres de l'algorithme implémenté par le QoSComponent. Dans le cas des composants applicatifs ou des composants de services tel qu'un décodeur le niveau de QdS objectif quantifie le niveau de QdS du composant (par exemple, un nombre d'images par seconde ou une fréquence de décodage), tandis que la contrainte locale fixe l'algorithme de décodage utilisé (si plusieurs implémentations sont possibles) ou les paramètres de l'algorithme. Il est à noter que ce type de composants peut ne proposer qu'un seul choix pour la valeur de la contrainte locale. Cela signifie alors qu'il n'existe qu'une seule implémentation possible et que celle-ci n'est pas paramétrable. C'est par exemple le cas des composants d'exécution. Les niveaux de QdS objectifs indiquent donc si le composant peut être exécuté ou pas. Typiquement, le niveau de QdS objectif à deux valeurs : ACTIF (pour signifier que le thread ou la socket peut être ordonnancé) et INACTIF (pour signifier que le

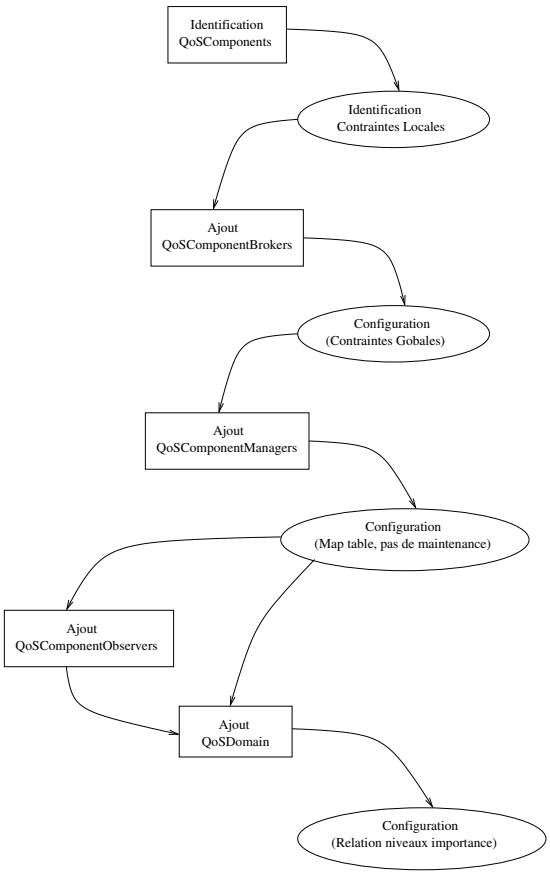

FIG. 3.15 – Étapes de mise en place de Qinna.

thread ou la socket ne peut pas être ordonnancé). Les contraintes locales, elles, configurent les composants d'exécution afin de pouvoir s'exécuter. Dans le cas d'un composant thread, elles spécifient, par exemple, un temps d'accès au CPU (sous forme de pourcentage, de MIPS, du couple délai d'exécution/période) ou la priorité du thread, tandis que dans le cas d'un composant socket, elles expriment le temps d'accès au réseau. Dans le cas des composants ressources, tels que la mémoire, la contrainte locale et le niveau de QdS objectif sont identiques et identifient la quantité de mémoire pouvant être utilisée au travers du composant. De même, les contraintes locales et les niveaux de QdS objectifs des composants gestionnaires de CPU, de réseau et de la batterie sont identiques et identifient les niveaux courants d'utilisation de chaque ressource.

2. A chaque classe de QoSComponents identifiée à l'étape précédente est associé un QoSComponentBroker. Les QoSComponentBrokers doivent être configurés via leurs contraintes globales au travers de l'interface *iQoSBrokerAdministration*. Les contraintes globales identifient le niveau de QdS maximal pouvant être fourni par une classe de QoSComponents, alors que les contraintes globales courantes identifient le niveau de QdS contractualisé. Dans le cas des QoSComponents ressources, les contraintes globales identifient la quantité maximale de ressource pouvant être allouée : dans le cas du QoSComponentBroker mémoire, la contrainte globale est la taille de la mémoire, alors que dans le cas des QoSComponentBrokers gestionnaire de CPU, de réseau et de batterie, il s'agit de la charge maximale acceptée. Dans le cas des QoSComponentBrokers associés aux threads et aux sockets, la contrainte globale identifie le nombre maximal d'instances de threads et de sockets pouvant s'exécuter simultanément, c'est-à-dire le nombre maximal de thread ayant un niveau de QdS objectif fixé à ACTIF. De même, dans le cas des QoSComponentBrokers associés aux

composants services et applicatifs, la contrainte globale représente un nombre maximal d'instances : ce nombre maximal peut être soit un nombre absolu (par exemple : 3 instances pour le composant vidéo), soit un nombre relatif aux niveaux de QdS (par exemple : 2 instances de niveau bon et 1 de niveau mauvais).

Les QoSComponentBrokers configurent ensuite leurs opérateurs d'acceptation de niveau de QdS objectif permettant de réaliser le test d'admission. Le test d'admission additionne le niveau de QdS objectif à la contrainte globale courante, puis compare le résultat à la contrainte globale ((*T_STATUS compare(T_CG cg, add(T_CG cg_current, T_QoS q_objective)*)). Nous nous intéressons alors plus particulièrement à l'opérateur de comparaison. Dans le cas des composants applicatifs ou de services, l'opérateur est différent suivant la contrainte globale :

- si la contrainte globale est un nombre absolu, il y a uniquement prise en compte du nombre de réservations de QoSComponent effectuées.
- si la contrainte globale est un nombre relatif, l'opérateur prend en compte le nombre de réservations de QoSComponents effectuées, ainsi que le niveau de QdS objectif de ceux-ci.

Dans le cas des composants d'exécution (threads ou sockets), l'opérateur se comporte comme dans le cas d'un nombre absolu afin de comparer que le nombre de réservations est inférieur à la contrainte globale identifiant le nombre maximal de composants pouvant être ordonnancés. Enfin, dans le cas des composants ressources, l'opérateur compare le niveau de QdS objectif et la valeur de la contrainte globale courante à la valeur de la contrainte globale.

Une fois le test d'admission validé, il est à noter que pour les composants ressources de type gestionnaire de CPU ou de réseau, il y a duplication de l'information entre le QoSComponentBroker et le QoSComponent. En effet, le niveau de QdS objectif est égal à la valeur de la contrainte globale courante. L'un permet de réaliser l'observation du QoSComponent, tandis que l'autre permet de réaliser le test d'admission.

3. A chaque QoSComponentBroker mis en place, il est nécessaire d'associer un QoSComponentManager. Les QoSComponentManagers doivent être, eux aussi, configurés. La configuration d'un QoSComponentManager est réalisée en trois étapes :
   - initialisation de la table de mapping. La table de mapping permet de faire la correspondance entre un niveau de QdS fourni par un QoSComponent et les niveaux de QdS requis ainsi que de la contrainte locale correspondante : elle implémente l'opérateur *T_QoS2 map(T_QoS1 q)*. Dans le cas des composants applicatifs et services, la table de mapping effectue le lien entre le niveau de QdS fourni par le composant et les niveaux de QdS requis ainsi que sa contrainte locale. Dans le cas des composants ressources, la table de mapping identifie uniquement la contrainte locale nécessaire pour un niveau de QdS donné car ces composants n'ont pas d'interface requise avec des besoins de gestion de QdS. Par exemple, la table de mapping du composant vidéo indique que pour un niveau de QdS bon (niveau de QdS fourni), le composant vidéo doit requérir 25 images par seconde du composant décodeur (niveau de QdS requis) et doit afficher les images en plein écran (contrainte locale). Pour sa part, la table de mapping du composant décodeur indique que pour fournir 25 images/secondes, il requiert un thread utilisant 35% du CPU, 50 Ko de mémoire et que sa contrainte locale doit être fixée à MPEG4. Enfin, la table de mapping du thread indique que pour fournir un thread utilisant 35% du CPU, il requiert un thread et un gestionnaire de CPU l'ordonnançant à 35%. Les données de la table de mapping sont souvent difficiles à déterminer a priori. C'est pourquoi Qinna laisse la possibilité de les spécifier par *default* et de mettre en oeuvre les politiques de maintenabilité. Cet aspect de Qinna fait l'objet d'une section particulière dans le chapitre suivant (cf. section 4.2.2 page 81).
   - initialisation des pas de maintenance. Les pas de maintenance permettent de configurer la politique de maintenance. Étant donné que la politique de maintenance a pour objectif de faire varier les niveaux de QdS requis pour un niveau de QdS fourni donné, les pas de maintenance ne sont présents que lorsque

le QoSComponent a des dépendances vers d'autres QoSComponents. Il existe donc un pas de maintenance différent pour chaque dépendance identifiée dans la table de mapping. Par exemple, si le pas de maintenance de la table de mapping du décodeur relatif au thread est fixé à 3%, alors, lors de l'appel des mécanismes de maintenance, la variation du niveau de QdS requit sur le thread se fera par pas de 3%.

– enfin il est nécessaire de lier le QoSComponentManager aux autres QoSComponentManagers par la même relation de dépendance existante au niveau des QoSComponents.

4. Si les valeurs de la table de mapping sont inconnues, elles sont initialisée à *default*. De plus, une table de mapping doit être caractérisée comme étant soit fiable (*reliable*), c'est-à-dire que les données sont connues et justes, soit non fiable (*unreliable*). Ce dernier cas est utile lorsque les données de QdS sont des valeurs approximatives, dans le cas où des mesures ont été effectuées sur des plateformes similaires, ou inconnues, dans le cas où aucune mesure n'a été effectuée. Une table de mapping comprenant, au moins, une donnée fixée à *default* est considérée comme non fiable. Lorsque la table de mapping est non fiable, il est nécessaire d'associer un QoSComponentObserver à la classe de QoSComponents correspondante. Le QoSComponentObserver configure alors l'opérateur de comparaison des niveaux de QdS (*bool compare(T_QoS q1, T_QoS q2)*).
5. La dernière étape de mise en place de l'architecture est d'intégrer un composant QoSDomain. La configuration du QoSDomain consiste à définir la relation d'ordre totale existant entre les différents niveaux d'importance, c'est-à-dire à définir l'opérateur *bool compare(T_IMP i1, T_IMP i2)*. Les relations d'ordre peuvent être soit fixes (par exemple : *niveau 1 est plus important que niveau 2*), soit dépendantes d'un ou plusieurs paramètres. Les paramètres peuvent être d'ordre temporel (par exemple : *niveau 1 est plus important que niveau 2 entre 8 heures et 18 heures et inversement entre 18 heures et 8 heures*) ou contextuel (par exemple : *niveau 1 est plus important que niveau 2 au bureau et inversement à la maison*).

3.3.2 Illustration

Nous illustrons ici les principes de mise en place énoncés au paragraphe précédent sur un système à composants représenté à la figure 3.16.

3.3.2.1 Description du système initial

Le système considéré est composé d'un composant d'interface utilisateur (UI) permettant de contrôler (start et stop) l'exécution d'un composant VIDEO. Afin de pouvoir s'exécuter, le composant Vidéo requiert un thread (THREAD1), de la mémoire (MEM1) et des services fournis par le composant DECODEUR. Ce dernier requiert, lui-même, un thread (THREAD2) et de la mémoire (MEM2) pour s'exécuter. Les threads sont ordonnancés par un ordonnanceur (ORDO) pouvant pondérer l'exécution d'un thread par rapport à un autre. Enfin, les threads, ainsi que l'ordonnanceur, requièrent, eux aussi, de la mémoire (MEM3, MEM4 et MEM5) lors de leur exécution. La mémoire considérée dans cet exemple est uniquement celle utilisée lors de l'allocation dynamique de données car l'on considère que les composants ont déjà été installés (la vérification de la place mémoire des composants a été effectuée).

Deux niveaux de QdS peuvent être fournis par le composant VIDEO. Le niveau BON requiert un thread à 20% du CPU, de 30 Ko de mémoire et d'un décodeur fournissant 25 images par seconde. Le niveau MAUVAIS requiert, quant à lui, un thread à 20% du CPU, 30 ko de mémoire et d'un décodeur fournissant 10 images par seconde. De

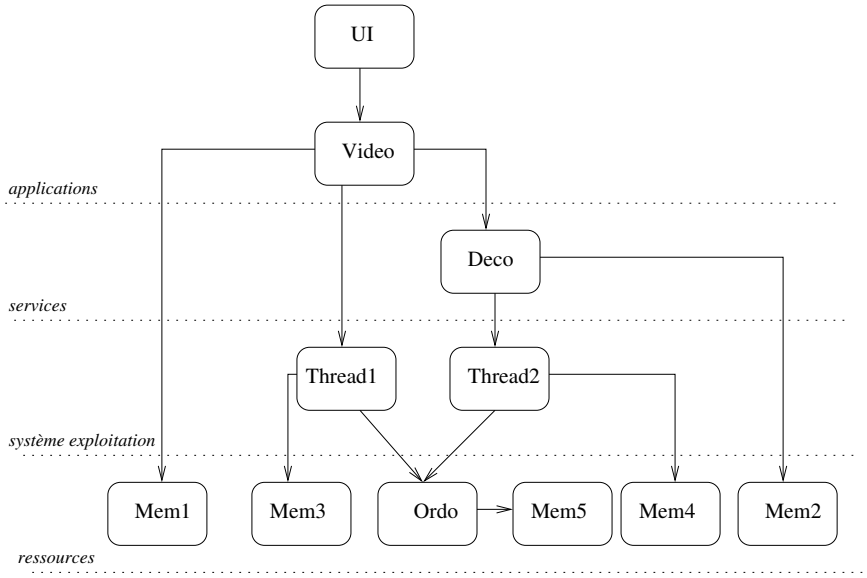

FIG. 3.16 – Représentation du système initial.

VIDEO				
QdS objectif	Contrainte locale	QdS requises		
		DECODEUR	THREAD	MEM
BON	GRAND	25 i/s	20%	30 ko
MAUVAIS	PETIT	10 i/s	20%	30 ko

TAB. 3.2 – Table de mapping du composant VIDEO.

plus, pour le niveau BON, le composant VIDEO affiche l'image en plein écran (GRAND) et en petit (PETIT) pour le niveau de QdS MAUVAIS.

Quand au décodeur, il peut fournir soit 25 images par secondes (si 40% du CPU et 100 ko de mémoire lui sont attribués), soit 10 images par seconde (si 20% du CPU et 150 ko de mémoire lui sont attribués), suivant l'algorithme utilisé. Le décodeur possède deux algorithmes de décodage suivant la QdS fournie : un algorithme RAPIDE nécessitant beaucoup de ressources et un algorithme LENT.

De plus, chaque thread nécessite 5 ko de mémoire et l'ordonnanceur en nécessite 3 ko supplémentaire pour chaque thread ordonnancé.

Pour cet exemple, nous considérons que toute la puissance du processeur peut être allouée et que seul 400 ko de

DECODEUR			
QdS objectif	Contrainte locale	QdS requises	
		THREAD	MEM
25 i/s	RAPIDE	40%	100 ko
10 i/s	LENT	20%	150ko

TAB. 3.3 – Table de mapping du composant DECO.

mémoire sont disponibles. De plus, le nombre d'instances de VIDEO doit être limité à deux. Enfin, deux niveaux d'importance sont définis pour l'exécution des composants : HAUT et BAS, avec HAUT plus important que BAS.

3.3.2.2 Mise en place de Qinna

L'intégration de l'architecture Qinna dans le système présenté suit les étapes définies précédemment. Il s'agit, tout d'abord, d'identifier les QoSComponents. Les composants fournissant une interface avec un besoin de gestion de QdS sont les composants gérant l'accés aux ressources (ORDO, MEM1, MEM2, MEM3, MEM4 et MEM5), les composants de la couche système d'exploitation (THREAD1 et THREAD2), le composant service (DECO) et le composant application (VIDEO). Le composant UI, réifiant l'interface homme machine, n'est pas un QoSComponent car il ne fournit pas d'interface nécessitant une gestion de la QdS.

L'étape suivante est d'expliciter les contraintes locales et les niveaux de QdS objectifs des QoSComponents identifiés. La contrainte locale du composant vidéo identifie le mode d'affichage à réaliser (petit ou grand), tandis que sa QdS objectif est à BON ou MAUVAIS. Le composant décodeur possède, comme contrainte locale, la variable permettant de choisir un des algorithmes de décodage et comme niveau de QdS objectif, le nombre d'images par seconde devant être décodée (25 ou 10 images par seconde). Les contraintes locales et les niveaux de QdS objectifs des composants mémoires sont les tailles mémoires respectives qu'il permettront d'allouer. Les contraintes locales des threads indiquent le pourcentage de CPU pouvant être utilisé par les threads, tandis que leurs niveaux de QdS objectifs indiquent s'ils doivent être ordonnancé ou pas (valeur à ACTIF ou INACTIF). Enfin, la contrainte locale et le niveau de QdS objectif de l'ordonnanceur identifient la charge courante ordonnancée. Ces composants sont alors renommés en QoSVideo, QoSDeco, QoSOrdo, QoSThread1, QoSThread2 et QoSMem*i*, avec $i \in [1;5]$.

A chaque classe de QoSComponents, nous ajoutons un QoSComponentBroker (QoSDecoBroker, QoSMemBroker, QoSThreadBroker, QoSVideoBroker, QoSOrdoBroker) . Les différentes contraintes globales des QoSComponentBrokers sont :

- 400 ko pour le QoSMemBroker, correspondant à la quantité totale de mémoire pouvant être allouée. La contrainte globale courante correspond alors à la quantité totale de mémoire contractualisée. Le test d'admission consiste alors à comparer la somme de la QdS objectif demandée et de la contrainte globale courante, à la contrainte globale.
- 100% pour le QoSOrdoBroker, représentant la charge maximale du processeur pouvant être allouée. La contrainte globale courante correspond alors à la charge processeur contractualisée. Le test d'admission compare la somme du niveau de QdS objectif demandé et de la contrainte globale courante, à la contrainte globale.
- 2 pour le QoSDecoBroker et le QoSVideoBroker, afin de limiter à deux le nombre d'instances de QoSVideo et QoSDeco, quelque soit la QdS fournie. La contrainte globale courante de QoSDecoBroker et de QoSVideoBroker correspond alors au nombre de QoSDeco, respectivement de QoSVideo, réservé. Le test d'admission compare alors uniquement la contrainte globale courante plus un, à la contrainte globale.
- 10 pour le QoSThreadBroker, identifiant le nombre maximal de threads autorisés ayant un niveau de QdS objectif à ACTIF. La contrainte globale courante correspond au nombre de threads déjà réservé. Enfin, le test d'admission est le même que précédemment : il compare le nombre de réservations de threads plus un à la contrainte globale.

L'étape suivante est d'ajouter les QoSComponentManagers aux QoSComponentBrokers respectifs, puis d'initialiser les tables de mapping. Les tables de mapping des QoSVideoManager et QoSDecoManager sont données

par les tableaux 3.2 et 3.3. La table de mapping de QoSThreadManager indique que pour qu'un thread fournisse un niveau de QdS égal à x% du CPU , il requiert un composant mémoire lui fournissant 5 ko de mémoire, un ordonnanceur pouvant l'ordonnancer avec une QdS égale à x% du CPU et un thread ACTIF. La table de mapping de QoSOrdoManager indique que pour chaque nouvelle demande d'ordonnancement, le composant QoSOrdo requiert 3 ko de mémoire supplémentaire. La table de mapping QoSMemManager est simple car elle ne possède pas de QdS requise et la valeur de la contrainte locale est égale à la QdS fournie.

Nous considérons que les tables de mapping sont fiables et, étant donné qu'aucune donnée n'est fixée par défaut, il n'est pas nécessaire de mettre en place de QoSComponentObservers.

Enfin, la dernière étape consite à ajouter un QoSDomain et de configurer la relation d'ordre sur les niveaux d'importance tels que $Imp(HAUT) > Imp(BAS)$.

La figure 3.17 donne une vue globale du système après intégration de l'architecture Qinna.

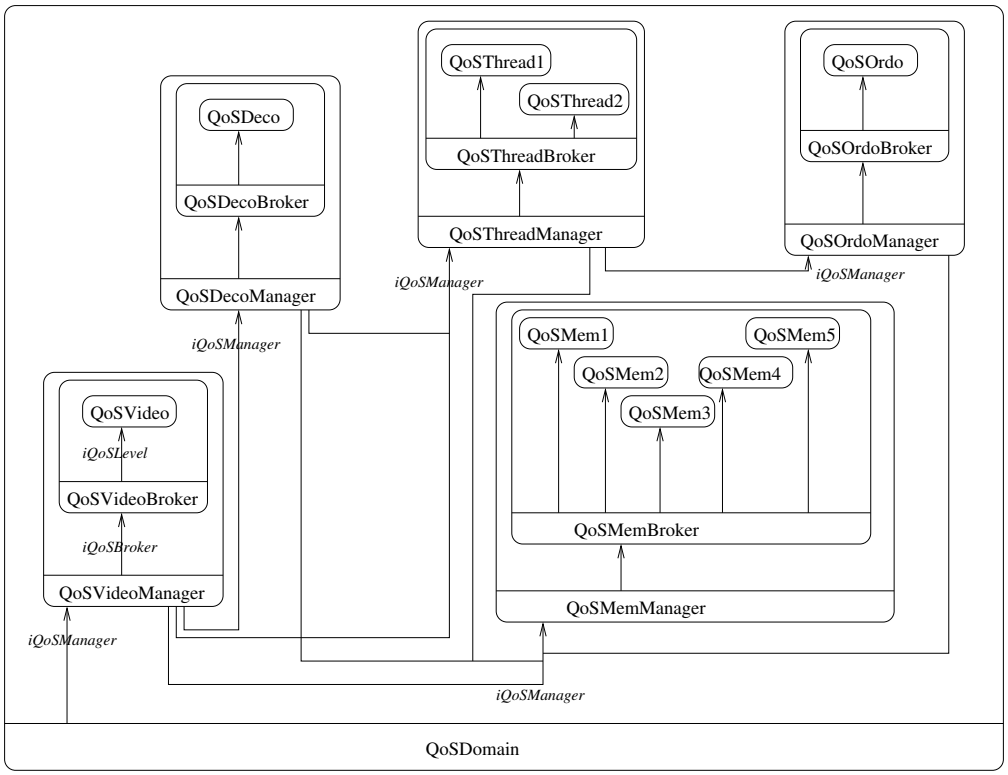

FIG. 3.17 – Représentation du système présenté à la figure 3.16 après intégration de Qinna.

La figure 3.18, page 58, représente les interactions entre les différents composants lorsque l'utilisateur requiert un composant QoSVideo de niveau BON et d'importance HAUT. De plus, le tableau 3.4 page 57 résume l'état des différentes contraintes globales après chaque appel de réservation sur un QoSComponentBroker.

L'utilisateur effectue la demande au QoSDomain, qui transmet la demande au QoSVideoManager.

Le QoSVideoManager examine sa table de mapping et réserve, tout d'abord, le composant QoSDeco à 25

images/seconde à QoSDecoManager.

Le QoSDecoManager examine sa table de mapping et requiert, tout d'abord, un thread à 40% du CPU au QoSThreadManager. Celui-ci requiert alors le composant QoSOrdo à QoSOrdoManager pouvant ordonnancer un thread à 40% du CPU.

Le QoSOrdoManager examine sa table de mapping et requiert, tout d'abord, un composant mémoire de 3 ko à QoSMemManager, ce qui est accepté. La nouvelle quantité de mémoire disponible est alors de 397 ko.

Le QoSOrdoManager demande alors à QoSOrdoBroker si QoSOrdo peut ordonnancer un thread à 40% du CPU. Aucun thread n'est en cours d'ordonnancement et donc la référence de QoSOrdo est retournée à QoSOrdo-Manager qui la transmet à QoSThreadManager. La quantité de CPU libre est de 60%.

Le QoSThreadManager requiert alors 5 ko de mémoire à QoSMemManager qui lui retourne la référence de QoSMem4 configuré à 5 ko. La quantité de mémoire disponible est alors égale à 392 ko. Puis, le QoSThreadManager requiert à QoSThreadBroker un thread de libre et le lie (QoSThread2) à QoSOrdo et QoSMem4. Il reste donc 9 threads de libre. Enfin, la référence de QoSThread2 est retournée à QoSDecoManager.

Le QoSDecoManager requiert alors 100 ko de mémoire à QoSMemManager (QoSMem2) et un QoSDeco configuré à RAPIDE à QoSDecoBroker. La quantité de mémoire disponible est alors de 292 ko et il n'y a plus qu'un QoSDeco de libre. Le QoSDeco est ensuite lié à QoSThread2 et QoSMem2 et sa référence est passée à QoSVideoManager.

Le QoSVideoManager requiert à son tour un thread à 20% du CPU à QoSThreadManager et 30 ko de mémoire à QoSMemManager. Après ces deux réservations, la charge courante de QoSOrdo est de 60% et il reste 8 threads de libres. Il est à noter que lors de la demande d'activation d'un nouveau thread, le QoSOrdoManager ne demande pas un nouveau composant mémoire de 3 ko, mais demande une modification du composant mémoire précédemment obtenu. De plus, la taille mémoire disponible est maintenant de 254 ko. Une fois les références de QoSThread1 et QoSMem1 reçues, il effectue une demande de QoSVideo libre configuré à GRAND à QoSVideoBroker. Enfin, QoSVideo est lié à QoSDeco, QoSThread1 et QoSMem1, puis sa référence est retournée au QoSDomain qui la transmet à l'utilisateur.

Appel de service	QoSComponentBroker appelé	Nombre de QoSVideo disponibles	Nombre de QoSDeco disponibles	Nombre de QoSThread disponibles	Quantité mémoire disponibles (ko)	Quantité CPU disponibles (%)
VALEURS INITIALES DES CONTRAINTES GLOBALES COURANTES		2	2	10	400	100
reserve(3,3)	QoSMemBroker				**397**	
reserve(40,40)	QoSOrdoBroker					**60**
reserve(5,5)	QoSMemBroker				**392**	
reserve(40,ACTIF)	QoSThreadBroker			**9**		
reserve(100,100)	QoSMemBroker				**292**	
reserve(RAPIDE,25i/s)	QoSDecoBroker		**1**			
reserve(3,3)	QoSMemBroker				**289**	
reserve(20,20)	QoSOrdoBroker					**40**
reserve(5,5)	QoSMemBroker				**284**	
reserve(20,ACTIF)	QoSThreadBroker			**8**		
reserve(30,30)	QoSMemBroker				**254**	
reserve(GRAND,BON)	QoSVideoBroker	**1**				
VALEURS FINALES DES CONTRAINTES GLOBALES COURANTES		1	1	8	254	40

TAB. 3.4 – Évolution de l'état du système au travers des contraintes globales courantes lors de la réservation d'un QoSVideo de niveau BON.

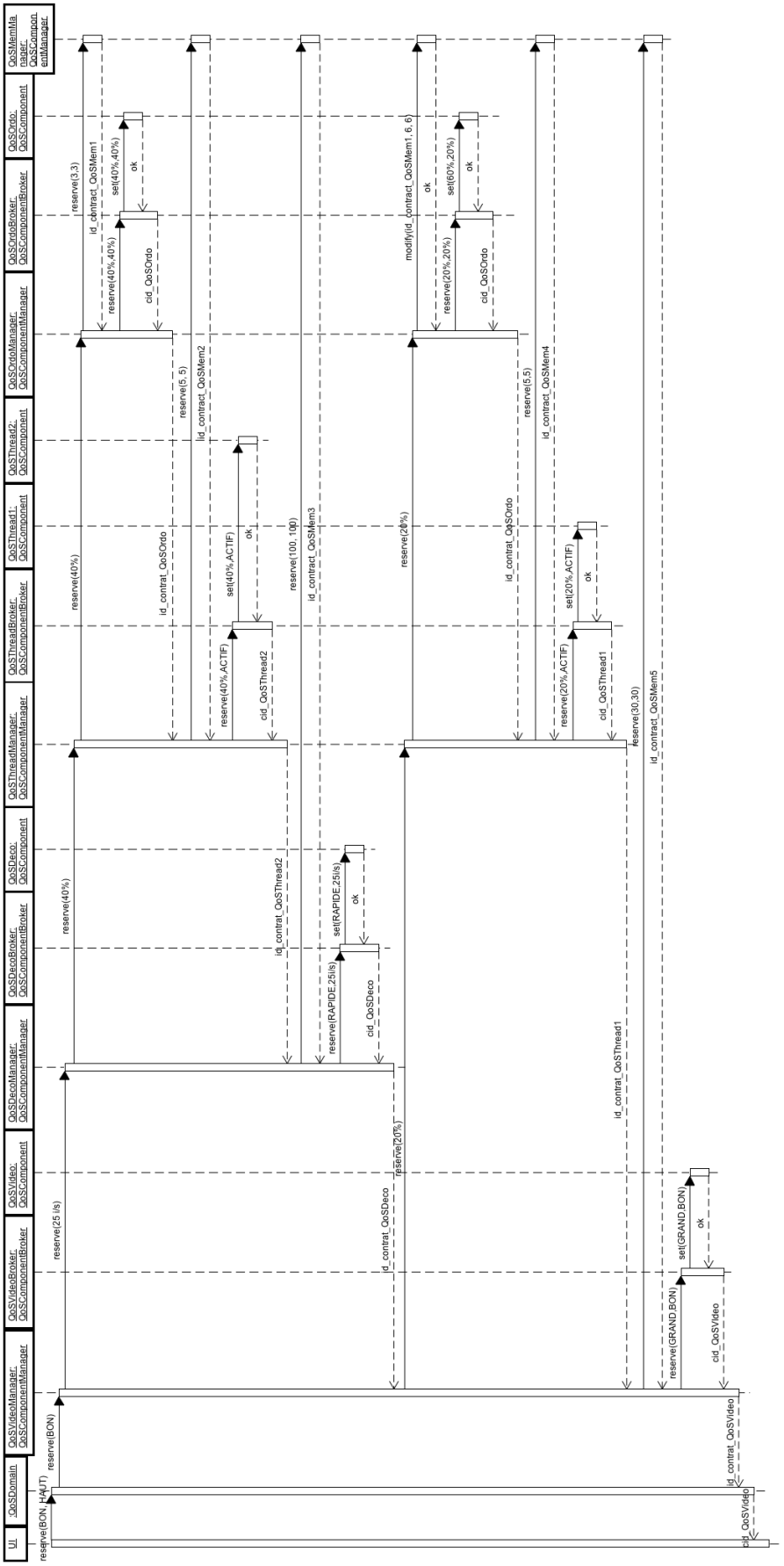

FIG. 3.18 – Réservation réussie du composant QoSVideo de niveau BON. Afin de ne pas surcharger le diagramme, les requêtes liées au QoSMemBroker ont été omises.

3.4 Conclusion

Au regard des objectifs fixés par l'architecture à la partie 3.1.1 page 33, Qinna apporte les réponses suivantes :

- **Généricité** : par rapport à l'objectif de généricité, la définition de Qinna, aussi bien au niveau de la définition des interfaces que des types de composants, ne fait aucune hypothèse sur la façon dont va être gérée la QdS. Cela permet l'intégration de gestion de QdS diverses tel que celles utilisées dans les systèmes temps réel [90] ou les systèmes multimédia [91]. En effet, Qinna définit uniquement des types abstraits de composants (boîtes vides) à partir desquels la gestion de la QdS est intégrée.
- **Dynamicité** : la gestion dynamique de la QdS est permise grâce à la prise en compte par Qinna des concepts de gestion de contrats de QdS identifiés à la figure 2.7 page 26. Les aspects liés à la spécification sont pris en charge par l'interface iQoSDomain. La phase d'initialisation est réalisée par les QoSComponentManagers (pour le mapping) et par les QoSComponentBrokers (pour le test d'admission et la réservation). Enfin, les aspects de gestion sont pris en charge par les QoSComponentObserver (pour l'observation), par les QoSComponentManagers (pour les mécanismes de d'adaptation et de maintenabilité) et le QoSDomain (pour les politiques d'adaptation et de maintenabilité). De plus, Qinna définit un ensemble de scénarios de gestion dynamique de contrats (mise en place/annulation, adaptation et mise à jour de contrats).
- **Hétérogénéité** : Qinna permet l'utilisation de langages de spécification de QoS différents entre deux composants grâce à la définition de l'opérateur *T_QoS translate(T_QoS q1)* au niveau des QoSComponentManagers. Cet opérateur permet de traduire une spécification initiale en une spécification compréhensible par la table de mapping. Par exemple, si une spécification sous forme de fonction f est transmise à un QoSComponentManager et que celui-ci ne travaille qu'avec des valeurs fixes, l'opérateur *translate* peut retourner $Max(f)$, $Min(f)$, ou *default* s'il ne peut analyser f. L'implication de cet opérateur sur l'ensemble de l'architecture est détaillée au chapitre suivant.
- **Confiance** : la réponse à cette problématique s'appuie sur la philosophie implicite de Qinna. Celle-ci interdit à un QoSComponent de consommer une QdS (identifiée par la QdS courante) supérieure à celle demandée et allouée (identifiée par la QdS objectif), via des contrôles effectués à chaque demande de service. Par exemple, dans le cas des composants mémoire (QoSMem) une vérification de la quantité de mémoire réservée est effectuée à chaque demande d'allocation. Pour autant un QoSComponent peut requérir l'ensemble des ressources du système. Dans ce cas, soit il s'agit du QoSComponent ayant le niveau d'importance le plus élevé auquel cas Qinna remplit son rôle en lui fournissant l'ensemble des ressources disponibles. Soit il s'agit d'un QoSComponent malveillant, et il serait alors nécessaire d'intégrer des aspects de sécurité tels que l'authentification ou la cryptographie, ce qui dépasse le cadre de ce travail.
- **Auto-configurabilité** : afin de pouvoir déterminer dynamiquement les valeurs des QdS requises identifiées dans les tables de mapping, Qinna met en oeuvre trois mécanismes. Tout d'abord, les niveaux de QdS requis sont fixés à la valeur *default* lorsqu'ils inconnus, et la table de mapping est caractérisée comme non fiable (*unreliable*). Puis l'architecture met en place des QoSComponentObserver afin d'observer et d'évaluer dynamiquement le niveau de QdS réellement fourni. Enfin, les informations transmises au QoSDomain par le QoSComponentObserver permettent de mettre en oeuvre les politiques de maintenabilité afin de déterminer les niveaux de QdS réellement requis.
- **Réutilisabilité** : afin d'augmenter ses possibilités de réutilisation, l'architecture Qinna respecte les principes de séparation des préoccupations et de séparation entre les politiques et les mécanismes de gestion de QdS. Ces séparations sont identifiées par des composants différents. En effet, la séparation entre les préoccupations fonctionnelles et de QdS sont implémentées dans des composants disjoints : les QoSComponents implémentent les préoccupations fonctionnelles et les configurations de QdS, alors que les préoccupations de QdS sont implémentées par les composants QoSComponentBrokers, QoSComponentManagers, QoS-

ComponentObservers et QoSDomain. De plus, au sein de la préoccupation de QdS, les politiques et les mécanismes de gestion de QdS sont séparés : les politiques sont implémentées par le QoSDomain, alors que les mécanismes sont à la charge des QoSComponentManagers.
De plus, chaque composant de l'architecture possède un ensemble de propriétés configurables afin de pouvoir être adapté à chaque nouvel environnement d'exécution. Par exemple, il est possible de réutiliser le QoSDomain dans un autre système en modifiant uniquement sa relation d'ordre sur les niveaux d'importance. De même, chaque QoSComponent peut-être réutilisé en conservant les mêmes QoSComponentBroker et QoSComponentManager. Tous ces aspects ont alors permis de mettre en place une première bibliothèque de composants Qinna à partir desquels un nouveau système peut être construit.

Ce chapitre a permis de définir l'architecture de gestion de QdS proposée dans cette thèse. La définition de l'architecture, appelée Qinna, est réalisée à l'aide de type de composants, ainsi que d'un ensemble d'interfaces et de types de données. Parallèlement, Qinna possède un comportement dynamique de gestion des contrats de QdS.

Nous avons ensuite présenté un guide méthodologique d'intégration de l'architecture Qinna aux systèmes à composants, que nous avons illustré au travers d'un exemple.

Enfin, nous avons présenté les réponses apportées par Qinna aux objectifs fixés en introduction (cf. partie 3.1.1, page 33). Cependant, la façon dont Qinna répond à ces objectifs induit des coûts aussi bien qualitatifs (par exemple, par un modèle de programmation particulier) que quantitatifs (CPU, mémoire). Le chapitre suivant permet d'évaluer ces coûts au travers de diverses illustrations et expérimentations menées sur un ensemble de cas types présents dans les systèmes visés par cette étude.

Chapitre 4

Illustrations, expérimentations et évaluations de l'architecture Qinna

Il semble difficile, voire impossible, de valider formellement l'intérêt des principes mis en place par une architecture, ainsi que l'architecture elle-même. En revanche, afin d'en percevoir les apports et les limites, il est intéressant de l'illustrer, de l'expérimenter et de l'évaluer. C'est l'objectif de ce chapitre pour l'architecture Qinna présentée au chapitre précédent. Cet objectif passe par l'étude d'un ensemble de cas types variés et représentatifs des systèmes ouverts à composants. Nous avons donc privilégié le domaine d'application visé par l'étude. Ces cas types ont été décomposés en deux catégories. La première catégorie porte sur l'utilisation de ressources, tandis que la deuxième catégorie se focalise sur les comportements dynamiques. Enfin, la dernière partie de ce chapitre effectue une évaluation quantitative de l'architecture Qinna au travers de deux réalisations.

4.1 Utilisation de ressources

Le premier ensemble de cas types présente les différentes configurations possibles d'utilisation de ressources par les composants. Nous analysons, dans un premier temps, le cas où un composant utilise différentes ressources, puis le cas où une ressource est utilisée par plusieurs composants. Un dernier cas permet d'étudier les mécanismes mis oeuvre par l'architecture pour prendre en compte des spécifications hétérogènes de demandes de QdS. Nous présentons la façon dont Qinna prend en compte chacun de ces cas, puis nous l'illustrons au travers d'expérimentations avant d'en effectuer une analyse.

4.1.1 Un composant utilise *n* ressources différentes

4.1.1.1 Présentation

La première évaluation porte sur un composant A fournissant une interface *a* et requérant les services de *n* ressources différentes. Les ressources sont réifiées sous forme de composants, notés $R_1 ... R_n$, et leurs services sont

fournis au travers d'interfaces notées respectivement $r_1 ... r_n$ (cf. figure 4.1).

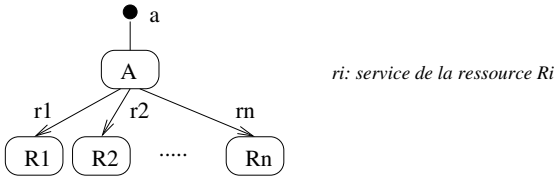

FIG. 4.1 – Représentation graphique d'un composant A utilisant les services de n ressources R_i.

L'implémentation du composant A est configurée via un paramètre noté cl_A. De plus, on considère, pour ce cas, qu'il ne peut exister qu'une seule instance de A. La QdS fournie sur l'interface *a* est notée q_a, tandis que celle requise sur les interfaces r_1 à r_n est notée qr_i avec i identifiant l'interface du composant ressource R_i. Le niveau maximal de QdS pouvant être fournie par un composant ressource sur l'interface r_i est noté Qr_i.

Afin de pouvoir intégrer Qinna, il doit exister une fonction F_{map} telle que :

$$(qr_1, ..., qr_n, cl_A) = F_{map}(q_a)$$

La fonction F_{map} implémente l'opération de mapping permettant d'effectuer le lien entre les niveaux de QdS fournis et requis par A ainsi que de sa contrainte locale.

4.1.1.2 Intégration de Qinna

Afin d'intégrer Qinna, nous appliquons la méthodologie présentée à la figure 3.15 page 50. Chaque composant est un QoSComponent car il fournit une interface dont on souhaite gérer la QdS. Un QoSComponentBroker et un QoSComponentManager sont associés à chaque composant ressources R_i et sont notés respectivement QoSRiBroker et QoSRiManager. De même, un QoSComponentBroker et un QoSComponentManager sont associés au composant A, noté QoSABroker et QoSAManager. Enfin, un QoSDomain est intégré afin de délimiter la portée de la gestion de QdS (cf. figure 4.2).

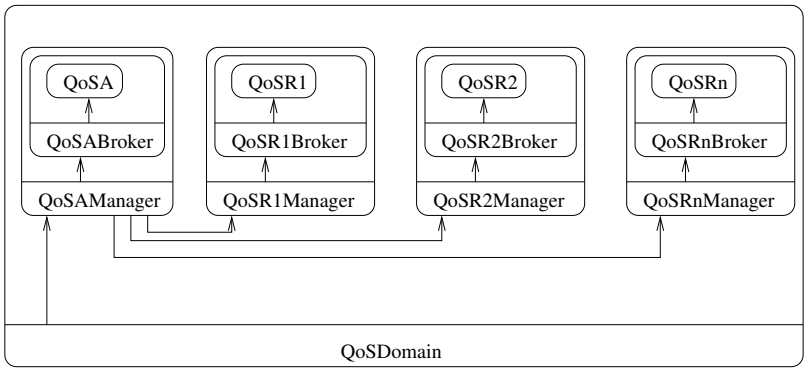

FIG. 4.2 – Intégration de Qinna au cas type d'utilisation de n ressources par un composant tel que présenté à la figure 4.1.

La contrainte globale de chaque QoSRiBroker est le niveau de QdS maximal pouvant être fourni par le composant ressource $QoSR_i$ et est égale à Qr_i. La contrainte globale courante, notée $QCourantr_i$, de chaque QoS-

RiBroker est le niveau de QdS contractualisé par le composant ressource $QoSR_i$. La contrainte locale et le niveau de QdS objectif de chaque composant $QoSR_i$ est le niveau de QdS fourni sur l'interface r_i et est égale à qr_i.

La contrainte locale de $QoSA$ est le paramètre de configuration cl_A, tandis que son niveau de QdS objectif est le niveau de QdS devant être fourni noté q_a. On souhaite limiter le nombre d'instances de $QoSA$ à 1 quelque soit la valeur de q_A : la contrainte globale de QoSABroker est donc égale à 1. La table de mapping du composant QoSAManager est donnée par la fonction $F_{map}(qa)$.

Le QoSAManager gère les contrats entre le composant QoSA et les composants QoSRi. Pour cela, il se base sur les composants QoSRiManagers, qui eux-mêmes se basent sur leur QoSRiBroker associé. Ces derniers effectuent les tests d'admission afin de vérifier que la somme de qr_i et de $QCourantr_i$ soit inférieure à Qr_i grâce à la fonction $compare(Qr_i, add(QCourantr_i, qr_i))$. Si le test d'admission est validé alors la contrainte globale courante est mise à jour : $QCourantr_i = add(QCourantr_i, qr_i)$. Il est à noter que le QoSDomain a uniquement connaissance du contrat portant sur l'interface a de QoSA.

4.1.1.3 Illustration

Prenons le cas où le composant A est un composant permettant la visualisation d'une vidéo distante au travers de l'interface a. Le composant A, pour fournir sur l'interface a un niveau de QdS égal à $q_a = BON$, requiert 30 ko de mémoire, 40% du temps CPU et 15% de la bande passante du réseau. De plus, l'algorithme de lecture vidéo doit être paramétré à $cl_A = RAPIDE$. Le composant A peut fournir un autre niveau de QdS sur l'interface a, noté $q_a = MAUVAIS$. Pour ce niveau, A requiert 20 ko de mémoire, 20% du CPU et 10% de la bande passante du réseau et doit être configuré à $cl_A = LENT$. Le niveau maximal de QdS des trois ressources est fixé à 100 ko pour la mémoire, 100 % du temps CPU, et 50 % de la bande passante du réseau. Enfin, on souhaite limiter le nombre d'instance de A à 1 quelque soit sa configuration (RAPIDE ou LENT).

L'intégration de Qinna implique l'ajout des composants QoSComponentBroker et QoSComponentManager pour la mémoire, le CPU, le réseau et le composant A. Les contraintes globales de QoSMemBroker, QoSCPUBroker et QoSReseauBroker sont respectivement 100 ko, 100 % et 50 %, et celle de QoSABroker est 1. A l'initialisation, les contraintes globales courantes de QoSMemBroker, QoSCPUBroker, QoSReseauBroker et QoSABroker sont respectivement 0 ko, 5% (dû à l'exécution du système et de l'architecture), 0 % et celle de QoSABroker est 0. Enfin, la table de mapping du composant QoSAManager est donné par les relations :

$$(qa = BON) \underset{F_{map}^{-1}}{\overset{F_{map}}{\longleftrightarrow}} (qr_{mem} = 30ko, qr_{cpu} = 40\%, qr_{reseau} = 15\%, cl_A = RAPIDE)$$

et

$$(qa = MAUVAIS) \underset{F_{map}^{-1}}{\overset{F_{map}}{\longleftrightarrow}} (qr_{mem} = 20ko, qr_{cpu} = 20\%, qr_{reseau} = 10\%, cl_A = LENT)$$

QoSABroker est en charge d'admettre la réservation d'un seul QoSA quelque soit son niveau de QdS objectif (BON ou MAUVAIS). Le QoSCPUBroker vérifie que la somme de qr_{cpu} et de $QCourant_{cpu}$ soit inférieure à 100%. Le QoSMemBroker vérifie que la somme de qr_{mem} et de $QCourant_{mem}$ soit inférieure à $100ko$ et le QoSReseauBroker que la somme de qr_{reseau} et de $QCourant_{reseau}$ soit inférieure à 50%.

4.1.1.4 Analyse

Qinna impacte le système par l'ajout de composants supplémentaires. En effet, l'architecture impose l'ajout de 2 composants supplémentaires pour chaque composant nécessitant une gestion de QdS, ainsi que d'un composant délimitant la portée de gestion de QdS (QoSDomain). Le nombre de composants introduit par Qinna pour ce cas type est donné par :

$$\begin{aligned} Nb\_Comp\_Qinna &= (Nb\_QoSComp \times 2) + 1 \\ &= (Nb\_Comp\_appli + Nb\_Comp\_Ressources) \times 2 + 1 \\ &= (1 + n) \times 2 + 1 \end{aligned}$$

Il est à noter le rôle central de la fonction de mapping. Celle-ci permet, à partir d'un niveau de QdS fourni donné, de retourner une seule valeur de QdS requise (cf. illustration précédente) ou plusieurs. En effet, dans le cas du composant vidéo, un niveau de QdS BON peut nécessiter soit 40% du CPU, 30 ko de mémoire et 15% du réseau, soit dans un autre mode de fonctionnement 20% du CPU, 60 ko de mémoire et 35% du réseau. Dans ce cas de figure, le QoSVideoManager teste la première configuration, puis la deuxième, si la première a échoué. Enfin, les valeurs de la fonction de mapping peuvent être paramétrées : par exemple, la mémoire requise par le composant vidéo peut dépendre de la taille de l'image à afficher.

Cette fonction mapping peut être facilement gérée si les niveaux de QdS des composants peuvent être discrétisés. Mais il est alors nécessaire de prévoir une correspondance entre les besoins et l'offre de QdS. Par exemple, dans le cas où un composant A requiert soit 10, 20 ou 30 unités de QdS au composant B qui n'offre, lui, que 25 ou 35 unités de QdS, il est nécessaire de prévoir une correspondance entre les niveaux de QdS. La prise en compte de la non compatibilité de l'offre et des besoins est traitée dans la section relative à l'hétérogénéité de la QdS (cf. section 4.1.3 page 74). De plus, plus la granularité des composants d'une application est fine et plus il est difficile de déterminer leur niveau de QdS. Par exemple, dans le cas où chaque fonction réalisée par le composant vidéo est réifiée en composant, il est alors difficile d'exprimer une QdS sur chacun de ces composants. Si ces niveaux ne peuvent être déterminés à l'avance, il sont fixés par défaut. Ces aspects font l'objet des études des comportements dynamiques présentés dans la partie suivante.

4.1.2 Deux composants partagent les services d'un composant

4.1.2.1 Présentation

La deuxième évaluation porte sur le cas classique de partage de composant. Soit deux composants, A et B, fournissant respectivement les interfaces *a* et *b* et requérant l'interface *c* d'un composant C (cf. figure 4.3).

Le niveau de QdS sur l'interface *a*, noté q_a, dépend du niveau de QdS fourni par le composant C pour le composant A, noté q_{ac}, à travers l'interface *c*. Cette dépendance est définit par la fonction F_{Amap}. De même, le niveau de QdS sur b, noté q_b, dépend du niveau de QdS fourni par C pour B, noté q_{bc}, à travers l'interface c.

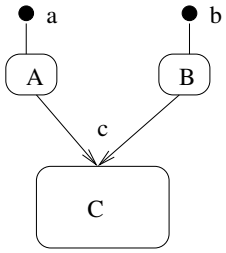

FIG. 4.3 – Cas type de partage de composant.

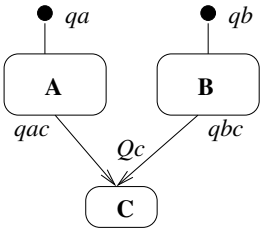

FIG. 4.4 – Représentation des notations de QdS pour le cas type de partage de composant.

Cette dépendance est définie par la fonction F_{Bmap}. De plus, le niveau maximal de QdS pouvant être fourni par le composant C est noté Q_C (cf. figure 4.4).

L'intégration de Qinna, pour ce cas type, dépend de la nature du composant C. On considère alors deux cas : soit le composant C est sécable, soit il est insécable. Nous définissons un composant sécable comme un composant pouvant être utilisé simultanément par plusieurs composants (sans effet de bord), alors qu'un composant insécable ne peut être utilisé simultanément par deux composants. Afin de pouvoir gérer de manière isolée (pour chaque composant requérant C) la QdS fournie par un composant sécable, il est possible de le diviser en sous-composants de même nature, tandis que dans le cas d'un composant insécable, les accès à ce composant doivent être ordonnancés dans le temps. L'intégration de Qinna, dans ces deux cas, est analysée dans la suite de cette partie.

4.1.2.2 Partage d'un composant sécable

Dans le cas où le composant C est sécable, il est nécessaire de le diviser en deux composants Ca et Cb, dédiés respectivement aux composants A et B. Cela permet alors de gérer séparément la QdS fournie par les composants Ca et Cb aux composants A et B. Les composants Ca et Cb fournissent l'interface c issue de C (cf. figure 4.5).

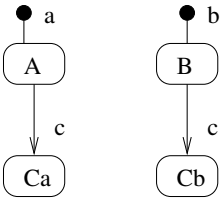

FIG. 4.5 – Représentation du partage d'un composant sécable C entre les composants A et B.

4.1.2.2.1 Intégration de Qinna Les composants A et B sont identifiés en tant que QoSComponents : un QoSComponentBroker et un QoSComponentManager leurs sont donc associés. Les composants Ca et Cb sont aussi des QoSComponents, mais étant donné qu'il s'agit de composant de même classe, un seul QoSComponentBroker et QoSComponentManager leurs sont associés. De plus, un QoSDomain est ajouté afin de délimiter la portée de la gestion de la QdS (cf. figure 4.6).

La contrainte globale du composant QoSCBroker représente le niveau de QdS maximal total pouvant être fourni par les composants QoSCa et QoSCb : la contrainte globale est alors égale à Q_C. Les niveaux de QdS objectifs

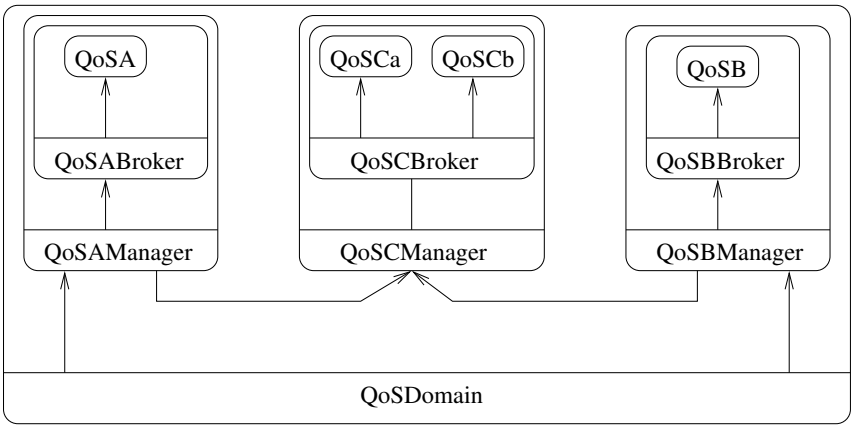

FIG. 4.6 – Intégration de Qinna dans le cas d'un partage de composant sécable.

des composants QoSCa et QoSCb représentent les niveaux de QdS fournis par chacun des deux composants : ces niveaux sont alors égaux respectivement à q_{ac} et q_{bc}. La somme de ces deux niveaux correspond à la contrainte globale courante. Le rôle de QoSCBroker est de s'assurer que la somme des niveaux de QdS devant être fournis par QoSCa et QoSCb soit inférieure au niveau de QdS maximal : i-e $q_{ac} + q_{bc} < Q_C$. Pour cela, le QoSCBroker utilise les opérateurs $compare(Q_C, add(q_{ac}, q_{bc}))$.

4.1.2.2.2 Illustrations Le partage de composants sécables peut être illustré au travers de deux exemples caractéristiques. Dans la première illustration le composant partagé réifie une ressource sécable (la mémoire), tandis que dans la deuxième illustration le composant partagé est un composant applicatif pouvant être dupliqué.

Composant ressource Prenons le cas où deux composants A et B requièrent de la mémoire dynamique pour fournir leurs interfaces a et b. La mémoire, étant une ressource, est réifiée sous forme de composant. Nous fixons la taille de la mémoire pouvant être allouée dynamiquement à 100 ko et nous déterminons que pour que les composants A et B fournissent un niveau de QdS BON, respectivement MAUVAIS, sur leurs interfaces, ils requièrent 45 ko, respectivement 25 ko, de mémoire chacun.

Afin d'intégrer Qinna, il est nécessaire de décomposer le composant mémoire en deux composants MemA et MemB. Puis trois QoSComponentBrokers et trois QoSComponentManagers sont intégrés respectivement aux composants QoSA, QoSB et QoSMemA et QoSMemB. La contrainte globale de QoSMemBroker est égale à $Q_C = 100ko$, tandis que les tables de mapping de QoSAManager et QoSBManager indiquent que pour un niveau de QdS égal à BON, respectivement MAUVAIS, 45 ko, respectivement 25 ko, de mémoire sont requis. Une fois que le composant QoSMemA est réservé pour QoSA, son niveau de QdS objectif est fixée à $q_{ac} = 45ko$. QoSMemA correspond alors à une zone de mémoire de taille 45 ko. Le niveau de QdS courant représente alors la quantité de mémoire réellement utilisée. Cela implique que le composant QoSA ne peut utiliser plus de 45 ko de mémoire sans renégociation de son contrat, car à chaque appel du service d'allocation de QoSMemA, celui-ci, vérifie que le niveau de QdS courant soit inférieur au niveau de QdS objectif grâce à l'opérateur $compare(q_{objectif}, q_{courant})$.

Enfin, il est à noter que lors d'une dégradation dynamique du contrat de QoSA de BON à MAUVAIS, la taille de QoSAMemA passe de 45 ko à 25 ko. Il est alors à la charge de QoSA de prévoir les mécanismes nécessaires

permettant d'identifier les données devant être libérées. Par exemple, si QoSA est une application de vidéo, la libération de la mémoire entraînera la perte d'images en cours de décodage.

Composant applicatif Considérons maintenant le cas, où deux composants A et B requièrent les services fournis par un composant de décodage vidéo, appelé DECO. Le composant DECO peut fournir au maximum un niveau de QdS égal à 28 images par seconde. Le nombre d'instances dans le système du composant DECO dépend du niveau de QdS fourni : pour 15 i/sec une instance est autorisée, pour 10 i/sec deux instances sont autorisées et pour 7 i/sec quatre instances sont autorisées. Nous déterminons que pour que A et B fournissent un niveau de QdS égal à BON, respectivement MAUVAIS, ils requièrent une QdS égale à 10 i/sec, respectivement 7 i/sec, sur l'interface de DECO.

L'intégration de Qinna s'effectue comme pour le cas précédent : le composant DECO est découplé en un composant dédié à A et un composant dédié à B. Puis, trois QoSComponentBrokers et trois QoSComponentManagers sont intégrés à QoSA, QoSB et QoSDECOA et QOSDECOB. Le niveau de QdS objectif des composants QoSDECOA et QoSDECOB est le niveau de QdS fourni par chaque composant. La différence porte sur la contrainte globale de QoSDECOBroker et sur le test d'admission qui dépend du nombre d'instances en cours d'exécution de QoSDECO et de leurs niveaux de QdS. En effet, la contrainte globale est égale à *1 QoSDeco à 15 i/sec ou 2 QoSDeco à 10 i/sec ou 4 QoSDeco à 7 i/sec*. Si seul QoSDecoA demande à être réservé, le QoSDecoBroker vérifie que la QdS objectif soit inférieur à 15 i/sec. Lorsque que QoSDecoA et QoSDecoB demandent à être réservés, QoSDecoBroker vérifie que chacune des QdS objectifs soit inférieure à 10 i/sec.

Analyse Dans le cas de l'utilisation simultanée d'un composant sécable, il est nécessaire de diviser ce composant en autant de composants l'utilisant. Cela permet de contrôler et de gérer séparément la QdS fournie aux composants. Cette approche permet alors de prendre en compte la problématique liée à la confiance de l'architecture sur les ressources utilisées en définissant des composants contrôlant leur niveau de QdS fourni (par exemple, les composants QoSMémoires définissent des zones mémoires contrôlées par composant avec un mécanisme de contrôle associé). Ce contrôle est particulièrement pertinent pour les ressources matérielles. En effet, ces dernières sont intrinsèques au système et leur politique de contrôle peut être mise en oeuvre de façon fiable par le concepteur. Enfin, l'approche préconisée permet d'éviter les effets de bords et d'identifier (c'est à dire de localiser) les problèmes.

En contre partie, cette approche entraîne une multiplication des composants, ce qui a pour impact d'augmenter le coût mémoire de l'architecture. En effet, dans le cadre des composants réifiant des ressources, bien que chaque sous composant soit de petite taille (quelques dizaines d'octets) il faut la multiplier par le nombre de composants les utilisant. Donc, plus la granularité est fine et plus le coût de l'architecture est important. Par exemple, une application décomposée en N composants, chacun utilisant de la mémoire dynamique, nécessite la mise en place de N composants mémoire. Une solution permettant de limiter le coût de la multiplication des composants est de factoriser les demandes par application. Par exemple, toutes les demandes en mémoire d'une application formée d'un ensemble de N composants sont gérées par un seul composant mémoire.

Enfin, il est à noter que l'architecture laisse à la charge des composants les conséquences d'une dégradation de contrats. En effet, on considère que si le composant a défini plusieurs niveaux de QdS dans la fonction de mapping, il est alors capable de gérer les conséquences du passage d'un niveau à l'autre. Par exemple, dans l'illustration précédente, lorsque la taille de la mémoire disponible pour un composant diminue, celui-ci est en charge de gérer

les conséquences de la diminution. Les composants pouvant définir différents niveaux de QdS mémoire sont, soit des composants traitant des flots de données (vidéos, sons, jeux, internet), où la perte de données dégrade le composant sans provoquer de dysfonctionnement, soit des composants pouvant s'appuyer sur une sauvegarde (propriété de persistance) des informations initialement stockées dans la mémoire dynamique afin de la libérer.

4.1.2.3 Partage d'un composant insécable

Dans le cas où le composant C est insécable, les appels sur l'interface c doivent être ordonnancés dans le temps. L'ordonnancement des appels est réalisé par un ordonnanceur, lui même réifié en composant. L'ordonnanceur du composant C fournit l'interface c aux composants A et B et requiert l'interface c à C (cf. figure 4.7).

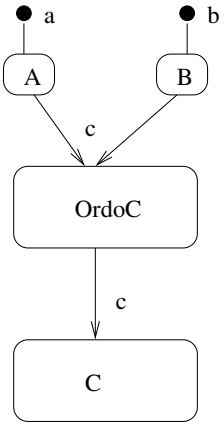

FIG. 4.7 – Partage d'un composant C insécable.

L'unique rôle de l'ordonnanceur est d'ordonnancer les appels de A et B avec une QdS respective égale à q_{ac} et q_{bc} : cela signifie que l'ordonnanceur implémente uniquement la politique d'ordonnancement (soit la gestion de la séquence d'utilisation) et non l'opération d'admission. Cette dernière est celle qui définit si l'ordonnancement est possible pour un niveau de QdS q donné. Cette opération d'admission nécessite la connaissance des niveaux de QdS en cours d'ordonnancement et du niveau de QdS maximal de C, noté Q_C.

4.1.2.3.1 Intégration de Qinna Les composants A, B et OrdoC sont des QoSComponents car on souhaite gérer la QdS de leurs interfaces. Le composant C lui n'est pas un QoSComponent. En effet, les aspects de QdS liée à c sont gérés au composant OrdoC. Par contre, c n'est accessible que via son ordonnanceur QoSOrdoC. Aux trois QoSComponents sont associés un QoSComponentBroker et un QoSComponentManager. De plus, un QoSDomain délimite la portée de la gestion de QdS (cf. figure 4.8).

Nous nous intéressons ici uniquement au rôle des composants liés à QoSOrdoC, étant donné qu'il s'agit de l'aspect original de ce cas type. Le composant QoSOrdoC possède comme niveau de QdS objectif, la somme des niveaux de QdS en cours d'ordonnancement, notée $q_{objectif}$. En effet, il s'agit du niveau de QdS que doit fournir QoSOrdoC aux composants l'utilisant. La contrainte globale du composant QoSOrdoCBroker représente le niveau maximal de QdS pouvant être ordonnancé, notée Q_C, et la contrainte globale courante, notée $QCourante_C$, est,

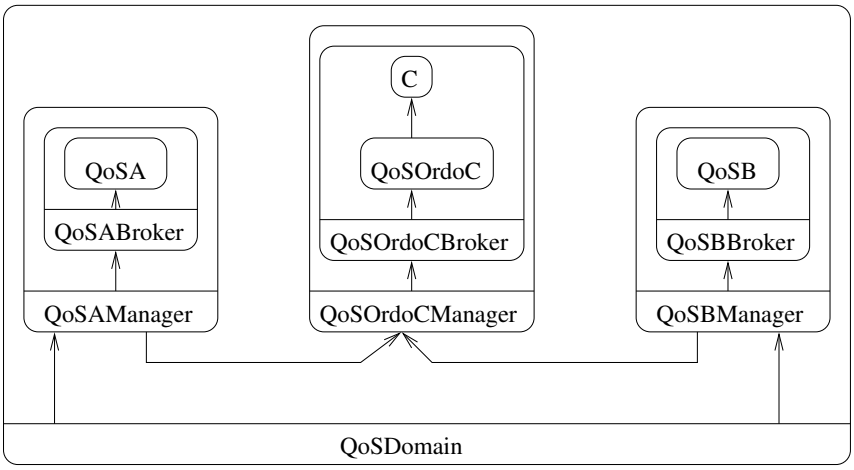

FIG. 4.8 – Intégration de Qinna pour le partage d'un composant insécable.

elle aussi, égale à la somme des niveaux de QdS ordonnancés, i-e : $QCourante_C = q_{objectif}$. Le rôle de QoSOrdoCBroker est d'implémenter l'opération d'admission, c'est-à-dire de déterminer si l'ordonnancement d'un niveau de QdS q donné est possible. Pour cela, il dispose des informations concernant le niveau de QdS en cours d'ordonnancement et du niveau de QdS maximal, ainsi que des opérateurs *compare(T_GC cg, add(T_CG cg_courant, T_QoS qobjectif))*.

A l'arrivée d'une requête de niveau q, le QoSOrdoCManager se base sur le QoSOrdoCBroker pour réaliser l'opération d'admission suivante :

$$compare(Q_C, add(QCourant_C, q))$$

Si la requête est acceptée, la contrainte globale courante, ainsi que le niveau de QdS objectif, deviennent alors égaux à $QCourant_C = add(QCourant_C, q)$.

4.1.2.3.2 Illustrations De même que pour le partage d'un composant sécable, nous pouvons illustrer ce cas type par le cas où le composant partagé est une ressource (CPU) et le cas où c'est un composant applicatif. De plus, nous étendons ce cas type par une illustration de Qinna pour la mise en oeuvre d'ordonnanceurs CPU hiérarchiques.

Composant ressource Prenons le cas où les composants A et B sont des composants de type thread requérant la ressource CPU. La ressource CPU réifiée sous forme de composant est un composant insécable. Il est donc nécessaire d'intégrer un composant d'ordonnancement permettant aux composants A et B de disposer du CPU. Nous prenons dans le cadre de cette illustration un ordonnancement classique du domaine du temps-réel qui est EDF [63] (EDF signifiant *Earliest Deadline First*). Nous définissons la QdS requise par A et B comme le taux d'utilisation du CPU modélisé par le couple de valeurs (E_a, P_a) et (E_b, P_b) avec E_i égal au délai d'exécution et P_i la période d'exécution du thread. De plus, d'après [63], pour un nombre de threads égal à N, un ordonnancement appliquant l'algorithme EDF est possible si et seulement si : $\sum_{i=1}^{N} \frac{E_i}{P_i} < 1$.

Après intégration des composants liés à Qinna, la contrainte globale de QoSOrdoCBroker est identifiée comme étant la charge maximale pouvant être ordonnancée : elle est donc égale à 1. De plus, la contrainte globale cou-

rante, notée $QCourant_C$, (ainsi que le niveau de QdS objectif de QoSOrdoC) est la somme des couples (E_i, P_i) des threads en cours d'ordonnancement. Lors de la demande d'activation du thread QoSA, le QoSEDFBroker détermine si le thread peut être ordonnancé en effectuant l'opération : $compare(1, add(QCourant_C, (E_a, P_a)))$, afin de vérifier que $QCourant_C + \frac{E_a}{P_a} < 1$. Dans le cas où les threads QoSA et QoSB sont exécutés la contrainte globale courante est alors égale à : $QCourant_C = add((E_a, P_a), (E_b, P_b))$, c'est-à-dire que $QCourant_C = \frac{E_a}{P_a} + \frac{E_b}{P_b}$.

Composant applicatif Prenons le cas où les composants A et B requièrent simultanément les services d'un composant de décodage vidéo, appelé DECO. Dans le cadre de cette illustration, nous considérons qu'une seule instance de DECO peut être exécutée. Le composant DECO est donc un composant insécable. Le niveau de QdS maximal pouvant être fourni par DECO est de 20 images/seconde. De plus, les composants A et B requièrent 10 i/sec chacun afin de fournir un niveau de QdS égal à BON sur les interfaces a et b.

Afin de pouvoir intégrer Qinna à ce système, il est nécessaire d'ajouter un composant permettant d'ordonnancer les accès de A et B à DECO. Dans le cadre de cette illustration, ce composant, nommé OrdoDECO, implémente une politique d'accès à DECO à partir d'un niveau de QdS exprimé en i/sec.

L'intégration de Qinna passe par la mise en place des composants QoSComponentBroker et QoSComponentManager pour les composants QoSA, QoSB et QoSOrdoDECO. La contrainte globale de QoSOrdoDECOBroker est fixée à la QdS maximale pouvant être fournie par DECO, c'est à dire que la contrainte globale est égale à 20 i/sec. La contrainte globale courante, et le niveau de QdS objectif de QoSOrdoDECO, est le niveau de QdS en cours d'ordonnancement : elle est notée $QCourante_{OrdoDeco}$ et est exprimée en i/sec. Lors de la demande d'activation du composant QoSA de niveau BON, une demande de réservation à DECO à 10 i/sec est effectuée au service *reserve* de l'interface *iQoSManager* du composant QoSOrdoDECOManager. Celui-ci transmet la requête au QoSOrdoDECOBRoker en appelant le service *reserve* de l'interface *iQoSBroker*, qui évalue s'il est possible d'ordonnancer cette demande. Pour cela, il se base sur sa contrainte globale et la contrainte globale courante afin de réaliser l'opération : $compare(20, add(QCourante_{OrdoDeco}, 10))$, c'est à dire $QCourante_{OrdoDeco} + 10 < 20$. Dans ce cas, étant donné qu'aucun composant n'est en cours d'ordonnancement, $QCourante_{OrdoDeco} = 0\, i/sec.$ Une fois que la requête de QoSA est acceptée $QCourante_{OrdoDeco} = 10\, i/sec.$ Enfin, lorsque la demande d'action de QoSB au niveau BON est effectuée, celle-ci est accepté et le niveau de la contrainte globale courante est égale à $QCourante_{OrdoDeco} = 10 + 10 = 20\, i/sec.$

Ordonnanceurs CPU hiérarchiques 1 Cette illustration se base sur les travaux de Goyal et al [37] portant sur la définition d'un framework d'ordonnanceurs CPU hiérarchique dans les systèmes d'exploitation multimédia. L'idée principale de ce framework est de pouvoir partager la ressource CPU entre différentes classes d'applications (temps-réel dur, temps-réel mou et best-effort), chacune d'elle étant ordonnancée suivant une politique différente.

Le framework ainsi défini est représenté sous forme d'arbre (cf. figure 4.9) : chaque noeud feuille représente les entités à ordonnancer (threads). Les noeuds intermédiaires représentent l'ordonnancement appliqué aux noeuds feuilles, tandis que le noeud racine, noté *root*, représente l'ordonnancement de type partage de temps appliqué aux noeuds intermédiaires : à chaque noeud intermédiaire i est associé un poids, noté r_i, permettant de déterminer le pourcentage d'accès au CPU que devra lui allouer le noeud racine. En conséquence, le pourcentage d'accès pour le noeud intermédiaire i est égal à :

$$B_i = \left(\frac{r_i}{\sum_{j=1}^{n} r_j} \right) \times 100$$

Enfin, le noeud racine de l'arbre permet l'accès à la totalité du CPU, son pourcentage d'accès est donc de $B_{root} = 100\%$.

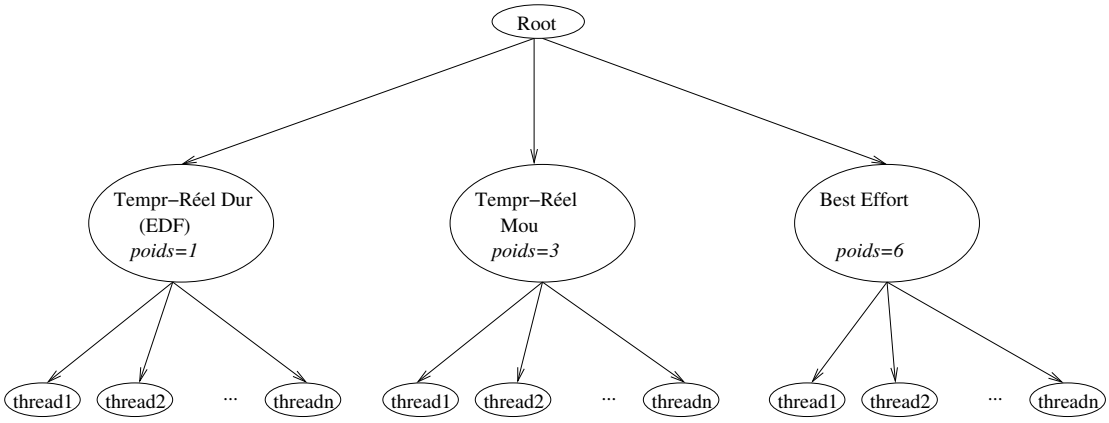

FIG. 4.9 – Représentation sous forme d'arbre du framework d'ordonnancement hiérarchique défini par [37].

On retrouve ici le cas type du partage de composant non sécable sur deux niveaux grâce à la représentation sous forme d'arbre du framework. Étant donné qu'il s'agit du partage de composants non sécables, il est nécessaire d'identifier les ordonnanceurs : le noeud racine, ainsi que les noeuds intermédiaires sont des ordonnanceurs. Tandis que les noeuds feuilles sont les threads à ordonnancer. Enfin, le paramètre B_i de chaque noeud ordonnancement représente la quantité de ressource pouvant être allouée par ce noeud.

L'intégration de Qinna est alors effectuée comme suit (cf. figure 4.10) :

– chaque thread est un QoSComponent. Il possède leur propre QoSComponentBroker et QoSComponentManager. La contrainte globale du QoSThreadBroker identifie le nombre maximal de threads pouvant être exécuté. Le niveau de QdS objectif identifie si le thread peut être ordonnancé ou pas d'après la contrainte portant sur le nombre de threads, tandis que la contrainte locale de chaque thread identifie les paramètres permettant un ordonnancement donné (par exemple, (E, P) pour un thread ordonnancé par EDF). Dans le cas où l'on souhaite limiter le nombre de threads par classe d'application, il est nécessaire de disposer d'un QoSThreadBroker et d'un QoSThreadManager dédiés à chaque type d'ordonnancement (cf. figure 4.10). Les contraintes globales des QoSThreadBrokers représentent alors le nombre de thread maximal autorisé par type d'ordonnancement.
– chaque noeud d'ordonnancement est un QoSComponent. A chacun de ces QoSComponents est alors associé un QoSComponentBroker et un QoSComponentManager. Les contraintes globales des QoSComponentBrokers liés aux noeuds intermédiaires sont égales à B_i. La contrainte globale courante de ces QoSComponents est égale à la charge courante devant être ordonnancée, notée $QCourant_i$. Les QoSComponentBrokers ont donc pour rôle de vérifier que $QCourant_i$ soit inférieure à B_i. Par exemple, à l'arrivée d'une nouveau thread, de QdS (e, p), pour l'ordonnanceur EDF, le QoSEDFBroker effectue l'opération suivante : $compare(B_{EDF}, add(QCourant_{EDF}, (e, p)))$. Enfin, La contrainte globale du QoSComponentBroker lié au noeud racine est égale à B_{root}. Sa contrainte globale courante, notée $QCourant_{root}$, est égale à la somme des pourcentages d'accès au CPU des noeuds intermédiaires : $QCourant_{root} = \sum_{i=1}^{n} B_i$. Le QoSRootBroker a donc en charge de vérifier que B_{root} soit inférieur à $QCourant_{root}$ grâce à l'opération : $compare(B_{root}, QCourant_{root})$.

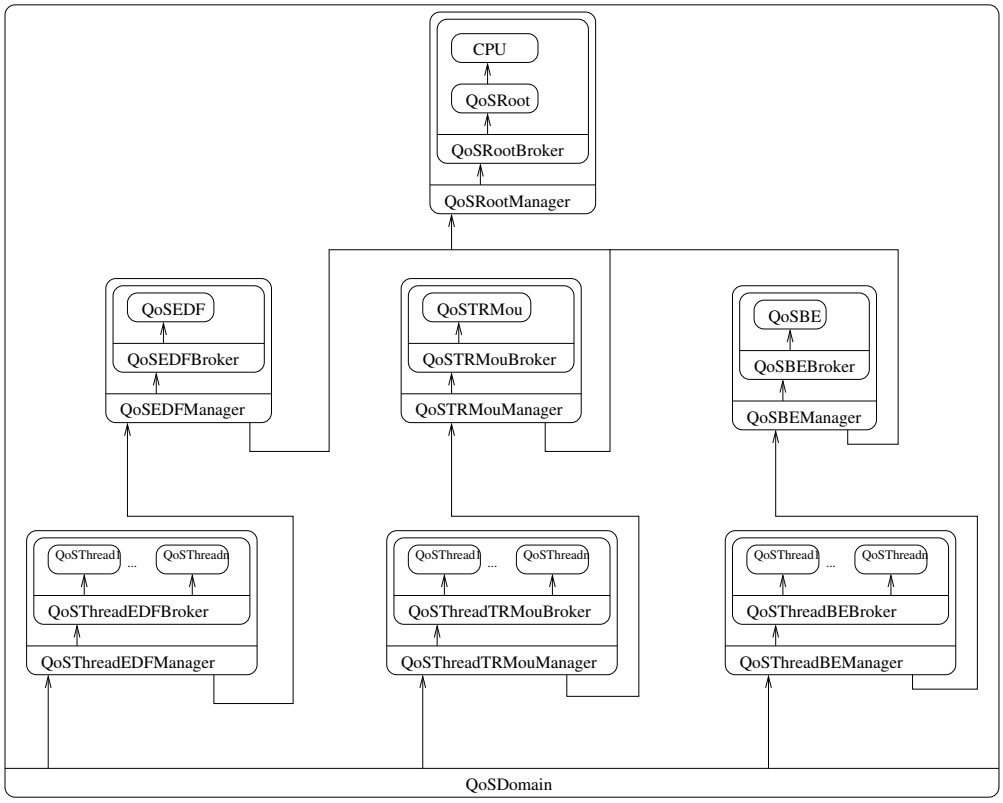

FIG. 4.10 – Intégration de Qinna au framework d'ordonnancement hiérarchique.

Ordonnanceurs CPU hiérarchiques 2 Une deuxième illustration intéressante d'ordonnancement hiérarchique porte sur le cas d'une machine virtuelle Java (JVM) s'exécutant sur un système d'exploitation temps-réel (cf. figure 4.11). Nous nous intéressons ici uniquement à l'aspect ordonnancement du système. L'ordonnanceur du système d'exploitation temps-réel implémente une politique de type EDF. Cet ordonnanceur gère un ensemble de tâches, dont l'une est dédiée à l'exécution de la JVM. La JVM possède elle-même un ordonnanceur de threads java. Cet ordonnanceur implémente une politique de type round-robin à partage de temps équitable.

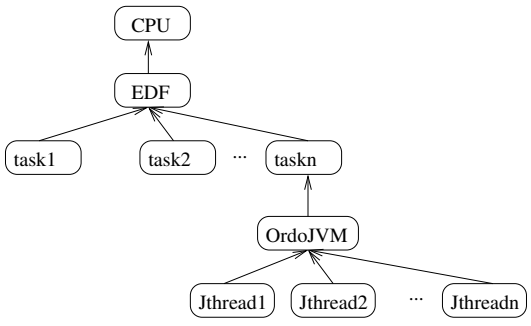

FIG. 4.11 – Représentation des tâches et des ordonnanceurs d'un système temps-réel exécutant une machine virtuelle Java.

L'intégration de Qinna (cf. figure 4.12) pour l'ordonnanceur EDF, ainsi que de ses tâches associées, s'effectue de la même manière que lors de la première illustration de partage de composant insécable. L'originalité de cette illustration est qu'une des tâches (tâche n) exécute elle-même un ordonnanceur (OrdoJVM). Cet ordonnanceur

est lui-même un QoSComponent possédant son propre QoSComponentBroker et QoSComponentManager. Étant donné que l'ordonnancement est de type round robin à partage de temps équitable, le test d'admission est toujours validé. La contrainte globale, le contrainte globale courante, et la contrainte locale de QoSOrdoJVM ne sont donc pas utilisées. En revanche, les threads Java, sont eux-mêmes des QoSComponents et la contrainte globale de leur QoSComponentBroker permet de limiter le nombre de threads (donc de limiter le nombre de threads par JVM).

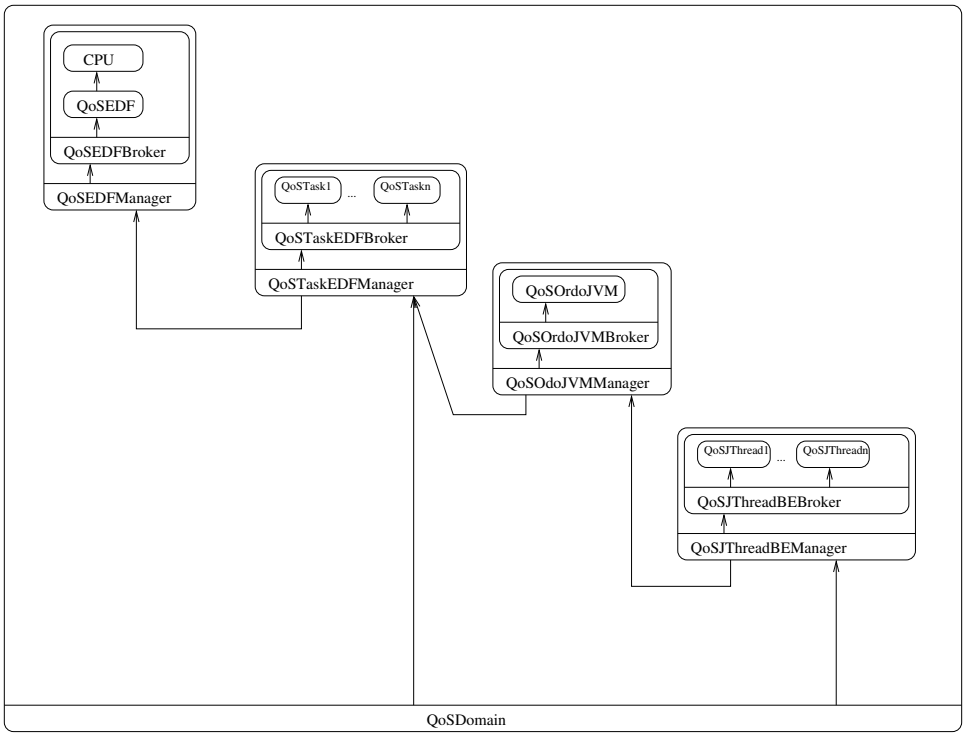

FIG. 4.12 – Intégration de Qinna au système de la figure 4.11 représentant des tâches et des ordonnanceurs d'un système temps-réel exécutant une machine virtuelle Java.

4.1.2.3.3 Analyse Dans le cas d'un partage de composant insécable, il est nécessaire de réifier l'ordonnancement sous forme de composants. Afin de pouvoir intégrer Qinna, le test d'ordonnançabilité doit être disjoint de l'ordonnancement lui-même. Dans le cadre des composants ressources (CPU, réseau), c'est une démarche classiquement utilisée. Dans le cadre des composants applicatifs, c'est une démarche plus originale. En effet, l'ordonnancement de l'application est généralement couplé à l'application elle-même. La démarche proposée ici permet de séparer clairement les préoccupations : le composant applicatif est en charge du code fonctionnel et de sa configuration alors que le composant d'ordonnancement effectue le contrôle d'accès à l'application. Cette approche permet de modifier la politique de gestion de l'application sans avoir à modifier l'application, comme par exemple, dans les architectures à méta niveau telles que RTGOL [7].

De plus, l'approche proposée par Qinna permet d'appliquer les concepts de façon homogène et d'en effectuer des combinaisons (par exemple, dans le cas des ordonnanceurs multi-niveaux). Enfin, elle permet de clairement identifier les rôles de chacun des éléments (les QoSThreadBrokers limitent le nombre de threads, tandis que les QoSOrdoBrokers effectuent les tests d'ordonnancement).

4.1.3 Hétérogénéité d'expression de QdS

4.1.3.1 Problématique

Les différents composants constituant un système peuvent provenir d'équipes de développement différentes. De plus, du fait de la diversité des langages de spécification de QdS, mais aussi des domaines d'applications, deux composants ayant un lien fonctionnel peuvent exprimer leur niveau de QdS de façon différente. Par exemple, un composant utilisant un thread initialement prévu pour s'exécuter sur un ordonnanceur temps réel mou, peut être réutilisé, dans un autre contexte, pour s'exécuter sur un ordonnanceur de type temps réel dur. Il est donc nécessaire de pouvoir traduire une spécification de QdS initiale en une spécification manipulable par l'entité concernée.

4.1.3.2 Présentation

Considérons deux composants A et B. Le composant A, pour fournir l'interface a, requiert l'interface b du composant B. Le composant A requiert sur b un niveau de QdS noté q de type *T_QoS1*, tandis que le composant B fournit sur b un niveau de QdS de type *T_QoS2*.

4.1.3.3 Prise en compte par Qinna

Une fois l'architecture intégrée aux composants A et B, la requête de niveau de QdS q de type *T_QoS1* est effectuée auprès du composant QoSBManager. Qinna définit un opérateur *T_QoS2 translate(T_QoS1 q)* permettant de convertir un niveau de QdS donné en un niveau compréhensible afin que la requête soit traitée.

L'opérateur *translate* correspond à trois types d'opérations :

- soit il effectue un changement d'unité permettant de convertir q en type *T_QoS2*. C'est le cas lorsque *T_QoS1* et *T_QoS2* désignent la même grandeur. Par exemple, dans le cas où *T_QoS1* est exprimé en kilo octets et que *T_QoS2* est exprimé en octets l'opérateur *translate* effectue une conversion simple ($\times 1000$). Il est à noter que cette opération peut engendrer une perte de précision sur le niveau de QdS manipulé. Par exemple, dans le cas où *T_QoS2* doit représenter un nombre entier de kilo octets et que *T_QoS1* est exprimé en octets, la division par 1000 peut engendrer une perte d'information. Dans ce cas, l'opérateur devra retourner l'entier supérieur du résultat de la division, ou l'entier inférieur mais au risque d'entraîner des violations de contrats.
- soit l'opérateur effectue une changement sémantique de *T_QoS1* pour le convertir en *T_QoS2*. Par exemple, dans le cas où *T_QoS1* est de type fonction, notée f, et que *T_QoS2* est du type constante, notée c, l'opérateur *translate* effectue une analyse de f telle que, par exemple, $c = Max(f)$. Dans ce cas, le risque de gaspillage de QdS peut être important (cf. analyse du cas type *profil de QdS variable* page 81). Si l'opérateur retourne c, tel que $c = Min(f)$, le coût, en nombre d'opérations de maintenance, sera élevé dû aux violations de contrat prévisibles. De même, lorsque les valeurs des niveaux de QdS de la table de mapping sont discrétisées et qu'une requête de QdS n'est pas égale à un des niveaux discrétisés, l'opérateur peut choisir soit la valeur directement supérieure, soit la valeur directement inférieure. Dans le premier cas, il y aura un gaspillage de QdS, tandis que dans le deuxième cas il y aura un risque de violation de contrat.
- enfin, lorsque l'opérateur ne peut analyser, ni convertir *T_QoS1*, celui-ci retourne la valeur *default*.

4.1.3.4 Illustrations

Nous présentons ici deux illustrations. Bien que celles-ci illustrent le cas où l'opération *translate* effectue un changement sémantique, les conséquences de ce changement ne sont pas les mêmes. Nous présentons tout d'abord le cas où un thread apériodique temps réel demande à être exécuté par un ordonnanceur périodique temps-réel [85], puis le cas où un thread de type temps-réel relaché est exécuté sur un ordonnanceur temps-réel dur.

4.1.3.4.1 Illustration 1 Illustrons tout d'abord le cas où une tâche apériodique demande à être ordonnancé par un ordonnanceur périodique temps-réel de type EDF. Le niveau de QdS requis par la tâche est uniquement exprimé au travers d'un paramètre, noté E_t représentant sa durée maximale d'exécution. Elle ne possède donc pas de période connue. L'ordonnanceur EDF exprime, classiquement, ses niveaux de QdS fournis par le couple de valeurs (E_i, D_i), ou (E_i, P_i), avec E_i la durée maximale d'exécution de la tâche i et D_i l'échéance de la tâche i (qui est égale à la période, P_i, de la tâche i).

Une fois l'architecture Qinna intégrée, la requête de niveau E_t est transmise par le QoSTaskManager au QoSEDFManager. Une solution pour l'opérateur *translate* est de convertir E_t en un couple de valeurs (E_t, D_t), où la valeur de D_t est fixée par l'algorithme proposée par Spuri et Butazzo [85]. Cet algorithme se propose d'ordonnancer au mieux des requêtes apériodiques pour un ordonnanceur EDF. Soit D_{t-1} l'échéance de la dernière tâche apériodique activée, alors l'échéance, D_t, de la tâche apériodique est égale à :

$$D_t = Max(now, D_{t-1}) + \frac{E_t}{U_a}$$

avec now étant la date d'arrivée de la requête et U_a représentant la charge libre pour les tâches apériodiques : $U_a + U_p < 1$ et $U_p = \sum_{i=1}^{Nb\_tches\_priodiques} \frac{E_i}{P_i}$.

4.1.3.4.2 Illustration 2 La deuxième expérimentation présente le cas où une tâche temps-réel relachée (noté TRR) s'exécute sur un ordonnanceur temps-réel dur (noté TRD). La QdS d'une tâche TRR est exprimée via son échéance et est spécifiée sous la forme d'un intervalle ordonné noté $[d_1; d_2]$. Une échéance égale à d_1 correspond à une bonne réponse alors qu'une échéance égale à d_2 correspond à une mauvaise réponse de la tâche. La QdS fournie par un ordonnanceur TRD, représentant l'échéance garantie à la tâche, est spécifiée sous la forme d'une valeur fixe notée d. Lors d'une demande d'activation d'une tâche sur l'ordonnanceur, l'opération *translate* doit donc choisir une valeur de d comprise entre d_1 et d_2.

Une implémentation optimiste de *translate* est de choisir tout d'abord la valeur minimale de d, c'est à dire $d = d_1$, puis de l'augmenter d'un pas d_{step} jusqu'à la valeur d_2 si le test d'ordonnançabilité échoue. Cette approche permet de disposer de l'échéance la plus petite possible pour la tâche, mais peut être en revanche très coûteuse. En effet, si le test d'ordonnançabilité est uniquement validé pour $d = d_2$, l'architecture aura donc effectué $\frac{d_2 - d_1}{d_{step}} + 1$ tests d'ordonnançabilité.

Une implémentation pessimiste de *translate* est de choisir tout d'abord la valeur maximale de d, c'est à dire $d = d_2$, puis de la diminuer d'un pas d_{step} jusqu'à ce que le test d'ordonnançabilité échoue. Cette approche peut aussi être coûteuse. En effet, si le test d'ordonnançabilité est validé jusqu'à $d = d_1$, l'architecture aura donc effectué $\frac{d_2 - d_1}{d_{step}} + 1$ tests d'ordonnançabilité.

Une troisième implémentation consiste à choisir soit d_1, soit d_2 et de ne pas tenter de modifier cette valeur si le test d'admission est accepté. Dans ce cas, l'opérateur est moins coûteux en termes de tests d'admission à réaliser, mais en contre partie il ne choisit pas une valeur de d optimale.

Le cas inverse de cette illustration est d'ordonnancer une tâche TRD sur un ordonnanceur TRR. La tâche spécifie sa QdS par la valeur d, alors que l'ordonnanceur garanti une QdS de la forme $[d_1; d_2]$. Dans ce cas, l'opération *translate* doit définir un pourcentage, noté p, afin de traduire d en $[d \times (1 - p); d]$. Il est à noter que p peut être égal à 0% auquel cas la tâche sera exécutée comme sur un ordonnanceur TRD, mais dans ce cas les avantages liés à l'ordonnancement TRR sont perdus.

4.1.3.5 Analyse

La prise en compte de l'hétérogénéité d'expression de la QdS amène généralement à effectuer un changement sémantique du niveau de QdS requis. Cette façon de prendre en compte l'hétérogénéité aboutit à un surcoût pris en charge par l'architecture. En effet, ce sûrcout est présent à l'initialisation du contrat. De plus, lorsqu'un niveau de QdS ne peut être analysé (niveau de QdS alors égal à *default*), les coûts impliqués sont présents aussi bien à l'initialisation qu'à l'exécution du contrat (cf. 4.2.2 page 81). Enfin, lorsque le changement de QdS aboutit à un changement d'unité, la précision du niveau de QdS peut être altérée. Le coût de la perte de précision d'un niveau de QdS peut se traduire soit par un gaspillage de QdS, dans le cas où la QdS traduite est supérieure à la QdS originale, soit par des opérations de maintenance liées aux violations de contrats, dans le cas où la QdS traduite est inférieure à la QdS originale.

L'hétérogénéité de la QdS impacte donc le comportement de l'architecture, mais sa prise en compte est primordiale pour les systèmes embarqués ouverts. Qinna permet de poser le problème de sa prise en compte en identifiant les opérateurs à mettre en place.

4.2 Comportement dynamique

Tandis que la section précédente se focalisait sur l'aspect statique de l'architecture, nous analysons ici son comportement dynamique au travers de cas types comportementaux. Le premier porte sur la mise en place de contrats et permet d'analyser le coût de la gestion dynamique des contrats de QdS. Le deuxième traite de l'adaptation dynamique à des profils variables de QdS, tandis que le troisième cas type analyse le comportement de Qinna lorsque la capacité d'une ressource est modifiée dynamiquement. Enfin, le dernier cas se concentre sur l'évolution dynamique de la relation d'ordre des niveaux d'importance.

4.2.1 Coût de gestion des contrats de QdS

4.2.1.1 Problématique

A partir d'un système intégrant l'architecture Qinna, l'activation d'un QoSComponent aboutit à la mise en place de contrats. Nous évaluons ici la façon dont ces contrats sont mis en place dans Qinna. Pour cela, nous

étudions la mise en place des contrats lors de l'activation d'un composant.

4.2.1.2 Présentation

Soit le cas générique où un composant A requiert les interfaces de n composants différents (notés de A_1 à A_n), qui eux-mêmes requièrent n_i composants (notés de A_{i_1} à $A_{i_{n_i}}$), qui eux-mêmes requièrent $n_{i_{n_i}}$ composants (notés de $A_{i_{i_1}}$ à $A_{i_{i_{ni}}}$), et ce, sur p niveaux de dépendance (cf. figure 4.13).

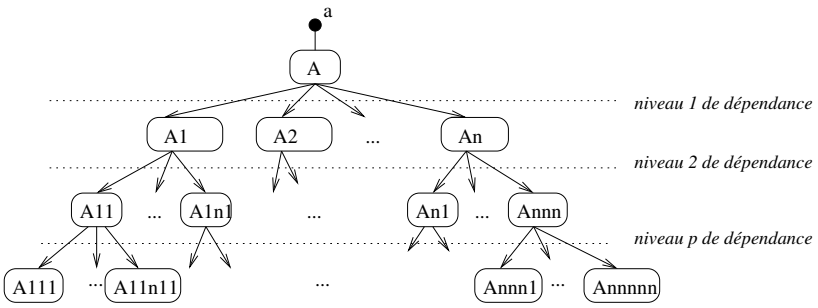

FIG. 4.13 – Représentation arborescente d'une composition et de ses liaisons.

L'intégration de Qinna pour ce système aboutit à la mise en place d'un QoSDomain de QoSComponentBrokers et QoSComponentManagers pour chacun des composants. Les relations entre les différents QoSComponentManagers sont à l'image de celles existantes entres les QoSComponents (cf. figure 4.14). Il est à noter que les QoSComponentManagers liées aux ressources matérielles sont les noeuds feuilles de l'arbre car ils ne possèdent pas de dépendance.

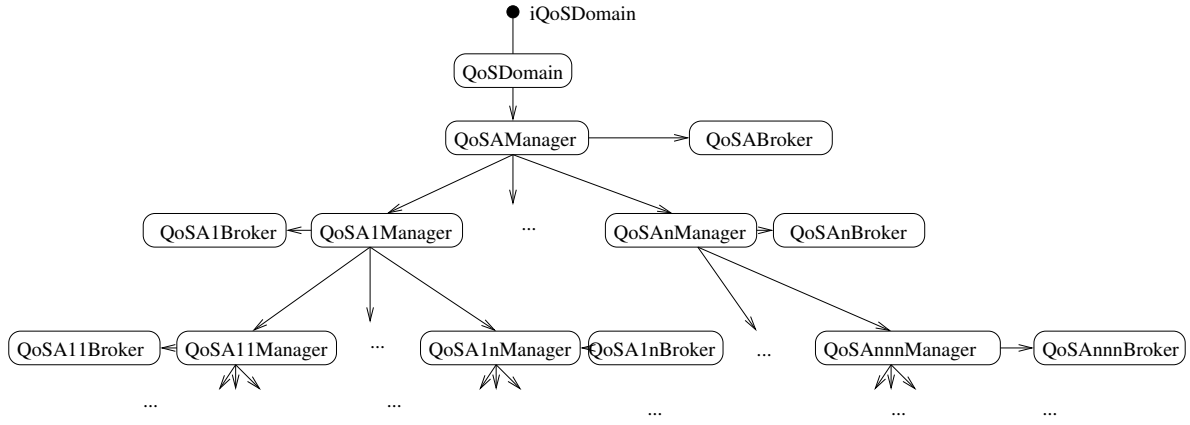

FIG. 4.14 – Représentation arborescente des liaisons des QoSComponentManagers après intégration de Qinna du système représenté à la figure 4.13.

4.2.1.3 Actions menées par Qinna

La demande d'activation d'un QoSComponent aboutit à la mise en place d'un contrat entre le QoSDomain et l'utilisateur. La mise en place de ce contrat implique la mise en place de l'ensemble des sous-contrats identifiant les dépendances entre les QoSComponents. Dans le cas générique, où chaque QoSComponenti possède n_i

dépendances, le nombre de sous contrats devant être mis en place est égal au nombre de liaisons entre les QoSComponentManagers :

$$N = 1 + \sum_{i=1}^{QCM} n_i$$

où QCM est le nombre de total de QoSComponentManagers autres que ceux gérant les ressources matérielles (c'est à dire le nombre de noeuds de l'arbre de la figure 4.13 autres que les noeuds feuilles), n_i le nombre de dépendance du $QoSA_iManager$ et N est le nombre de sous contrats à établir. Dans le cas particulier, où chaque QoSComponent requière n QoSComponents, le nombre de sous contrats à établir est à : $N = n \times QCM$.

Afin de s'assurer que chaque contrat soit mise en place, Qinna effectue un parcours en profondeur de l'arbre représenté à la figure 4.14. En effet pour chaque noeud, représentant un QoSComponentManager, Qinna explore le noeud de niveau inférieur. Un noeud établit un contrat avec son noeud père lorsque tous les contrats avec ses noeuds fils sont établis.

L'exploration d'un noeud est réalisée lors de l'appel du service *reserve* de l'interface *iQoSManager*, tandis que la connaissance des noeuds suivants est donné par la table de mapping de chaque QoSComponentManager. Pour chaque niveau de QdS fourni q, la table de mapping peut retourner q_{level} combinaisons différentes de QdS requises.

La complexité de ce type de parcours exhaustif de l'arbre est en $\theta(N)$ où N est le nombre total de noeuds à parcourir. Dans l'approche proposée par Qinna, lorsqu'un contrat est possible il est établi (*reserve* est un service de type *test_and_set*). Donc, si l'établissement d'un sous contrat échoue, tous les sous contrats précédemment établis doivent être annulés. Si pour chaque niveau de QdS fourni il n'existe qu'une seule combinaison des niveaux de QdS requis ($q_{level} = 1$), alors dans le pire cas, l'échec de la mise en place du dernier sous contrat revient à parcourir deux fois l'arbre. En effet, il faut un premier parcours pour établir les $N - 1$ sous contrats et un deuxième parcourir pour les annuler.

Lorsque l'établissement d'un sous contrat échoue, le QoSDomain en est informé. Celui-ci peut alors prendre la décision de dégrader les contrats en cours d'exécution de niveau d'importance inférieur (mise en oeuvre de la politique d'adaptation). Pour chaque contrat de niveau d'importance inférieur, identifié grâce à l'opérateur *compare(T_IMP i1, T_IMP i2)*, et en commençant par le contrat minimum, le QoSDomain le dégrade au niveau de QdS suivant par l'appel du service *degrade* de l'interface *iQoSManager*. Après chaque dégradation, la demande d'activation du QoSComponent est re-émise : il y a donc de nouveau un parcours de l'arbre de la figure 4.14. Si l'activation échoue de nouveau, les contrats suivants sont dégradés. Lorsque tous les contrats de niveau d'importance inférieur sont dégradés et qu'il n'est toujours pas possible d'activer le QoSComponent, l'utilisateur en est alors informé et les contrats précédemment dégradés sont re-évalués à leur niveau de QdS initiaux par l'appel du service *upgrade* de l'interface *iQoSManager*.

Il est à noter que lorsque l'activation du QoSComponent aboutit après une dégradation des contrats existants, il est nécessaire de re-évaluer les contrats dégradés afin de maximiser le niveau de QdS fourni. En effet, le niveau de QdS libéré lors des dégradations peut être récupéré. Par exemple, du fait des discrétisations des niveaux de QdS, si 5 ko de mémoire ont été libéré lors des dégradations et que le nouveau QoSComponent n'en réserve que 3 ko, les 2 ko restant peuvent être redistribuées à un composant précédemment dégradé afin de respecter le principe de maximisation des niveaux de QdS contractualisés.

Lors de la demande d'activation, si le QoSDomain possède M contrats de niveaux d'importance inférieur possédant chacun d niveaux de dégradations, l'arbre de la figure 4.14 sera alors parcouru dans le pire cas :

$$nb\_parcours\_max = 1 + M \times d$$

De plus, dans le pire cas, c'est à dire lors de l'échec de l'activation d'un composant, il y a $M \times d$ dégradations de contrat suivi de $M \times d$ annulation des dégradations (cf. figure 4.15).

La figure 4.15 représente l'enchaînement des opérations réalisées lors d'une demande d'activation d'un QoSComponent, ainsi que le nombre minimum et maximum d'exécutions de chacune des opérations.

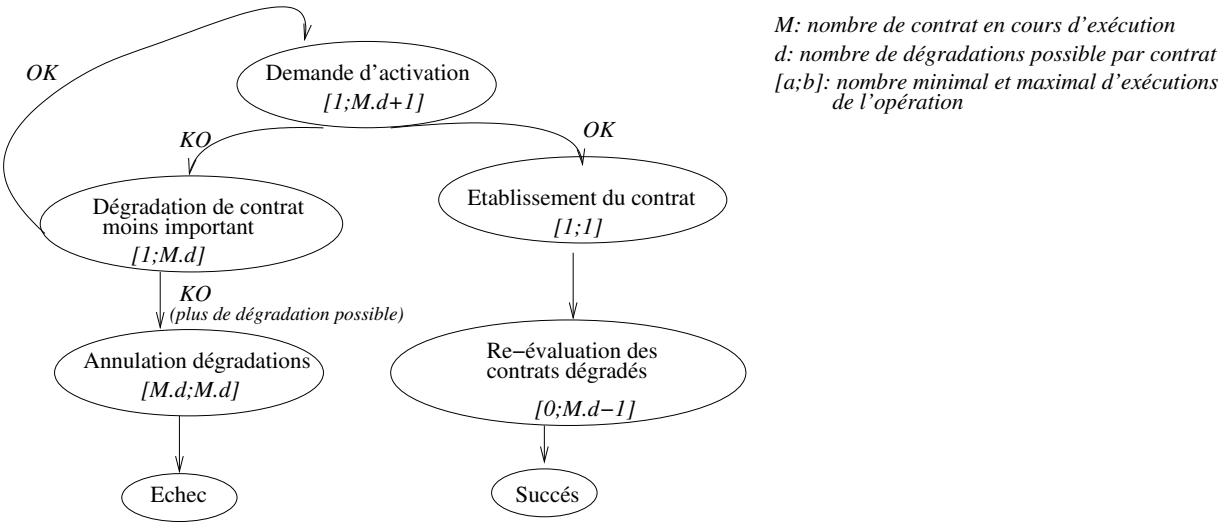

FIG. 4.15 – Enchaînement des opérations lors de l'activation d'un QoSComponent.

4.2.1.4 Illustrations

4.2.1.4.1 Illustration 1 En première illustration, prenons le cas simple où deux composants A et B utilisent chacun un composant mémoire, via les composants Ma et Mb, afin de fournir leur interface respective a et b.

Nous considérons que A et B peuvent fournir deux niveaux de QdS, BON et MAUVAIS, nécessitant respectivement 7 et 3 ko de mémoire. Nous fixons la capacité totale de la mémoire à 10 ko. De plus, il existe deux niveaux d'importance, 1 et 2, avec $Imp(1) > Imp(2)$.

Le scénario est alors le suivant : une demande d'activation du composant B est tout d'abord effectuée avec un niveau de QdS égal à BON et d'importance 2. Puis, une demande d'activation du composant A de niveau BON et d'importance 1 est réalisée.

Une fois Qinna intégrée à ce système, l'activation du composant QoSB avec un niveau de QdS égal à BON et d'un niveau d'importance à 2 est réalisée et validée puisque aucun autre QoSComponent n'est activé. Puis, lorsque une deuxième demande d'activation du composant QoSA de niveau BON et d'importance 1 est effectuée, le test

Niveau de QdS	A	B	C
BON	45 ko	65 ko	20 ko
MOYEN	37 ko	35 ko	15 ko
MAUVAIS	28 ko	23 ko	10 ko

TAB. 4.1 – Quantité de mémoire requis par les composants A,B et C pour chaque niveau de QdS.

d'admission effectué au niveau du QoSMemBroker échoue ($7ko + 7ko > 10ko$). Le composant QoSB est alors dégradé à MAUVAIS (utilisation de 3 ko de mémoire) car l'importance de QoSB est moindre que celle de QoSA. Puis, QoSA est fixé à BON, car l'admission est acceptée ($3ko + 7ko = 10ko$).

4.2.1.4.2 Illustration 2 La deuxième illustration permet de montrer que l'architecture maximise les niveaux des contrats de QdS. Soit trois composants, A, B et C, pouvant fournir trois niveaux de QdS différents : BON, MOYEN et MAUVAIS. Les niveaux de QdS fournit par chacun de ces composants dépendent de la quantité de mémoire utilisée (cf. tableau 4.1). Nous fixons la capacité totale de la mémoire à 100 ko. De plus, il existe trois niveaux d'importance, 1, 2 et 3 tels que $Imp(1) > Imp(2) > Imp(3)$.

Le scénario évalué est alors le suivant : une première demande d'activation de C de niveau BON et d'importance 3 est effectuée. Puis, une seconde demande d'activation de B de niveau BON et d'importance 2 est réalisée, suivie d'une demande d'activation de A de niveau BON et d'importance 1.

Une fois l'architecture Qinna intégrée, l'activation du composant QoSC de niveau BON et d'importance 3 est réalisée, ainsi que celle du composant QoSB de niveau BON et d'importance 2. La mémoire allouée est alors de : $65ko + 20ko = 85ko < 100ko$. Lors de l'activation du composant QoSA de niveau BON et d'importance 1, le test d'admission échoue au niveau du QoSMemBroker car la mémoire demandée est supérieure à celle disponible ($85ko + 45ko > 100ko$). Le contrat le moins important, celui lié à QoSC, est alors dégradé de BON à MOYEN, puis de MOYEN à MAUVAIS. Étant donné que le test d'admission ne peut toujours pas être validé ($10ko + 65ko + 45ko > 100ko$), le contrat lié à QoSB est à son tour dégradé de BON à MOYEN permettant ainsi d'activer QoSA à BON ($10ko + 35ko + 45ko < 100ko$).

Enfin, le QoSDomain maximise les niveaux de QdS des contrats dégradés. Le contrat de QoSB ne peut être amélioré ($10ko + 65ko + 45ko > 100ko$), mais celui lié à QoSC peut remonter de MAUVAIS à BON ($20ko + 35ko + 45ko < 100ko$) sans perturber l'exécution de QoSA, ni de QoSB.

Cet exemple illustre l'impact de l'implémentation de la recherche d'une bonne solution en nombre d'opérations effectuées. Il pose alors le problème plus général de la recherche du bon contrat par processus itératif.

4.2.1.5 Analyse

L'activation d'un QoSComponent peut être coûteuse du fait de l'établissement des sous contrats. En effet, nous avons vu que l'établissement d'un contrat entraîne au minimum l'établissement de N sous contrats et au maximum l'établissement puis l'annulation de $1 + M \times d \times (N - 1)$ sous contrats. La minimisation du coût d'établissement de contrat est alors un problème d'optimisation de parcours du graphe de dépendance. A l'implémentation, il peut

être indispensable d'effectuer le parcours de manière optimisée. Trois stratégies sont alors envisageables :

- la première consiste à privilégier l'exploration des sous noeuds ayant par le passé mis en échec l'établissement d'un contrat.
- pour la seconde, le QoSDomain peut effectuer un pré-parcours du graphe afin de disposer directement de toutes les informations nécessaires. En contre partie l'architecture perd en flexibilité, car cela revient à n'avoir qu'un seul composant gérant toute la QdS de tous les composants.
- la troisième stratégie consiste à effectuer une analyse a priori du système et de remplacer les tests d'admission par les résultats pré-analysés. Cette approche réduit considérablement l'utilité de Qinna, et n'est pas possible dans un contexte fortement dynamique. Par exemple, l'évaluation de la consommation mémoire d'un composant dépend fortement de son contexte d'exécution. Il peut être alors difficile d'extraire le niveau de QdS mémoire requis. Ce pose alors le problème des profils variables de QdS traité ci après (cf. section 4.2.2 page 81).

Enfin, se pose le problème du découpage d'un contrat global en sous contrats. Ceci pose alors le problème plus généralement de la granularité des composants. Plus un contrat est décomposé en sous contrats, plus son coût d'établissement est élevé (parcours d'établissement du contrat, adaptation) mais plus sa gestion de QdS est fine. En effet, lors de l'établissement du contrat on peut précisément identifier le composant faisant échouer le contrat. Une solution permettant de limiter le coût des composants est de les regrouper par applications afin d'obtenir un composant de plus gros grain.

4.2.2 Profil variable de QdS requise

4.2.2.1 Problématique

Le niveau de QdS requis d'un composant n'est pas forcément constant ou prédictible et peut évoluer en cours d'exécution du composant. Il est alors intéressant d'évaluer Qinna pour un profil générique de QdS correspondant à un pic de demande pour une ressource.

4.2.2.2 Présentation

Afin d'analyser la façon dont Qinna prend en compte ce cas type, nous considérons un composant requérant un profil de QdS représenté à la figure 4.16. Le niveau maximum de QdS requis est noté Q_{max}, tandis que le niveau minimum est noté Q_{min}. De plus, le profil de QdS considéré est périodique de période T. Les instants de début de croissance et de fin de décroissance de la QdS sont notés respectivement t_1 et t_2, tandis que les instants de début et de fin de stabilisation au niveau Q_{max} sont notés respectivement $t_1^{'}$ et $t_2^{'}$.

4.2.2.3 Prise en compte par Qinna

Afin de prendre en compte ce cas type, Qinna peut s'appuyer sur trois stratégies différentes :

- La première stratégie consiste à établir un contrat avec un niveau de QdS requis égal à Q_{max}. Cette première stratégie a deux conséquences : premièrement l'architecture doit pouvoir déterminer le niveau de QdS maximal, ce qui n'est pas toujours possible, et deuxièmement il y a un gaspillage important de la QdS dispo-

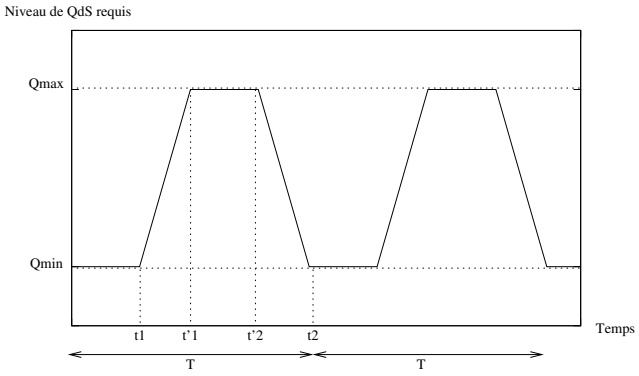

FIG. 4.16 – Profil de QdS requise variable.

nible. En effet, pour chaque période T, l'écart, noté E, entre le niveau de QdS réellement requis et le niveau contractualisé est évalué à :

$$E = (Q_{max} - Q_{min}) \times \left(\frac{(t_1^{'} - t_1) + (t_2^{'} - t_2)}{2} + (T - (t_2 - t_1)) \right)$$

Dans ce cas, le gaspillage de QdS dépend de la durée pendant laquelle le composant requiert Q_{max} et de l'écart entre Q_{max} et Q_{min}.

– La seconde stratégie consiste à exprimée la QdS requise sous forme de paramètres caractérisant la courbe de demande (soit ici Q_{max}, Q_{min}, T, t_1, t_2, $t_1^{'}$ et $t_2^{'}$) directement dans la table de mapping. Cela nécessite alors de pouvoir caractériser, de manière exacte ou de manière approchée, le profil de QdS et d'assurer que le QoSComponentManager traite ce type de spécification complexe.
– La troisième stratégie consiste à établir un contrat avec un niveau de QdS requis égal à Q_{min}. Il est ensuite nécessaire d'observer le niveau de QdS fourni par le composant afin de déterminer les instants où les niveaux de QdS requis augmente (t_1) ou diminue (t_2) et ainsi d'augmenter ou de diminuer les niveaux de QdS contractualisés. Cette stratégie est obligatoire lorsque le niveau maximal de QdS est inconnu à priori.

Dans le cadre de Qinna, cette stratégie correspond à fixer les valeurs de la table de mapping à Q_{min}, puis de la caractériser comme étant non fiable (*unreliable*) afin que l'architecture mette en place les QoSComponentObservers. Il est à noter que si la valeur de Q_{min} est inconnue, les valeurs sont fixées à *default*. Les QoSComponentObservers permettent ainsi de notifier au QoSDomain des niveaux de QdS fournis afin que celui-ci exécute la politique de maintenabilité. La politique de maintenabilité permet d'augmenter ou de diminuer les niveaux de QdS requis contractualisés.
Lorsqu'une violation de contrat est notifiée au QoSDomain, celui-ci demande d'augmenter les niveaux de QdS requis du QoSComponent concerné auprès du QoSComponentManager (service *upgrade* de l'interface *iQoSMaintener*). Le QoSComponentManager a alors en charge d'effectuer les demandes d'augmentation des niveaux de QdS requis auprès des différents QoSComponentManager d'un *pas de maintenabilité* noté p.
Enfin, il est à noter que si la politique de maintenance échoue, c'est à dire que les niveaux de QdS requis ne peuvent plus être augmentés, le QoSDomain dégrade le contrat de QdS.
Les paramètres rentrant en compte pour cette stratégie sont le pas de maintenabilité. Le choix de ce pas est considéré comme optimal, et noté $p_{optimal}$, lorsqu'il est égal à :

$$p_{optimal} = Q_{max} - Q_{min}$$

En effet, dans ce cas, à chaque changement de niveau de QdS, il n'y a qu'une seule opération de maintenance à effectuer. Cela permet donc, par rapport à la stratégie précédente, de réduire le gaspillage de QdS, mais pas de l'annuler. Le gaspillage de QdS est alors égal à :

$$E_{p_{optimal}} = (Q_{max} - Q_{min}) \times \left(\frac{(t_1^{'} - t_1) + (t_2^{'} - t_2)}{2} \right)$$

Si la valeur de Q_{max} ne peut être connue, le pas de maintenabilité est fixée à une valeur notée p. Dans ce cas, le nombre d'opérations de maintenance, noté OM, à effectuer pour chaque changement de niveau de QdS est évalué à :

$$OM = \left\lceil \frac{Q_{max} - Q_{min}}{p} \right\rceil$$

Il est à noter que dans ce cas, suivant la valeur de p, le niveau de QdS requis contractualisé peut-être supérieur à Q_{max}. Ce gaspillage de QdS est alors donné par :

$$E_p = Q_{max} - (Q_{min} + (OM \times p))$$

Le choix de la valeur de p est donc un compromis entre le nombre d'opérations de maintenabilité et le gaspillage de QdS. En effet, plus le pas p est petit et plus le nombre d'opérations de maintenabilité a effectuer est grand, mais plus le gaspillage de QdS est faible, et inversement.

Afin de notifier des violations de contrat, Qinna permet de mettre en place deux types d'observation différents (cf. section 3.2.3.3 page 46) : l'observation est initiée soit par le QoSComponentObserver, soit par le QoSComponent.
Dans le cadre d'une observation périodique, lorsque la première violation de contrat intervient à $t = t_1$, le QoSDomain en est notifié au pire à $t = t_1 + T_{obs}$. Cela signifie que pendant un délai égal à T_{obs}, le QoSComponent ne peut fournir son service. A $t = t_1 + T_{obs}$, les niveaux de QdS requis sont alors augmentés de p et le QoSComponent respectera de nouveau son contrat : le profil de QdS requis est alors décalé de T_{obs}. A chaque violation de contrat, le profil de QdS est ainsi décalé de T_{obs}. Lorsque la QdS requise est à son niveau maximal, le décalage du profil de QdS, noté D_{max}, est égal au nombre de violations de contrat multiplié par la période d'observation :

$$\begin{aligned} D_{max} &= OM \times T_{obs} \\ &= \left\lceil \frac{Q_{max} - Q_{min}}{p} \right\rceil \times T_{obs} \end{aligned}$$

Suite à une opération de maintenance, si le QoSDomain n'est pas notifié d'une violation de contrat au bout d'un délai égal à T_{obs}, il tente de diminuer les niveaux de QdS requis afin de minimiser le gaspillage de QdS dans le cas où le profil de QdS diminue. L'observation périodique permet alors de détecter les instants où les niveaux de QdS requis diminuent. Cette régulation peut entraîner des retards dû à une diminution de QdS offerte trop importante (posant ainsi le problème de la valeur du pas de maintenabilié) et/ou d'un profil particulier (par exemple, une descente lente du profil de QdS).
Dans le cadre d'une observation initiée par le QoSComponent, les violations de contrats sont notifiées au QoSDomain sans retard. En effet, dès que le QoSComponent s'aperçoit que les niveaux de QdS requis ne sont plus suffisants, il en informe le QoSComponentObserver qui transmet au QoSDomain. Bien que ce type d'observation permet d'augmenter, sans retard, les niveaux de QdS requis, il est est moins adapté pour suivre les descentes de QdS requises. En effet, le QoSComponent doit de lui-même évaluer le moment

où il gaspille trop de QdS et en informer le QoSComponentObserver. Cette décision est complexe et fait notamment intervenir la valeur du pas de maintenabilité (p) connue, lors de l'exécution, uniquement au niveau des QoSComponentManagers. En effet, pour que la notification soit pertinente, il faut que la QdS gaspillée soit supérieure à p. Cela signifie que le QoSComponent doit contenir des politiques liées à la gestion de la QdS ce qui est contraire aux principes de séparation des préoccupations de Qinna.
Au final, l'observation initiée par le QoSComponent est plus pertinente si le niveau de QdS requis est stable une fois atteint son niveau maximal. En effet, cette stratégie permet de suivre sans retard les demandes croissantes de QdS requises. Par contre, elle n'est pas adaptée au pic de demande de QdS car les demandes décroissantes de QdS requises sont difficiles à évaluer.

4.2.2.4 Illustration

Cette illustration présente la mise en oeuvre de la troisième stratégie utilisant une observation périodique initiée par le QoSComponentObserver pour un composant ne connaissant pas son niveau de QdS requis.

Considérons un composant A nécessitant un composant réseau pour s'exécuter. Le composant A peut fournir un niveau de QdS égal à BON, mais ne connaît pas la quantité de bande passante requise correspondante. Nous évaluons l'architecture pour un profil de QdS requis tel qu'il est donné par la figure 4.16 page 82 avec $Q_{max} = 40\%$ et $Q_{min} = 10\%$ de la bande passante.

Lors de l'intégration de Qinna, le niveau de QdS requis par QoSA est fixé à 10%, tandis que le pas de maintenabilité est lui aussi fixé à 10% et la période d'observation de QoSAObserver est fixée à 1 unité de temps.

Lorsque le composant QoSA est activé avec un niveau de QdS égal à BON ces niveaux de QdS requis évoluent suivant la politique de maintenabilité du QoSDomain. La figure 4.17 représente le niveau de QdS requis original, c'est à dire sans retard dû à l'observation, le niveau de QdS réellement contractualisé et le niveau de QdS requis avec retard dû à l'observation périodique. A chaque modification du niveau de QdS requis, $OM = \frac{40-10}{10} = 3$ opérations de maintenance sont effectuées. Le niveau maximal est alors atteint avec un retard égal à $D = OM \times T\_Obs = 3 \times 1 = 3$ unités de temps. De plus, à chaque stabilisation du niveau de QdS, étant donné que le QoSDomain n'est plus notifié d'une violation de contrat, tente de diminuer les niveaux contractualisés jusqu'à ce que le profil diminue.

4.2.2.5 Analyse

Qinna peut traiter un pic de demande de QdS de deux façons différentes : soit de façon statique (le niveau de QdS maximal est contractualisé ou le niveau de QdS réel est contractualisé par la fonction du niveau de QdS requis), soit de façon dynamique (le niveau minimal est contractualisé puis adapté).

La première approche consistant à contractualiser le niveau maximal de QdS est intéressante si l'écart entre les niveaux de QdS maximal et minimal est faible, ou si la durée du niveau de QdS requis au minimum est courte. En effet, dans ces deux cas le gaspillage de QdS est minimisé et cette approche permet d'éviter toutes les adaptations de contrat liées aux politiques de maintenabilité.

La seconde approche est celle permettant d'éviter le gaspillage de QdS et ne nécessite aucune opération sur

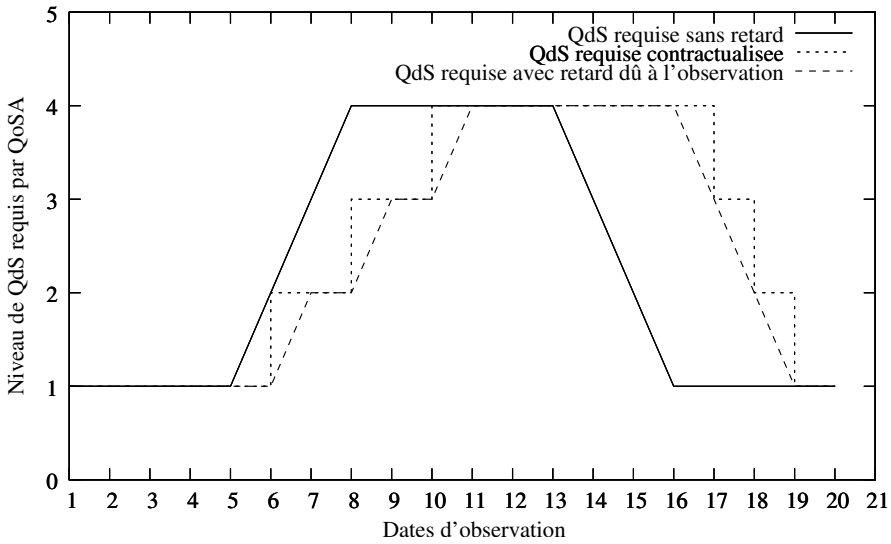

FIG. 4.17 – Évolution du niveau de QdS requis contractualisé par QoSA.

le contrat. Cependant, elle nécessite une connaissance fine des profils des niveaux de QdS requis et la mise en oeuvre de tests d'admission complexes ce qui est difficile, voire souvent impossible, à déterminer dans un contexte applicatif donné.

La dernière approche peut être coûteuse en termes d'opérations de maintenabilité à réaliser. Cependant, il s'agit de la seule approche possible lorsque les niveaux de QdS requis sont inconnus. De plus, par rapport à la première approche, le gaspillage de QdS est moindre. Cette approche n'est pas adaptée aux profils de QdS fortement variables, c'est à dire lorsque la période du pic de demande est faible. En effet, dans ce cas l'architecture est toujours en train de réaliser des opérations de maintenance. Cette approche a permis d'étudier les deux types d'observations offerts par Qinna. Bien que l'observation initiée par les QoSComponentObservers implique un retard sur les services rendus par les QoSComponents, elle permet de détecter les diminutions des niveaux de QdS requis. Cependant, l'observation initiée par les QoSComponents est particulièrement adaptée pour déterminer des niveaux de QdS inconnus et stables à partir d'un niveau de QdS minimal.

Cette étude permet de mettre en avant la complexité d'une optimisation de QdS sous plusieurs critères. En effet, dans le cadre d'une observation initiée par le QoSComponentObserver, Qinna impacte le comportement temporel des composants en créant des retards, ce qui modifie les propriétés de QdS temporelles.

4.2.3 Modification de la capacité d'une ressource

4.2.3.1 Problématique

La capacité d'une ressource matérielle n'est pas forcément constante tout au long du cycle de vie du système, mais peut évoluer en fonction de l'environnement (par exemple pour le débit sur le réseau) ou du temps (par exemple dans le cas où le niveau de charge de la batterie influe sur la vitesse du CPU pour des processeurs à vitesse variable en fonction du niveau d'énergie). A partir d'un système intégrant l'architecture Qinna, la capacité d'une ressource est identifiée par la contrainte globale du QoSComponentBroker de la ressource. La modification d'une

contrainte globale aboutit dans Qinna à une renégociation globale des contrats en cours d'exécution utilisant cette ressource.

Le principal problème soulevé par ce cas type est d'identifier les politiques de mise à jour de la contrainte globale par rapport à la capacité de la ressource.

4.2.3.2 Présentation

Soit le cas générique d'un système intégrant l'architecture Qinna et possédant N contrats de QdS en cours d'exécution (cf. figure 4.18). De plus, nous considérons que chaque contrat possède n_i niveaux de QdS distincts.

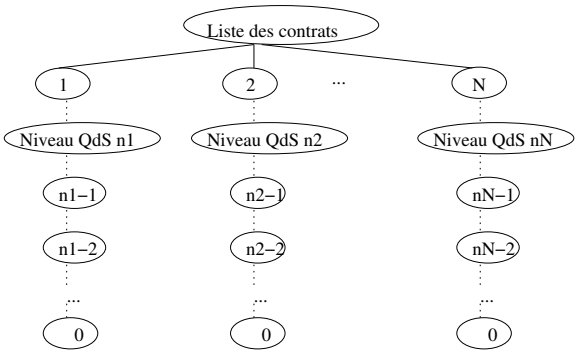

FIG. 4.18 – Représentation arborescente des n_i niveaux de QdS de N contrats.

Enfin, considérons que la valeur de la capacité d'une ressource évolue de C_{old} à C_{new}. Pour ce cas type, nous considérons que tous les contrats en cours utilisent cette ressource.

4.2.3.3 Prise en compte par Qinna

Qinna identifie la capacité d'une ressource au travers de la contrainte globale du QoSComponentBroker associée à cette ressource, tandis que les N contrats sont gérés à partir du QoSDomain. Enfin, les n_i niveaux de QdS sont donnés par les N tables de mapping. Les contraintes globales sont mises à jour directement par les QoSComponentBrokers. Cette mise à jour entraîne une renégociation globale des contrats dans les deux cas suivants :

- si la contrainte globale est supérieure à l'ancienne, c'est à dire que $C_{new} > C_{old}$. En effet, afin de respecter le principe de maximisation des niveaux de QdS contractualisés, il est nécessaire d'évaluer si des contrats peuvent être augmentés.
- si la nouvelle valeur de la contrainte globale est inférieure à l'ancienne contrainte globale et à la contrainte globale courante, c'est à dire que $C_{new} < C_{old}$ **et** $C_{new} < C_{courante}$.

Dans ces deux cas, une renégociation globale des contrats utilisant la ressource est effectuée. Dans le pire cas, chaque contrat devra passer du niveau de QdS maximal au niveau minimal, ou inversement. Le nombre d'adaptations total à effectuer est alors dans le pire cas de :

$$N_{adapt} = \sum_{i=1}^{N} n_i$$

avec n_i le nombre de niveaux de QdS du contrat i.

Afin de minimiser le nombre de renégociations globales, les QoSComponentBrokers peuvent mettre en oeuvre trois politiques différentes de mise à jour de la contrainte globale :

- la première possibilité consiste à refléter chaque changement de la capacité de la ressource au travers de la contrainte globale. Cette première stratégie, facile à mettre en oeuvre, peut générer un grand nombre de renégociations suivant le profil de la capacité de la ressource. En effet, si celle ci est fortement variable, le QoSDomain devra continuellement renégocier les contrats. Cependant, elle permet de maximiser au mieux les niveaux de QdS des contrats en utilisant au plus juste la capacité de la ressource.
- la deuxième approche est de discrétiser les valeurs possibles de la contrainte globale. La valeur discrétisée de la contrainte globale doit toujours être inférieure à la capacité de la ressource afin de ne pas provoquer de violations de contrat. La différence entre la valeur de la contrainte globale, notée cg, et la capacité, notée c est donc toujours positive : $cg - c > 0$. Cette approche permet de réduire le nombre de renégociations globales à effectuer. Le nombre de renégociations est alors égal au nombre de valeurs possibles de la contrainte globale, noté d. Plus la valeur de d est petite, c'est à dire qu'il y a peu de valeurs possibles, et plus le nombre de renégociations à effectuer sera faible. En revanche, la capacité de la ressource n'est pas utilisée à son maximum et le gaspillage de capacité est égal à : $gaspillage_{capacite} = cg - c$. Dans le pire des cas, le gaspillage de capacité est donc égal à la différence entre deux valeurs successives de la contrainte globale. Cette approche permet de limiter le nombre de renégociations, tout en acceptant qu'une partie de la capacité de la ressource ne puisse pas être utilisée. Cependant, cette politique n'est pas adaptée aux profils fortement variables de changement de capacité de ressource. En effet, si la capacité à un profil contenant de brefs pics d'augmentation, il est alors plus judicieux de ne pas modifier la contrainte globale.
- la dernière approche consiste à joindre un attribut temporel aux valeurs discrétisées afin d'augmenter la valeur de la contrainte globale que si la capacité de la ressource est stable depuis un temps t. Cette approche permet alors de diminuer le nombre de renégociations globales. En effet, la contrainte globale est mise à jour que lorsque la capacité a changé de valeur depuis un certain temps t. Cette approche est particulièrement adaptée au ressource dont la capacité varie de manière temporaire et fréquemment. Cette politique ne peut être appliquée que lorsque la capacité de la ressource augmente. En effet, si cette politique est appliquée lorsque la capacité diminue, elle peut entraîner des violations de contrats car la contrainte globale correspondra, pendant un temps t, à une capacité supérieure à la capacité réelle.

4.2.3.4 Illustration

Cette illustration présente la mise en oeuvre des deux dernière politique de mise à jour de la contrainte globale.

Considérons la ressource réseau dont la capacité est caractérisée en kilo bits par seconde (kb/s). La capacité maximale du réseau est de 40 kb/sec alors que la capacité minimale est de 0 kb/sec. L'évolution au cours du temps de la capacité du réseau est représentée à la figure 4.19.

Nous considérons dans un premier temps que le QoSComponentBroker associé au réseau identifie uniquement 5 valeurs de contraintes globales correspondants à la capacité de 0, 10, 20, 30 et 40 kb/sec. De $t = 0$ à $t = 8$ la contrainte globale est alors égale à 20 kb/sec. Durant ce temps, la somme des niveaux de QdS objectifs est donc inférieure à 20 kb/s : il y a donc gaspillage de capacité car le réseau peut supporter plus de 20 kb/sec. A $t = 8$ le gaspillage de capacité est maximal et est égal à 10 kb/sec. Pour cette illustration la contrainte globale change

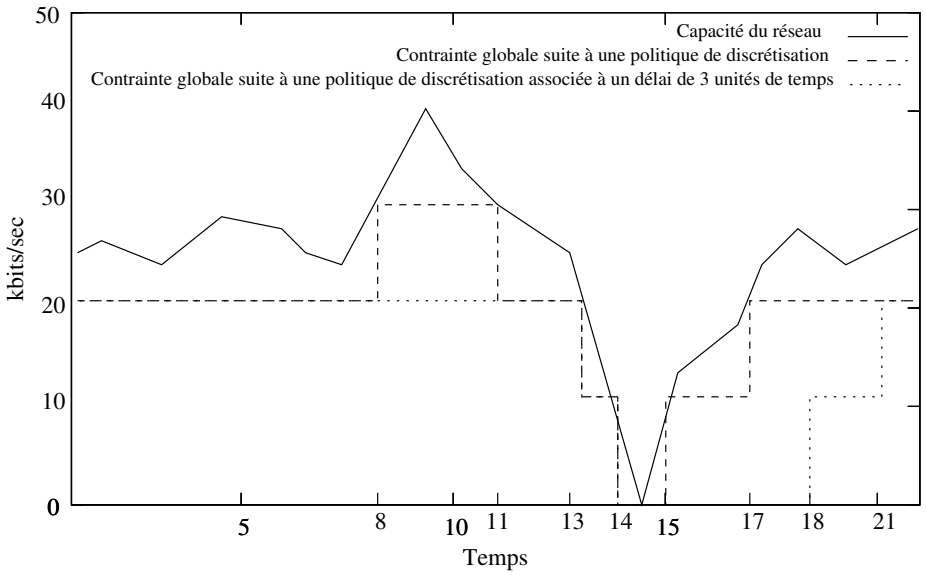

FIG. 4.19 – Évolution de la capacité du réseau en fonction du temps.

six fois de valeur : 20 kb/sec pour $t \in [0;8]$, 30 kb/sec pour $t \in]8;11]$, 20 kb/sec pour $t \in]11;13]$, 10 kb/sec pour $t \in]13;14]$, 0 kb/sec pour $t \in]14;15]$, 10 kb/sec pour $t \in]15;17]$ et enfin 20 kb/sec pour $t > 17$. A chaque changement de la valeur de la contrainte globale, une renégociation globale des contrats peut être nécessaire.

Considérons maintenant les mêmes valeurs possibles pour la contrainte globale, mais augmentées d'une propriété temporelle telle que : la contrainte globale est modifiée si la capacité de la ressource est dans le même intervalle pendant un délai égal à 3 unité de temps. Dans ce cas, la contrainte globale est égale à 20 kb/sec pour $t \in [0;13]$, car au moment où la capacité devient supérieure à 30 kb/sec ($t = 8$) le QoSReseauBroker attend 3 unités de temps avant de changer la contrainte globale. Étant donné qu'au bout de ces trois unité de temps la capacité de la ressource redevient inférieure à 30 kb/sec, la contrainte globale n'est pas modifiée. Puis, pour $t \in]11;13]$ la contrainte globale est égale à 10 kb/sec, car la politique précédente s'applique lorsque la capacité devient inférieure à la contrainte globale. Ensuite, la contrainte globale est égale à 0 kb/sec pour $t \in]13;18]$ car il est nécessaire que la capacité soit pendant 3 unités de temps aux paliers supérieurs. Puis, la contrainte globale est égale à 10 kb/sec pour $t \in]18;21]$ et à 20 kb/sec pour $t > 21$. Avec cette politique il n'y donc eu que 4 renégociations, mais en contre partie la capacité de la ressource n'a pas été utilisée à son maximum.

4.2.3.5 Analyse

La prise en compte de l'évolution de la capacité d'une ressource implique un coût à l'architecture en termes de renégociation globale des contrats. Ce coût dépend de la politique de mise à jour de la contrainte globale des QoSComponentBrokers. Trois politiques de mise à jour ont été identifiées :

- la première consiste à modifier la contrainte globale dès que la capacité est modifiée. Cette approche peut être très coûteuse mais possède les avantages d'être simple à implémenter et d'utiliser au maximum la capacité de la ressource. Elle peut être particulièrement efficace pour des ressources ayant des capacités discrétisées (par exemple dans le cas d'un processeur possédant différents paliers de vitesse en fonction du niveau d'énergie).
- la deuxième approche est celle consistant à modifier la valeur de la contrainte globale par palier. Cette

approche revient donc à discrétiser les valeurs d'une capacité. Elle permet de limiter le nombre de renégociations, mais sous estime la capacité réelle de la ressource. Cette approche est adaptée aux ressources ayant une capacité variant lentement (par exemple la batterie). Cette approche pose les problèmes de choix d'un pas tel qu'il est identifié à la section 4.2.2 page 81 portant sur l'adaptation à un profil variable de QdS.

– enfin, la troisième approche consiste à adjoindre une propriété temporelle aux valeurs précédemment discrétisées. Elle permet de limiter les renégociations globales, mais est plus difficile à implémenter. Elle est particulièrement adaptée aux ressources variant fréquemment ou temporairement de capacité (par exemple la bande passante d'un réseau). Lorsque la capacité de la ressource augmente sous forme de pics, cette politique est préconisée afin d'éviter les renégociations globales des contrats, mais lorsque la capacité diminue la politique précédente doit être appliquée afin de se prémunir des violations de contrats.

4.2.4 Évolution de la relation d'ordre

4.2.4.1 Problématique

Le dernier cas portant sur le comportement dynamique de l'architecture se focalise sur la relation d'ordre des niveaux d'importance. En effet, celle-ci peut ne pas être constante tout au long du cycle de vie du système. Elle peut-être modifiée soit explicitement par l'utilisateur, soit être dépendante d'un ou plusieurs paramètres liés au cycle de vie du système. Par exemple, la relation d'ordre peut dépendre du lieu d'utilisation du système : les contrats liés aux composants d'applications bureautiques doivent être plus importants que ceux liés aux jeux quand le système est utilisé dans l'enceinte de l'entreprise, et inversement lorsque le système est utilisé au domicile.

La modification de la relation d'ordre entraîne alors une renégociation globale des contrats en cours d'exécution.

4.2.4.2 Présentation

Soit le cas générique comprenant N contrats en cours d'exécution et possédant chacun n_i niveaux de QdS distincts (cf. figure 4.18 page 86). Chaque contrat possède un niveau d'importance, noté i, tel que i soit un nombre entier et que $i+1$ soit plus important que i. Nous considérons que chaque contrat a été établi initialement au niveau de QdS maximal, mais que les contrats ont pu être dégradés au profit de contrats plus important. De plus, d'après le comportement de Qinna lors de l'établissement d'un contrat, l'utilisateur est assuré que le contrat de niveau d'importance maximal est établi au niveau de QdS maximal.

Dans ce cadre, considérons que la relation d'ordre des niveaux d'importance est inversée de telle sorte que i devienne plus important que $i+1$.

4.2.4.3 Prise en compte par Qinna

Dans Qinna, la modification de la relation d'ordre des niveaux d'importance peut-être effectuée soit explicitement par l'administrateur du système (service *f_Order_Relation* de l'interface *iQoSDomainAdministration*), soit parce que la relation est fonction d'un ou plusieurs paramètres (par exemple le temps ou la localisation du système).

Une modification de la relation d'ordre nécessite une renégociation globale des contrats en cours d'exécution afin de prendre en compte les nouveaux niveaux d'importance. Ces contrats sont identifiés par le QoSDomain. La renégociation correspond à parcourir l'arbre de la figure 4.18 en commençant par le contrat devenu le plus important. D'après le comportement dynamique de Qinna lors de l'établissement d'un contrat, on sait que le contrat devenu le plus important peut être amélioré à son niveau de QdS maximal, en annulant au pire tous les autres contrats.

La renégociation des contrats en cours d'exécution s'effectue comme suit (cf. figure 4.20) : le contrat devenant le plus important est identifié (opérateur *compare(T_IMP i_1,T_IMP i_2)*) puis est évalué afin de déterminer si son niveau de QdS est égal au niveau de QdS spécifié initialement par l'utilisateur (correspondant au niveau de QdS maximal). Si ce n'est pas le cas, cela signifie que le contrat a été précédemment dégradé. Les contrats de niveau d'importance inférieur sont alors dégradés (service *degrade* de l'interface *iQoSManager*) jusqu'à ce que le contrat le plus important retrouve un niveau de QdS maximal, qui correspond alors au niveau initial lorsqu'il a été établi. Puis, ces opérations sont réitérés sur les contrats suivants d'importance inférieur. Il est alors possible que les contrats de niveaux inférieurs doivent être annulés.

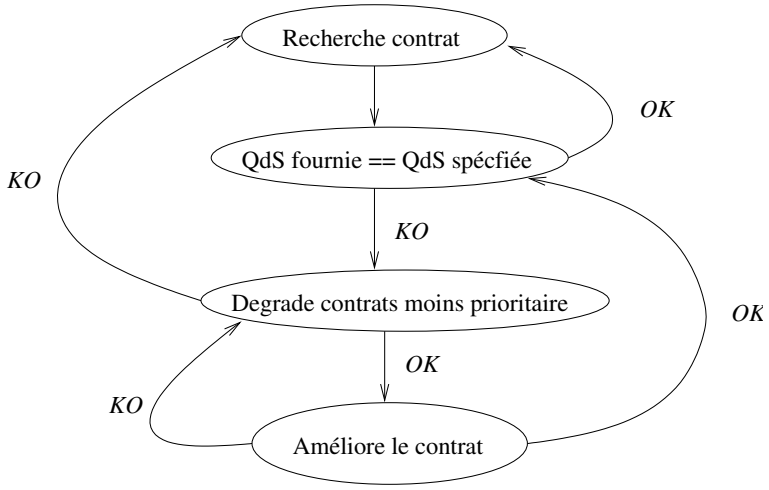

FIG. 4.20 – Renégociation des contrats lors d'une modification de relation d'ordre.

Dans le pire cas, lors d'une modification de la relation d'ordre, le nombre d'adaptations de contrats à effectuer est égal à :

$$nb\_adapt\_max = \sum_{i=1}^{N} n_i$$

avec N le nombre de contrats en cours d'exécution et n_i le nombre de niveaux de QdS du contrat i.

En effet, dans le pire cas, le contrat devenu le plus important a été dégradé au niveau de QdS minimal et les autres contrats sont au niveau de QdS maximal : le contrat le plus important doit donc être amélioré de niveau de QdS minimal au niveau maximal et tous les autres contrats doivent être annulés.

Par contre, dans le cas le plus favorable, aucune adaptation n'a été effectuée avant la modification de la relation d'ordre : cela signifie que tous les contrats sont à leur niveau maximal. Il n'y a donc pas d'adaptations à réaliser :

$$nb\_adapt\_min = 0$$

4.2.4.4 Illustration

Nous présentons pour ce cas type deux illustrations. La première illustre le cas où la relation d'ordre évolue dynamiquement en fonction du temps. La deuxième illustre une politique de mise à jour dynamique des niveaux d'importance des contrats, ce qui revient à une modification de la relation d'ordre.

4.2.4.4.1 Evolution de la relation d'ordre Soit deux composants A et B utilisant chacun un composant mémoire via les composants Ma et Mb. Les composants A et B fournissent deux niveaux de QdS BON et MAUVAIS nécessitant respectivement 7 et 3 ko de mémoire. La capacité totale de la mémoire est fixée à 10 ko. De plus il existe deux niveaux d'importance *TRAVAIL* et *REPOS*. La relation d'ordre définie est telle que entre 8 heures et 18 heures $Imp(TRAVAIL) > Imp(REPOS)$ et qu'entre 18 heures et 8 heures $Imp(REPOS) > Imp(TRAVAIL)$.

Une fois l'architecture Qinna intégrée, considérons le scénario où l'utilisateur active les composants QoSA et QoSB de niveaux BON et d'importance *TRAVAIL* pour QoSA et *REPOS* pour QoSB. Cela signifie qu'entre 8h et 18h, QoSA, respectivement QoSB, est exécuté au niveau BON, respectivement MAUVAIS. Passé 18h, les niveaux de QdS sont inversés par le QoSDomain : le nouveau contrat le plus prioritaire est identifié (QoSB), puis étant donné que QoSB n'est pas au niveau de QdS spécifié par l'utilisateur (BON), le contrat moins important est dégradé, soit celui de QoSA. Le contrat lié à QoSA est donc dégradé à MAUVAIS et le contrat de QoSB peut alors passer à BON. Enfin, le contrat de niveau inférieur à QoSB est identifié (QoSA) afin d'effectuer une demande d'amélioration. Celle-ci échoue et donc les niveaux de contrats de QoSA et QoSB ont été inversé.

4.2.4.4.2 Evolution dynamique des niveaux d'importance La deuxième illustration présente la mise en oeuvre d'une politique de mise à jour dynamique des niveaux d'importance des contrats. Cette politique s'appuie sur les travaux de Delacroix [26] portant sur la définition d'un régisseur d'ordonnancement de tâches temps-réel. Un régisseur permet d'adapter dynamiquement les configurations des tâches d'un système temps-réel afin de répartir équitablement les violations des échéances lorsque le système est en surcharge. Nous proposons d'implémenter via Qinna les principes de ces travaux afin de permettre une répartition équitable des adaptations à réaliser sur les contrats.

Le régisseur temps réel introduit la notion d'importance pour les tâches et s'appuie sur trois principes. Le premier est de supprimer les tâches les moins importantes ainsi que toutes les tâches associées par une relation de précédence lorsque le système est en surcharge. Le deuxième est de garantir un nombre de suppressions maximal pour les tâches d'une importance donnée. Pour cela, une grammaire a été définie permettant de spécifier qu'une tâche ne peut pas être supprimée, par exemple, plus de 2 fois sur 5 tentatives d'exécution. Enfin, le dernier principe consiste à définir des politiques cohérentes afin d'augmenter le niveau d'importance d'une tâche lorsque celle-ci est supprimée et à le diminuer lorsqu'elle est exécutée afin de respecter les contraintes énoncées précédemment.

Transposer dans Qinna, l'idée principale du régisseur est d'augmenter le niveau d'importance d'un contrat lorsque celui-ci est dégradé et de le diminuer lorsqu'il est amélioré. Prenons un ensemble de N contrats de niveau d'importance, noté i, et dont la relation de d'importance est telle que :$Imp(i+1) > Imp(i)$ et $Imp(i) > Imp(i-1)$. Lorsque le QoSDomain recherche un contrat de niveau d'importance i à dégrader, un contrat parmi les N est choisi puis est dégradé. La mise en place du régisseur revient alors à augmenter le niveau d'importance du contrat, en passant par exemple à $i+1$. Si une seconde dégradation de contrat de niveau i est nécessaire le même contrat ne peut pas être choisi car il est de niveau d'importance supérieur. De même, lorsque le QoSDomain

améliore un contrat de niveau d'importance i, son niveau d'importance doit diminuer, en passant par exemple à $i-1$. De ce fait, tous les contrats de niveau i seront améliorés un par un et niveau de QdS par niveau de QdS. Dans le pire cas, la mise en place du régisseur permet de garantir que deux contrats de même niveau d'importance auront au plus un niveau de QdS d'écart.

De plus, il est à noter que la mise en oeuvre du régisseur par Qinna permet de respecter le principe de suppression des tâches associées. En effet, lors de l'adaptation d'un contrat par le QoSDomain, tous les sous contrats associés sont aussi adaptés.

Enfin, la mise en oeuvre de cette politique ne doit pas interférer avec les relations d'ordre sur les niveaux d'importance spécifiés par l'utilisateur comme constants. En effet, il ne faut pas qu'un contrat de niveau d'importance minimal devienne un contrat de niveau d'importance maximal. Il faut donc sous diviser les niveaux d'importance utilisateurs en *classes d'importance* afin de gérer de manière équitable les adaptations des contrats d'une même classe.

4.2.4.5 Analyse

Le coût de la renégociation pour ce cas type est compris entre O et $\sum_{i=1}^{N} n_i$ adaptions de contrats. Afin de minimiser ce coût, les dégradations de contrats effectuées avant la modification de la relation d'ordre peuvent être mémorisées afin de d'améliorer directement le contrat le plus important au niveau maximal plutôt que de l'effectuer niveau par niveau.

De plus, il est à noter que la réévaluation de l'ensemble des contrats peut aboutir à une annulation de contrats moins importants. Par exemple, si le niveau de MAUVAIS de QoSA requiert 4 ko de mémoire au lieu de 2 ko, le contrat associé à QoSA sera annulé afin de laisser assez de mémoire pour QoSB. Dans le pire des cas, Qinna annule tous les contrats sauf le contrat le plus important : celui-ci est en effet certain de retrouver son niveau de QdS initial.

Enfin, comme dans le cas de l'évolution de la capacité d'une ressource, le paramètre dont dépend, éventuellement, la politique de mise à jour de la relation d'ordre doit être discrétisé. De plus, ce paramètre doit être suffisamment stabilisé avant d'effectuer la renégociation des contrats. Par exemple, dans le cas où le paramètre est un lieu, la modification de la relation d'ordre doit intervenir lorsque le lieu est identifié depuis un certain délai, afin de ne pas déclencher un enchaînement de renégociations.

4.3 Evaluations quantitatives

Cette partie propose une évaluation quantitative de l'architecture Qinna en termes d'espace mémoire et de temps CPU. Pour cela, nous évaluons deux expérimentations complètes de systèmes intégrant l'architecture Qinna parmi l'ensemble des expérimentations réalisées durant cette thèse. La première expérimentation permet d'évaluer les coûts liés aux différents composants de l'architecture, ainsi que les coût d'établissement, d'annulation et d'adaptation de contrats de QdS. La deuxième expérimentation se focalise sur les coûts liés à l'observation et la maintenance des contrats de QdS.

Les deux expérimentations présentées ont été réalisées à partir du framework à composants Think pour la plateforme matérielle iPaq™ H3800 [22] comprenant un microprocesseur StrongARM™ SA1100 [45] à 206 MHz, 60 Mo de mémoire DRAM et 20 Mo de mémoire flash.

4.3.1 Évaluations de la mise en place et de l'adaptation de contrats

4.3.1.1 Présentation

La première expérimentation permet d'évaluer le coût mémoire et CPU des différents composants induits par l'architecture, ainsi que le coût lié aux opérations de gestion de contrats de QdS.

Nous présentons maintenant le système intégrant l'architecture Qinna (cf. figure 4.21). Celui-ci est composé d'un composant (UI) permettant de piloter deux composants applicatifs (QoSDoom et QoSVideo) : le premier exécute le jeu Doom [1] tandis que le second exécute un clip vidéo. Les deux composants applicatifs requièrent chacun un thread (QoSThread1 et QoSThread2) et de la mémoire (QoSMem1 et QoSMem2) pour s'exécuter. L'ordonnanceur (QoSOrdo) ordonnance les threads suivant leur pourcentage requis d'accès au processeur. Pour cette expérimentation l'ordonnanceur permet d'utiliser 90% du processeur.

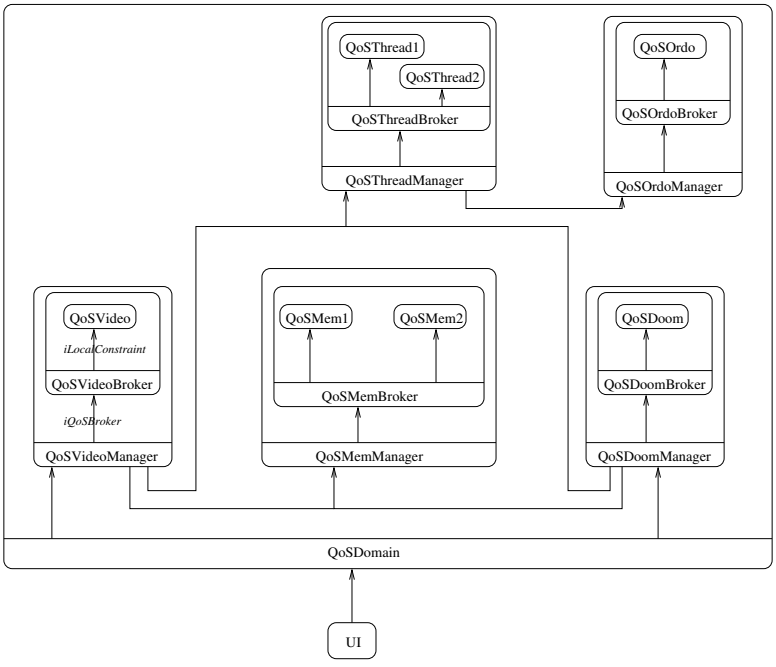

FIG. 4.21 – Système multimédia intégrant l'architecture Qinna et exécutant le jeu Doom et une vidéo.

Le composant QoSDoom possède trois niveaux de QdS : BON, MOYEN et MAUVAIS, tandis que le composant QoSVidéo en possède deux : BON et MAUVAIS. Les tableaux 4.2 et 4.3 représentent les quantités de mémoire, ainsi que les pourcentages d'accès au processeur des threads, requis par les composants pour chacun de leurs niveaux de QdS. De plus, deux niveaux d'importance sont définis, 1 et 2, tel que $Imp(1) > Imp(2)$.

QoSVideo		
Niveaux QdS	Pond. thread	Mem
BON	74	74
MAUVAIS	20	61

TAB. 4.2 – Table de mapping de QoSVideo.

QoSDoom		
Niveaux de QdS	Pond. thread	Mem
BON	75	892
MOYEN	50	892
MAUVAIS	20	663

TAB. 4.3 – Table de mapping de QoSDoom.

4.3.1.2 Mesures

4.3.1.2.1 Coût mémoire Les mesures des composants mémoires données dans cette partie correspondent à la taille des composants une fois compilés.

La taille totale du système intégrant Qinna et représenté à la figure 4.21 est de 749 ko. La taille des composants propres à Qinna est de 11,5 ko, tandis que la taille des composants fonctionnels est de 565 ko (cf. tableau 4.4). Par rapport à la taille totale du système, les composants propres à l'architecture représentent donc 1,5%.

Nous n'avons considéré ici que le coût lié à la mémoire statique. En effet, afin d'effectuer une analyse plus fine il serait nécessaire d'évaluer la mémoire requise dynamiquement par l'architecture : cette mémoire correspond à la taille des contrats mis en place par chaque QoSComponentManager et par le QoSDomain. D'après la définition d'un contrat (cf. page 42), la taille d'un contrat du QoSComponentManager reste faible et est au minimum de 6 octets.

Composants	Taille mémoire (octets)
QoSOrdo	1304
QoSThread1	223
QoSThread2	223
QoSMem1	152
QoSMem2	152
QoSVideo	77179
QoSDoom	498476
UI	1680
QoSOrdoBroker	1248
QoSOrdoManager	987
QoSThreadBroker	1043
QoSThreadManager	1123
QoSMemBroker	978
QoSMemManager	567
QoSVideoBroker	440
QoSVideoManager	1400
QoSDoomBroker	253
QoSDoomManager	1528
QoSDomain	2178

TAB. 4.4 – Taille mémoire des composants du système intégrant Qinna.

Composants	Taille mémoire (octets)
Ordo	1278
Thread1	223
Thread2	223
Mem1	0
Mem2	0
Video	76987
Doom	498398
UI	1501

TAB. 4.5 – Taille mémoire des composants du système n'intégrant pas Qinna.

4.3.1.2.2 Coût CPU Afin d'évaluer les coûts CPU liés à la gestion des contrats de QdS, considérons le scénario suivant : l'utilisateur requiert tout d'abord le composant QoSVideo avec un niveau de QdS égal à BON et un niveau

d'importance égal à 1.

Le temps d'établissement du contrat de QoSVideo, du point de vue de l'utilisateur, est de 4,08 msec. Ce temps est décomposé en temps d'établissement des sous contrats : par exemple le temps d'établissement du contrat entre QoSThreadManager et QoSOrdoManager est de 0,59 msec. Eux-mêmes sont décomposés en temps utilisé par le QoSComponentBroker, pour réaliser le test d'admission, et par le QoSComponentManager : le temps utilisé par le QoSOrdoBroker pour réaliser le test d'admission de la politique d'ordonnancement est de 0,12 msec.

Prenons maintenant le cas où l'utilisateur requiert le composant QoSDoom de niveau BON et d'importance 1. Étant donné que les deux composants ne peuvent être au niveau BON simultanément, dû à la contrainte sur la somme des pondérations des threads (74%+75%>100%), le composant QoSVideo est dégradé au niveau MAUVAIS.

Les mesures suivantes ont été effectuées :
- Temps d'établissement du contrat de QoSDoom : 4,36 ms. Ce temps intègre le temps lié à la dégradation de QoSVideo.
- Temps de dégradation du contrat QoSVideo : 1,08 ms.

Enfin, nous avons évalué le cas où l'utilisateur demande la désactivation du composant QoSDoom. Le contrat lié à QoSDoom est alors annulé et le contrat de QoSVideo peut être amélioré à BON.

Les mesures suivantes ont été effectuées :
- Temps de désactivation de QoSDoom : 2,26 ms. Ce temps comprend l'amélioration de contrat de QoSVideo.
- Temps d'amélioration du contrat de QoSVideo : 1,02 ms.

4.3.2 Évaluations quantitatives de la maintenabilité d'un contrat

4.3.2.1 Présentation

La deuxième expérimentation se focalise sur les aspects d'observation et de maintenance de contrat de QdS. Pour cela, nous considérons un composant pouvant exécuter une vidéo (QoSVideo) et piloter via une interface utilisateur (UI). Le composant QoSVideo requiert un thread et de la mémoire, et le thread est ordonnancé par un ordonnanceur appliquant une politique d'ordonnancement round-robin à pondération. Le composant QoSVideo possède deux niveaux de QdS, BON et MAUVAIS, mais ne connaît pas le niveau de QdS requis sur le composant QoSThread : le paramètre de la table de mapping correspondant à QoSThread est alors fixé à *default*. Enfin, l'architecture met en place le composant QoSVideoObserver (cf. figure 4.22).

4.3.2.2 Mesures

4.3.2.2.1 Coût mémoire La taille totale du système est de 112 ko, alors que la taille des composants propres à l'architecture Qinna est de 11,2 ko : dans cette expérimentation le surcoût de Qinna est donc de 10,1% (cf. tableau 4.6).

Il est à noter que par rapport à l'expérimentation précédente la taille des composants QoSDomain et QoSVideo-

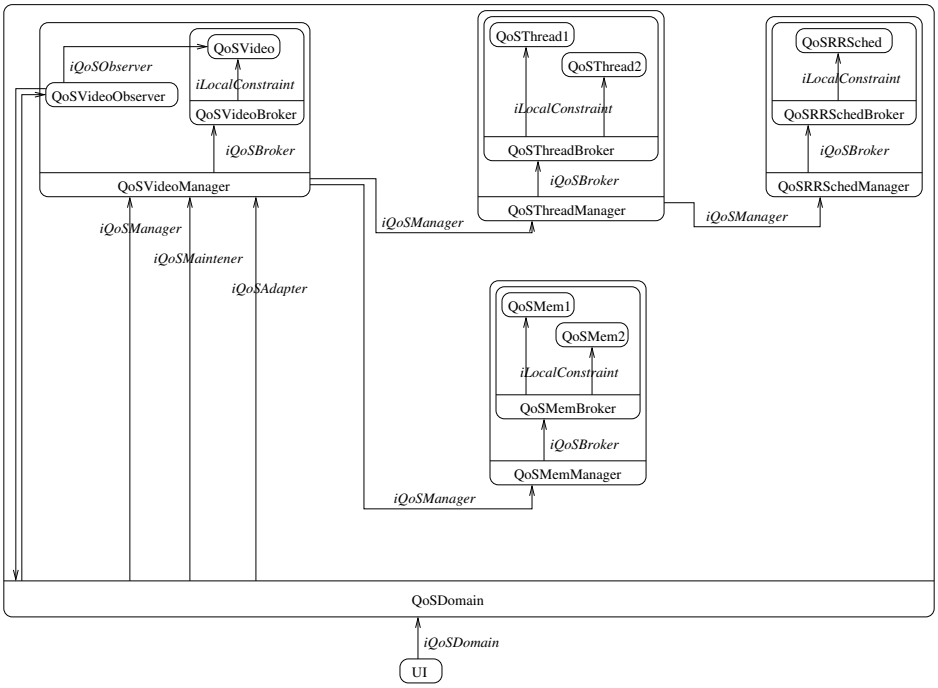

FIG. 4.22 – Système multimédia intégrant l'architecture Qinna et exécutant une vidéo.

Manager a augmenté : en effet dans cette expérimentation, ces derniers doivent, en plus, implémenter la politique et les mécanismes de maintenabilité.

4.3.2.2.2 Coût CPU Nous fixons la politique d'observation du QoSVideoObserver à la période de 100 ms. Dans la table de mapping de QoSVideoManager, le paramètre *default* du thread est fixé à 15%. De plus, le pas de maintenabilité de ce paramètre est fixé aussi à 15%. Comme pour l'expérimentation précédente la pondération réellement requise pour le niveau BON est égale à 75%.

Une fois que l'utilisateur a effectué la demande d'activation du composant QoSVideo, le délai d'établissement du niveau de QdS à BON est évalué à 402 msec. En effet, le nombre de violations de contrat notifiés par le QoSVideoObserver au QoSDomain est égal à 4. Plus finement, du point de vue du QoSDomain, l'appel aux mécanismes de maintenabilité de QoSVideoManager dure 1,9 msec.

4.3.3 Analyse

Cette partie portant sur l'évaluation quantitative de Qinna permet de montrer que le coût de l'architecture n'est pas aussi élevé que ce que laisse présager la lecture des chapitres 3 et 4 sur la définition et l'évaluation de Qinna. La principale raison de ce constat est dû au fait que les applications s'exécutant sur les systèmes visés (systèmes embarqués mobiles) sont, en général, simples et peu nombreuses. En effet, on peut constater que chaque application ne possède que peu de niveaux de QdS différents et que la granularité de leurs composants est grosse (peu de composants). De plus, les implémentations de Qinna tirent profit des avantages du framework Think permettant de ne compiler et d'assembler, et donc de ne charger, que les composants réellement utilisés.

Composant	Taille mémoire (octets)
QoSOrdo	1304
QoSThread1	223
QoSMem1	152
QoSVideo	77179
UI	1680
QoSOrdoBroker	1248
QoSOrdoManager	987
QoSThreadBroker	1043
QoSThreadManager	1123
QoSMemBroker	978
QoSMemManager	567
QoSVideoBroker	440
QoSVideoManager	1826
QoSVideoObserver	478
QoSDomain	2863

TAB. 4.6 – Taille mémoire des composants.

Enfin, le coût de gestion de la QdS des composants ne peut être nul et doit être pris en compte lors du dimensionnement du système. Cependant, ceci n'empêche pas le besoin d'optimisation des implémentations de l'architecture Qinna. Plusieurs optimisations peuvent être envisagées : la première est de regrouper toutes les sous contrats d'un contrat au sein du QoSDomain afin de limiter les appels entre composants. La deuxième peut être de mémoriser les opérations d'adaptations réalisées par le QoSDomain afin que, lorsqu'une configuration du système a déjà été rencontrée, les bons niveaux de QdS puissent être directement établis. Une troisième optimisation possible est de paramétrer les demandes d'adaptations afin de pouvoir, par exemple, dégrader un contrat du niveau *bon* au niveau *mauvais* sans avoir à passer par le niveau intermédiaire *moyen*.

4.4 Conclusion

Ce chapitre a permis d'évaluer aussi bien qualitativement que quantitativement les apports et les limites de l'architecture Qinna au travers d'illustrations et d'expérimentations sur un ensemble de cas types. Nous pouvons ainsi en déduire les conclusions suivantes :

– l'établissement des contrats se base sur les valeurs fournies par les tables de mapping. Celles-ci possèdent donc un rôle central au sein de l'architecture Qinna. Ces valeurs pouvant être difficiles à établir, l'architecture prévoit des mécanismes permettant de les déterminer dynamiquement grâce à la mise en oeuvre des politiques et mécanismes de maintenabilité. La maintenabilité est activée pour des valeurs de niveaux de QdS renseignées à *default* (pour des valeurs inconnues a priori) et des attributs de fiabilité des tables de mapping fixés à *unreliable* (pour des valeurs imprécises). De plus, les tables de mapping peuvent être complexes. Par exemple, plusieurs combinaisons de niveaux de QdS requis peuvent correspondre à un même niveau de QdS fourni. De même, les valeurs des tables de mapping peuvent être paramétrées. Enfin, il est à noter que la granularité des composants du système impacte la mise en place des tables de mapping. En effet, plus la granularité des composants est fine et plus les niveaux de QdS les reliant sont difficiles à déterminer.
– l'intégration de Qinna passe par une structuration particulière des composants du système. L'accès à un composant partagé insécable impose une conception à méta niveau en introduisant un composant de contrôle

d'accès (classiquement appelé ordonnanceur). De plus, à ce composant d'accès est associé un composant réalisant le test d'admission (QoSComponentBroker) distinct du composant réalisant l'accès proprement dit. Cette structuration du système permet une intégration homogène de Qinna pour l'intégralité du système quelque soit le niveau considéré (niveau ressources, système d'exploitation, services ou applicatif). De même, dans le cas du partage de composant sécable, la division en sous composants nécessite une adaptation des composants existants, mais permet une gestion de la QdS fine et sûre. Par exemple, dans le cas du partage de la ressource mémoire, l'utilisation de Qinna aboutit à un partionnement de la mémoire par composant. Chaque composant possède alors son propre composant mémoire permettant de le contrôler individuellement.

- la prise en compte de l'hétérogénéité de la QdS par l'architecture se traduit par une interprétation sémantique des niveaux de QdS via l'opérateur *translate*. Cet opérateur peut entraîner, dans certains cas, à une perte d'information sur les niveaux de QdS. De plus, il peut conduire soit à un gaspillage de QdS, lorsque l'interprétation sur-estime les niveaux de QdS, soit à la réalisation d'opérations de mises à jour des données des contrats, lorsque l'interprétation sous-estime les niveaux de QdS requis.
- la vision décentralisée des contrats de QdS entre les différents composants du système implique la collaboration de l'ensemble de ces composants lors de la gestion de la QdS. L'ensemble des composants gérant les contrats de QdS forment alors un arbre qu'il convient de parcourir, soit exhaustivement, soit partiellement, lorsqu'une opération de gestion de la QdS est nécessaire. De plus, étant donné que chaque composant est lié à, au moins, un contrat, la granularité des composants du système impacte le nombre de contrats à établir et donc le coût de parcours de l'arbre.
- la prise en compte d'un profil variable de QdS peut s'effectuer soit de façon statique, si le profil est connu à priori, soit de façon dynamique via la mise en oeuvre d'observation et des politiques de maintenabilité. Deux stratégies ont été identifiées pour une prise en compte dynamique : l'une, dite *à scrutations*, permet de suivre l'augmentation et la diminution des niveaux de QdS requis, mais induit des retards, tandis que l'autre, dite *événementielle*, permet de suivre, sans retard, uniquement l'augmentation des niveaux de QdS requis. La première stratégie convient à un profil variable, tandis que la seconde permet de déterminer un niveau de QdS requis stable à partir d'une valeur minimale. Les coûts de cette prise en compte s'évaluent alors en retard et/ou en gaspillage de QdS et dépendent des paramètres d'observation (période d'observation) et des paramètres de maintenabilité (pas de maintenabilité).
- la prise en compte de l'évolution de la capacité d'une ressource doit être dépendante du type de ressource concernée. Trois types de ressources ont alors été identifiées : les ressources ayant des capacités discrétisées (par exemple les processeur à paliers de vitesse en fonction du niveau d'énergie), les ressources dont la capacité varie lentement (par exemple la batterie) et les ressources dont la capacité subit des variations temporaires (par exemple la bande passante d'un réseau). Afin de réagir à l'évolution des capacités de ces trois types de ressources, trois stratégies ont été identifiées. Dans le premier cas (ressources à valeurs discrétisées), il convient de modifier la contrainte globale à chaque changement de la capacité de la ressource. Dans le deuxième cas (variation lente de la capacité), les valeurs de la contrainte globale sont discrétisées afin de ne déclencher les opérations d'adaptation que lorsque la capacité a atteint un certain seuil. Enfin, dans le dernier cas (variations temporaires de la capacité), il est nécessaire d'adjoindre des propriétés temporelles aux valeurs discrétisées de la contrainte globale afin de ne déclencher les opérations d'adaptations que lorsque la variation dépasse un certain seuil depuis un certain temps.
- enfin, le coût de l'architecture à l'implémentation n'est pas nul et doit être pris en compte lors du dimensionnement du système. Ce coût dépend alors de la granularité des composants, du nombre de niveaux de QdS de chaque composant, mais aussi des politiques d'observations et des pas de maintenabilité.

Ces conclusions permettent de préciser les réponses apportées par Qinna aux objectifs initialement fixés (cf. section 1 page 2) :

1. **la généricité :** l'objectif visé est de proposer un cadre générique permettant l'intégration de diverses politiques de gestion de QdS. L'architecture Qinna a alors été définie à l'aide de types de composants sans présupposer d'une politique particulière. La validation de cet objectif est impossible, car elle nécessiterait de tester de manière exhaustive toutes les politiques de gestion de QdS. Cependant, à la vue des résultats de ce chapitre nous pouvons identifier les domaines d'applications privilégiés de l'architecture. Premièrement, Qinna est particulièrement adaptée aux systèmes constitués d'applications pouvant être configurées ou ayant plusieurs modes de fonctionnement identifiant des dégradations des services. Il s'agit naturellement des applications multimédia travaillant sur des flots de données telles que les lecteurs vidéo, de musique ou les jeux. Qinna apporte alors tous les aspects liés à la gestion dynamique de la QdS : adaptation des niveaux de QdS, suivis des niveaux variables de QdS requis, prise en compte de l'évolution des capacités des ressources et des relations d'ordre des niveaux d'importance. D'un autre coté, les applications ne possédant qu'un mode de fonctionnement ne permettent pas de tirer profit ces aspects de l'architecture. Pour autant, l'étude de ce chapitre a permis de montrer que l'architecture peut aussi cibler ces systèmes. Par exemple, dans le cas des systèmes temps-réel, Qinna permet de structurer clairement les différents éléments de QdS en identifiant, en particulier, les éléments à mettre en oeuvre pour la gestion de l'hétérogénéité (par exemple, lors du passage du temps-réel relâché au temps-réel dur, et inversement ; ou lors de la cohabitation d'ordonnanceurs temp-réel et round-robin ; ou lors de la cohabition de tâches périodiques et apériodiques) et pour la gestion des modes dégradés des applications temps-réel (par exemple, lors de la mise en oeuvre du régisseur temps-réel). Parallèlement, la définition des types de composants impliquent la mise en oeuvre d'une politique de gestion de QdS par classe de QoSComponent : il n'est donc pas possible que deux QoSComponents d'une même classe soient gérés de manière différente. Ceci apparaît comme une limite de l'approche.
2. **la dynamicité :** l'objectif visé est de pouvoir gérer dynamiquement la QdS des composants. Les principes sur lesquels se repose Qinna pour répondre à cet objectif, sont les contrats de QdS inter-composants et leurs activités de gestion (spécification, initialisation et gestion). En poussant au maximum la philosophie des contrats et des composants, Qinna montre que la prise en compte de la dynamicité est possible, mais qu'elle possède un coût. Ce coût est fonction de nombreux paramètres tels que le nombre de niveaux de QdS des composants ou, surtout, la granularité des composants, et s'évalue en termes de nombre d'opérations d'adaptations à réaliser et de gaspillage de QdS. Les résultats de ce chapitre montre que les concepts présents dans l'architecture sont suffisants pour une gestion dynamique de la QdS, mais que leur mise en oeuvre revient à effectuer un compromis entre le gaspillage de QdS et le nombre d'opérations à réaliser. En particulier, dans le cas des composants possédant des profils variables de QdS requis, ou dans le cas des ressources à capacité variable, plus le gaspillage de QdS est important et moins le nombre d'opérations à réaliser est grand. En effet, dans ce cas, le niveau de QdS contractualisé est toujours inférieur au niveau de QdS réellement consommé. A l'inverse, la minimisation du gaspillage de QdS implique un grand nombre d'opérations afin d'adapter dynamiquement les niveaux de QdS requis contractualisés.
3. **l'hétérogénéité :** l'objectif visé ici porte sur l'intégration de composants possédant des QdS hétérogènes. Pour cela Qinna définit l'opérateur *translate* permettant de traduire des QdS hétérogènes. Ce chapitre a permis de montrer que, ici aussi, la prise en compte de l'hétérogénéité possède un coût aussi en bien termes d'opérations à réaliser que de gaspillage de QdS. De plus, les expérimentations de ce chapitre montrent que pour limiter ces coûts le concepteur doit avoir prévu les différentes QdS pouvant être rencontrées et doit définir des règles de traduction d'une QdS à une autre. Dans le cadre des systèmes ouverts, la définition de ces règles peuvent être difficiles à réaliser et l'hétérogénéité est prise en compte dans la limite prévue par la concepteur. Il est à noter que lorsque l'hétérogénéité n'est pas prévue, la valeur de la QdS traduite est fixée

à *default* entraînant ainsi les coûts liés à l'auto-configurabilité.

Cette forme d'hétérogénéité est la seule prise en compte explicitement par Qinna. Cependant d'autres formes peuvent être prises en compte grâce aux propriétés du framework à composants utilisé lors de l'implémentation. Par exemple, dans le cadre des expérimentations réalisées, l'utilisation du framework Think permet une hétérogénéité des langages de programmation étant donné que les composants peuvent être programmés dans divers langages (assembleur, C, Java). De même, l'utilisation du framework Julia (il s'agit du framework du modèle Fractal pour des composants du monde Java) permet l'hétérogénéité des systèmes d'exploitation grâce à l'utilisation d'une machine virtuelle. Cependant, dans ce dernier cas, Qinna se situe au-dessus de la machine virtuelle et ne peut donc disposer d'un contrôle total des ressources de la plateforme. Une solution serait alors d'intégrer Qinna à la machine virtuelle et au système d'exploitation.

4. **l'auto-configurabilité :** l'objectif visé est de permettre à un composant de déterminer dynamiquement ses niveaux de QdS requis. A cet effet, l'architecture permet de mettre en place des observateurs ainsi que des politiques de maintenabilité permettant de faire évoluer les niveaux de QdS requis d'un composant. Les résultats de ce chapitre montrent que cet objectif est coûteux à prendre en compte. Ce coût est lié aux paramètres des politiques de maintenabilité et d'observation, et se traduit par un nombre d'opérations à réaliser, un gaspillage de QdS et des retards. De même que pour la dynamicité, la prise en compte de cet objectif amène à effectuer un compromis entre le gaspillage de QdS, le nombre d'opérations à réaliser et les retards induits. En effet, lorsque l'on souhaite minimiser les retards, le nombre d'opérations à réaliser augmente, et inversement. De même, lorsque l'on souhaite minimiser le nombre d'opérations, le gaspillage de QdS est augmenté, et inversement. Il existe donc un impact de la mise en place des politiques de QdS sur les résultats de ces politiques (par exemple, création de retards).
5. **la réutilisabilité :** la séparation des activités de gestion de QdS entre les différents types de composants de Qinna permet de disposer d'une architecture réutilisable. En effet, lors des expérimentations présentées dans ce chapitre, les politiques d'adaptations implémentées par le QoSDomain sont identiques : cela montre que les QoSComponentBrokers et QoSComponentManagers ne sont pas liés au QoSDomain. A l'inverse, la dernière illustration, portant sur la modification de la relation d'ordre des niveaux d'importance, montre que la modification du QoSDomain n'impacte pas les implémentations des autres composants. De plus, les QoSComponentObservers réalisent uniquement l'activité d'observation permettant ainsi de facilement l'ajouter ou la supprimer étant donné que cette activité est optionnelle. Enfin, la prise en charge du test d'admission et de la réservation par le QoSComponentBroker permet de réutiliser ce composant pour différentes classes de QoSComponents. Par exemple, dans le cas des composants de services ou applicatifs les tests d'admissions sont souvent les mêmes (par exemple, vérification du nombre d'instances en cours d'exécution) et seule la contrainte globale est modifiée. A l'inverse, les tests d'admission liés aux ressources sont très différents et peuvent être complexes dans le cas, par exemple, de tests d'admissions pour les ressources insécables (CPU, réseau). Afin d'augmenter les possibilités de réutilisation apportées par Qinna, la séparation des préoccupations au sein des QoSComponents devrait être encore plus forte en séparant l'aspect purement fonctionnel et l'aspect de configuration pour un niveau de QdS donné (tel qu'identifié dans les architectures à méta-niveaux).

   De plus, la réutilisabilié est liée aux objectifs d'hétérogénéité et de d'auto-configurabilité. En effet, un composant pouvant être réutilisé doit prévoir de s'exécuter dans des contextes différents et donc prévoir l'hétérogénéité des QdS. De même, lorsqu'un composant est réutilisé dans des contextes différents, ses niveaux de QdS requis doivent être re-évalués. Cela signifie que pour que la réutilisabilité puisse être effectuée à moindre coût, elle doit se faire entre des systèmes proposant des caractéristiques matérielles et des contextes logiciels relativement similaires.
6. **la confiance :** l'objectif, ici, est de permettre la mise en oeuvre de mécanismes s'assurant que les niveaux

de QdS utilisés sont conformes aux niveaux contractualisés. La confiance de l'architecture se base sur deux concepts qui sont les attributs des tables de mapping (*reliable/unreliable*), permettant de définir leur niveau de fiabilité, et les tests de QdS consommée, réalisés par les QoSComponents. Le premier concept permet à l'architecture de fixer un paramètre de confiance aux différents contrats et de gérer la QdS à partir des niveaux contractualisés. En particulier, bien qu'une table de mapping puisse être définie comme fiable, elle peut requérir un niveau de QdS bien supérieur au niveau réellement utilisé. Qinna ne permet pas de se prémunir de genre de comportement qui aboutit à un gaspillage de QdS. Ce problème pose alors le problème de la définition d'une politique de sûreté permettant de définir si une table de mapping est fiable ou non. Le deuxième concept proposé par Qinna, permet de contrôler la consommation de QdS à partir des niveaux de QdS contractualisés. Les tests réalisés par les QoSComponents peuvent être difficiles à mettre en oeuvre suivant le service rendu. Dans les cas des composants ressources (CPU, mémoire, réseau) ces tests sont plus classiques et faciles à implémenter. C'est pourquoi, en pratique, l'architecture se base principalement sur la confiance des contrats des niveaux de ressources.

L'identification de ces réponses permettent de justifier à posteriori le choix d'une architecture à base de composants et l'utilisation de contrats pour la gestion de la QdS. En effet, les composants sont à la base de la réponse apportée à la problématique de réutilisation. De plus, la mise en oeuvre de l'architecture à base de composant permet de quantifier précisément le coût de chaque opération de gestion de QdS dans une optique d'optimisation de l'architecture. Par exemple, dans le cas du QoSComponentBroker, il est alors possible d'identifier clairement le coût du test d'admission.

L'utilisation de contrats, et de sous-contrats, permet d'identifier les niveaux de QdS de chaque composant du système apportant ainsi une partie de la réponse à la problématique de confiance. Les contrats permettent aussi de faciliter la gestion des niveaux de QdS des composants en contractualisant les niveaux de QdS de chaque composant. En effet, les contrats identifie une *enveloppe* de la QdS réellement nécessaire aboutissant ainsi à une meilleure maîtrise de la gestion de la QdS. Cette approche est à rapprocher de celle utilisée dans la conception de systèmes temps-réel où les tâches sont caractérisées par leur pire durée d'exécution, constituant une *enveloppe* sûre, et non sur la durée réelle (souvent difficile à caractériser). Dans le cas de Qinna, le choix de l'*enveloppe* revient à effectuer un compromis entre le gaspillage de QdS et le nombre d'opérations d'adaptation à effectuer. Enfin, les contrats, tels que définis par Qinna, permettent d'expliciter des liens, et des choix, nécessairement existants entre les offres et les besoins de QdS des différents composants du système.

Chapitre 5

Conclusion et perspectives

Cette thèse propose une architecture à base de composants pour la gestion de la QdS pour les systèmes embarqués ouverts à composants. Les principaux objectifs de l'architecture sont la prise en compte de la dynamicité, de l'hétérogénéité, de la généricité, de la confiance, de l'auto-configurabilité et de la réutilisabilité.

L'étude des différents modèles de composants, ainsi que des architectures de gestion de QdS, ont permis d'identifier les principes clés devant être mis en oeuvre par une architecture de gestion de QdS pour les composants. Ces principes sont, premièrement, la séparation des préoccupations fonctionnelles et de QdS et la séparation des politiques et des mécanismes de gestion de QdS. Le deuxième principe clé est l'utilisation des contrats de QdS et des activités liées à leur gestion dynamique. Ces activités, identifiées dans les architectures de gestion de QdS, portent sur la spécification des contrats, leur initialisation (incluant les phases de mapping de QdS entre les niveaux applicatif, service, système d'exploitation et ressources d'un système, de tests d'admission et de réservation), leur observation, leur adaptation, ainsi que leur maintenanabilité. Le troisième principe porte sur l'utilisation du principe de conteneur de composants permettant de définir un point d'entrée unique pour la gestion de la QdS. Le quatrième principe consiste à maximiser les niveaux de QdS des composants afin d'utiliser au mieux les ressources matérielles de la plateforme. Le cinquième principe consiste à concevoir un système comme une composition homogène de composants (du niveau ressources au niveau applicatif) et à permettre leur identification pendant l'exécution afin de pouvoir les gérer dynamiquement, ainsi que la QdS qui leur est associée. Enfin, le dernier principe porte sur l'efficacité de l'architecture.

Afin de respecter ces principes, une architecture de gestion de QdS pour les systèmes à composants doit s'appuyer sur un modèle de composants suffisamment générique et flexible. Pour ces raisons, l'architecture proposée, appelée Qinna, se base sur la définition de composants conforme au modèle Fractal et utilise les propriétés du framework à composants Think. En effet, Think permet l'identification des composants en cours d'exécution du système. De plus, Think répond au besoin d'efficacité en étant implémenté par des langages de bas niveau (C et assembleur).

L'architecture proposée par cette thèse est définie au travers d'un ensemble de types de composants (QoSComponent, QoSComponentBroker, QoSComponentManager, QoSDomain et QoSComponentObserver), d'interfaces et de comportements dynamiques. Chacun des types de composants est en charge d'une des activités de gestion des contrats de QdS. La spécification des besoins de QdS est capturée par les interfaces du QoSDomain (cf. tableau 3.1

page 38), le mapping est réalisé par les QoSComponentManagers et le test d'admission, ainsi que la réservation, sont réalisés par les QoSComponentBrokers. Les aspects d'adaptation et de maintenance sont réalisés par les QoSComponentManagers et le QoSDomain, tandis que l'aspect d'observation est traité par les QoSComponentObservers. Cette approche permet de séparer, entre les différents composants, les politiques et les mécanismes de gestion de QdS, ainsi que les préoccupations fonctionnelles et de QdS du système. Les QoSComponents implémentent les préoccupations fonctionnelles du système, alors que les autres composants de l'architecture (QoSComponentBrokers, QoSComponentManagers, QoSDomain et QoSComponentObservers) en implémentent les préoccupations de QdS. Au sein de la préoccupation de QdS, le principe de séparation des politiques et des mécanismes est respecté : le QoSDomain implémente les politiques d'adaptation et de maintenance des contrats de QdS, tandis que les mécanismes correspondants sont implémentés par les QoSComponentManagers.

L'évaluation de Qinna au travers d'un ensemble de cas types présents dans les systèmes embarqués ouverts à composants a permis de caractériser les apports et les limites de l'architecture. L'architecture permet d'apporter des réponses aux problématiques de gestion de QdS identifiées dans les systèmes embarqués ouverts. Qinna identifie principalement deux domaines d'applications privilégiés que sont les systèmes multimédia et les systèmes temps-réel. Dans le cas des systèmes multimédia, Qinna apporte les bénéfices liés à la gestion dynamique de la QdS, tandis que dans le cas des systèmes temps-réel l'apport se situe au niveau de la structuration des éléments de gestion de la QdS. Bien que Qinna possède les concepts nécessaires à une gestion dynamique de la QdS, leur mise en oeuvre conduit à un compromis, que doit effectuer le concepteur du système, entre un gaspillage de QdS (discrétisation de la QdS) et un nombre important d'opérations d'adaptation à réaliser par l'architecture (suivi de la variabilité des profils de QdS requis et des capacité des ressources matérielles). La prise en compte de l'hétérogénéité de QdS peut être réalisée de façon peu coûteuse dans la mesure où les différents types de QdS ont été prévus. De même que pour la dynamicité, la prise en compte de l'autoconfigurabilité amène à effectuer un compromis entre le gaspillage de QdS (discrétisation du niveau de QdS contractualisé) et le nombre d'opérations à réaliser par l'architecture (niveau de QdS contractualisé proche du niveau réel utilisé). La réutilisabilité s'appuie sur l'utilisation de composants pour la définition de l'architecture, ainsi que sur la mise en oeuvre des principes de séparation des préoccupations fonctionnelles et de QdS, et des politiques et des mécanismes de QdS. Cependant, la réutilisation pourrait être encore augmentée en effectuant une séparation plus forte des préoccupations fonctionnelles et de configurations dans les QoSComponents. Enfin, Qinna apporte des éléments de réponses à la problématique de confiance des composants, en se basant sur des attributs de fiabilité des tables de mapping et des tests de QdS consommée réalisés par les QoSComponents. En pratique, la confiance de l'architecture se situe essentiellement au niveau des ressources matérielles de la plateforme et doit s'appuyer sur une politique de sécurité.

Perspectives

Les résultats obtenus lors de ce travail de thèse permettent d'entrevoir de nombreuses perspectives de recherche que nous exposons ici :

– premièrement, deux principales pistes ont été identifiées afin d'atténuer les coûts liés à la dynamicité. La première consiste à utiliser des méta données par application, une application étant constituée de composants, afin de regrouper les informations nécessaires à l'établissement d'un contrat et de ses sous-contrats. Cette approche permet alors d'éviter les coûts liés à la gestion décentralisée des contrats réalisée par Qinna. En effet dans ce cas, tous les tests d'admission sont réalisés directement sans être délégués à des composants spécialisés. La deuxième piste consiste à modifier la sémantique des services *reserve* des différents composants de l'architecture. Ces services permettent de tester si un sous-contrat peut être établi puis de

l'établir. Dans le cas où l'établissement d'un sous contrat échoue, il est alors nécessaire d'annuler tous les sous-contrats précédemment enregistrés. Il serait intéressant de découpler ces deux opérations en deux services distincts : l'un permettant de tester si un sous-contrat peut être établi et un autre permettant de l'établir. Dans le cas où l'établissement du dernier sous contrat échoue, cette approche permet de diviser par deux le nombre d'appels entre les différents composants (car il n'est pas nécessaire d'annuler les sous-contrats établis), mais multiplie par deux le nombre d'appels lorsque tous les sous-contrats sont établis (un appel de test et un appel de confirmation).

- la seule auto-configuration prévue par l'architecture Qinna prévoit l'évaluation des niveaux de QdS requis des composants. Cependant, dans le cadre des systèmes ouverts, il serait intéressant de prévoir les mécanismes permettant l'accueil d'un composant seul. Une piste possible est de lui associer, à son arrivée, des composants de gestion de QdS totalement génériques et de les spécialiser, par l'apprentissage, au composant. Il s'agirait alors de déterminer les mécanismes d'adaptation et de maintenabilité, les tables de mapping, ainsi que les tests d'admission et les éléments de configuration des QoSComponents.
- l'architecture proposée ne traite actuellement que les composants fournissant une seule interface nécessitant une gestion de la QdS. Bien que cette hypothèse permette déjà d'apporter des réponses aux problématiques de gestion de QdS des systèmes ouverts à composants, il est nécessaire d'étendre l'architecture afin de pouvoir gérer plusieurs interfaces fournies avec gestion de QdS. Dans cette optique, il serait nécessaire de définir plusieurs tables de mapping, chacune étant dédiée à une seule interface. Les QoSComponentManagers devraient alors gérer toutes les tables de mapping des interfaces fournies d'un même composant et devraient, en particulier, gérer les aspects d'inter-dépendances entre les différentes tables.
- l'architecture Qinna se base sur les propriétés du modèle à composants Fractal et de framework Think, il serait intéressant d'étudier si Qinna peut être appliquée à d'autres modèles. Cette perspective est actuellement en cours d'étude dans le cadre du projet RNRT PISE [4] visant à intégrer des mécanismes de gestion de QdS pour les plateformes OSGi. Dans ce cadre, la plateforme d'exécution OSGi ne permet pas d'accéder aux ressources. Il n'est donc pas possible de contrôler la consommation des niveaux de QdS de chaque composant. Cependant, en se basant uniquement sur les déclarations d'intentions des composants, Qinna permet de proposer une approche permettant de structurer les activités de gestion dynamique de la QdS. De plus, l'utilisation d'OSGi permettrait d'adresser les problématiques de gestion du cycle de vie des composants ce qui n'est pas traité par l'architecture Qinna.
- l'architecture Qinna fait l'hypothèse d'un système mono-processeur et non-distribué. Cependant, les systèmes embarqués mobiles embarquent, de plus en plus, plusieurs processeurs (par exemple un processeur dédié à l'interface homme/machine et un autre aux communications). De plus, Qinna devrait pouvoir être étendue aux systèmes distribués. Il est alors nécessaire d'étudier Qinna dans un contexte réparti et distribué. Deux approches peuvent alors être explorées. La première consiste à mettre en place un QoSDomain ayant une vision globale du système permettant ainsi de ne pas modifier l'architecture. Cette approche est celle mise en place par les plateformes distribuées telles que CORBA. La deuxième approche est de permettre à plusieurs QoSDomain de collaborer ensemble. Cette approche est celle utilisée par les technologies agents.
- à un niveau amont du développement, la définition d'un langage de spécification de QdS dédié aux composants permettrait de générer automatiquement les composants Qinna correspondants. Dans cette perspective, le langage CQML [49] apparaît comme une solution qu'il est nécessaire d'évaluer.
- la QdS telle que nous la traitons dans cette thèse n'est pas l'unique propriété extra-fonctionnelle devant être gérée dans les systèmes ouverts à composants. Nous pouvons citer, par exemple, les propriétés liées à la sécurité ou à la tolérance aux fautes. Il est donc utile de pouvoir disposer d'une architecture permettant la gestion de l'ensemble de ces propriétés (QdS, sécurité et tolérance aux fautes). Une approche choisie pourrait être de tout d'abord définir une architecture générique, qu'il faudrait ensuite instancier pour une propriété

particulière : cette approche permettrait alors de mutualiser les outils liés aux différentes propriétés. De plus, il sera alors nécessaire de se pencher sur le problème de la composition des architectures, qui reviendra dans ce cas au problème de la composition des propriétés extra-fonctionnelles.

- afin de faciliter la mise en oeuvre de l'architecture, il serait nécessaire de disposer d'une plateforme de tests permettant de simuler différents profils pour chaque ressource matérielle possible. Cet outil permettrait alors d'évaluer les profils des niveaux de QdS des composants de niveau applicatifs, services ou système d'exploitation, permettant ainsi de générer directement les composants Qinna associés.
- dans la même optique que l'aide à la mise en oeuvre de l'architecture, il serait utile de travailler à la formalisation de l'implémentation de Qinna. En effet, il serait intéressant de disposer de règles, qui à partir des caractéristiques du systèmes (tels que la granularité des composants, le nombre de niveaux de QdS par composant, les profils de QdS requis des composants ou encore les profils des capacités des ressources), permettraient de guider le concepteur dans les nombreux choix à réaliser lors de l'implémentation de l'architecture (par exemple, les pas de maintenance, les politiques d'observation ou les stratégies de suivi de la capacité d'une ressource).
- enfin, il serait nécessaire de travailler à la définition d'un méta-modèle de l'architecture Qinna. Les intérêts à une telle définition sont doubles. Premièrement, cela faciliterait le développement d'outils de génération automatique de l'implémentation de l'architecture suivant un contexte donné. Deuxièmement, l'utilisation d'un méta-modèle permettrait de vérifier des propriétés de l'architecture à l'implémentation, telle que, par exemple, la présence d'un QoSComponentObserver pour tous les contrats possédant des données non fiables ou les liaisons correctes entre les différents composants de l'architecture. Naturellement, une des pistes pour la définition d'un tel méta-modèle est d'utiliser les diagrammes de classes pour les types de composants Qinna et les diagrammes de séquences pour les comportements dynamiques de l'architecture.

Bibliographie

[1] LxDoom. Technical report, http ://prboom.sourceforge.net.

[2] The QoS Framework. Technical report, TINA-C, Internal Rechnical Report, 1995.

[3] Quality of Service, draft Paper. Technical report, Object Management Group, 1998.

[4] Projet PISE : Passerelle Internet Sécurisée et flexiblE. Technical report, Réseau National de Recherche en Télécommunications, 2005.

[5] Sten Amundsen, Ketil Lund, Frank Eliassen, and Richard Staehli. QuA : Platform-Managed QoS for Component Architectures. In *NIK 2004*, 2004.

[6] Christina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A Survey of QoS Architectures. *Multimedia Systems*, 6(3) :138–151, 1998.

[7] Jean-Philippe Babau and Jean-Louis Sourrouille. Expressing Real Time Constraints in a Reflective Object Model. *Control Engineering Practice* , 6 :421–430, 1998.

[8] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, Volume II. Technical Report CMU/SEI-2000-TR-008. Technical report, Software Engineering Institute, Carnegie-Mellon University, Mai 200.

[9] Fred Barwell, Richard Blair, Jonathan Crossland, Richard Case, Bill Forgey, Whitney Hankison, Billy S. Hollis, Rockford Lhotka, Tim McCarthy, and John C. Roth. *Professional VB.NET*. Number 1861007167. Wrox Press, 2002.

[10] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in pratice*. Addison-Wesley, 2003.

[11] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7) :38–45, 1999.

[12] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. OYOTE : A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4) :321–366, 1998.

[13] A.W. Brown. *Large-Scale Component-Based Development*. Upper Saddle River, Prentice Hall, 2000.

[14] Eric Bruneton. Fractal adl tutorial. Technical report, France Télécom R&D, http ://fractal.objectweb.org/tutorials/adl/, 2004.

[15] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, May 2004.

[16] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Specification of the Fractal Component Model. Technical report, http ://fractal.objectweb.org, 2003.

[17] Andrew Campbell, Geoff Coulson, Francisco Garcia, David Hutchison, and Helmut Leopold. Integrated Quality of Service for Multimedia Communications. In *Proceedings of the IEEE INFOCOM'93 conference*, pages 732–739, 1993.

[18] Andrew T. Campbell. *A Quality of Service Architecture*. PhD thesis, Computing Department - Lancaster University, 1996.

[19] Gerald Carter and Sébastien Pujadas. *LDAP : Admistration Système*. O'Reilly, 2003.

[20] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[21] CNRS. Rapport de synthèse as 195 - composants et architectures temps réel. Technical report, CNRS, 2005.

[22] Compaq. *iPAQ H3800 Series Pocket PC Getting Started Guide*, 2001.

[23] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House Publishers, July, 2002.

[24] Jérôme Daniel, Olivier Modica, Bruno Traverson, and Sylvie Vignes. Gestion de la Qualité de Service dans une palte-forme à composants. In *3eme Colloque International sur les NOuvelles TEchnologies de la REpartition (NOTERE'2000)*, 2000.

[25] Olivier Defour, Jean-Marc Jézéquel, and Noel Plouzeau. Extra-functional contract support in components. In *Proceedings of International Symposium on Component-based Software Engineering (CBSE7)*, pages 217–232, Mai 2004.

[26] Joëlle Delacroix. Stabilité et régisseur d'ordonnancement en temps-réel. *Technique et science informatique*, 2(2) :223–250, 1994.

[27] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-oriented Language for Communicating Systems. SDL - Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe, 1997.

[28] Jean-Philippe Fassino. *Think : vers une architecture des systèmes flexibles*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2001.

[29] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. Think : A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002.

[30] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of static properties for component-based architectures. In *Proc. 28th EUROMICRO Conference*, pages 33–39, Dortmund, Germany, September 2002. IEEE Comp. Soc. Press.

[31] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit : A Substrate for Kernel and Language Research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[32] III Redmond Frank E. *DCOM : Microsoft Distributed Component Object Model*. Number 0764580442. Hungry Minds Inc, 1997.

[33] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble Component-Based Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 267–282, 1999.

[34] The Open Services gateway Initiative. "osgi service platform release 3". Technical report, 2003.

[35] R. Gopalakrishnan and G. M. Parulkar. Efficient Quality of Service Support in Multimedia Computer Operating Systems. Technical report, Technical Report 94-26, Dept. of Computer Science, Washington University in St. Louis, 1994.

[36] Alan Ira Gordon. *The COM and COM+ Programming Primer*. Microsoft Technologies Series, Avril 2000.

[37] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[38] Object Management Group. "uml 2.0 infrastructure specification". Technical report, OMG Adopted Specification ptc/03-09-15, 2003.

[39] Object Management Group. Unified modeling language specification. Technical report, OMG Version 1.5 formal/03-03-01, 2003.

[40] Object Management Group. "uml profile for modeling quality of service and fault tolerance characteristics and mechanisms. Technical report, 2004.

[41] George T. Heinman and William T. Council. *Component-Based software engineering : Putting the piece togheter*. Addison Wesley, 2001.

[42] Anders Hejlsberg. *The C# Programming Language*. Number 0321154916. Addison-Wesley Pub Co, 2003.

[43] Johannes Helander and Alessandro Forin. MMLite : a highly componentized system architecture. In *EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103. ACm-Press, 1998.

[44] Alx Homer and David Sussman. *Professional MTS and MSMQ with VB and ASP*. John Wiley & Sons, 1998.

[45] Intel. Intel SA-1100 Processor. Technical report, Intel Corporation.

[46] ISO. Quality of Service Framework. Technical report, International Standards Organisation - ISO/IEC JTC1/SC21/WG1 N9680, 1995.

[47] ISO. Interface Description Language JTC 1/SC 7. Technical report, International Organization for Standardization, 1999.

[48] D. Isovic and C. Norstrom. Components in real-time systems. *8th International Conf. on Real-Time Computing Systems and Applications (RTCSA'2002)*, 2002.

[49] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[50] Tahar Jarboui, Jean-Philippe Fassino, and Marc Lacoste. Applying Components to Access Control Design : Towards a Framework for OS Kernels. In *International Conference on Dependable Systems and Networks (DSN 2004)*, Florence, Italy, 2004.

[51] JBoss. *http ://www.jboss.org*, 2004.

[52] JOnAS. *http ://jonas.obectweb.org*, 2004.

[53] M. Jones, J. Alessandro, F. Paul, J. Leach, D. Rosu, and M. Rosu. An Overview of the Rialto RealTime Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, 1996.

[54] Fabio Kon, Roy Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K : A Distributed Operating System for Dynamic Heterogeneous Environments. In *9th IEEE International Symposium on High Performance Distributed Computing*, Août 2000.

[55] Othmar Kyas and Gregan Crawford. *ATM Networks*. Number 0130936014. Prentice Hall, 2002.

[56] Linda Fedaoui Wassim Tawbi K. Thai L. Besse, Laurent Dairaine. Towards an Architecture for Distributed Multimedia Applications Support. In *Proceedings of International Conference on Multimedia Computing and Systems*, pages 164–172, 1994.

[57] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 3(16) :872–923, 1994.

[58] O. Layaïda and D. Hagimont. A Component-based Framework for Building Self-Adaptive Applications. In *Proceedings of SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications*, San Jose, USA, Janvier 2005.

[59] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, 1999.

[60] Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/mechanism separation in HYDRA. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*, pages 132–140, Austin, Texas, USA, Nov 1975.

[61] Jesse Liberty and Dan Hurwitz. *Programming ASP. NET*. Number 0596004877. O"Reilly, 2003.

[62] Wayne C. Lim. *Managing Software Re-Use*. Prentice Hall, 1998.

[63] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[64] Stéphane Lorcy. *Infrasctructure logicielle pour la gestion de la cohérence et de la qualité de service d'un environnement à objets réparti : application au télétravail coopératif*. PhD thesis, Université de Rennes 1, 2000.

[65] Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. A framework for QoS-aware software components. In *WOSP '04 : Proceedings of the fourth international workshop on Software and performance*. ACM Press, 2004.

[66] B. Meyer. *"Eiffel : The Language"*. Prentice Hall, 1991.

[67] Microsoft. Com : Component object model technologies - http ://www.microsoft.com/com/. Technical report, 1995.

[68] Microsoft. *.NET - http ://www.microsoft.com/net*, 2002.

[69] Microsoft. *Microsoft .NET Passport - http ://www.passort.net*, 2004.

[70] Sun Microsystems. *Java RMI Specification*, 1997.

[71] Sun Microsystems. *Java Message Service (JMS) Specification*, 2002.

[72] Sun Microsystems. JSR 153 : Enterprise JavaBeans 2.1. Technical report, Java Community Process, 2003.

[73] Mono. *http ://www.mono-prject.com*, 2004.

[74] K. Nahrstedt and J. Smith. Design, Implementation and Experiences of the OMEGA End-Point Architecture. Technical report, Technical Report (MS-CIS-95-22), University of Pennsylvania, 1995.

[75] Object Management Group. *Corba Component Model V3.0 - http ://www.omg.org/technology/documents/formal/components.htm*, 2002.

[76] Oracle. *Oracle9iAS Containers for J2EE - http ://otn.oracle.com/tech/java/oc4j/content.html*, 2004.

[77] OSGi. OSGi Service Gateway Specification, Release 1.0. Technical report, Open Service Gateway Initiative, http ://www.osgi.org.

[78] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, dec 1972.

[79] ARTIST Project. Selected topics in embedded systems design : Roadmaps for research. Technical Report Project IST-2001-34820, Information Society technologies, 2004.

[80] PECOS Project. *http ://www.pecos-project.org*, 1999.

[81] David Sceppa. *Microsoft ADO.NET (Core Reference)*. MicrosoftPress, 2002.

[82] Jens Schmitt and Lars Wolf. Quality of Service - An Overview. Technical report, Darmstad University of Technology - Department of Electrical Engineerring , 1997.

[83] Benedikt Schulz, Thomas Genssler, Alexander Christoph, and Michael Winter. Requirements for the Composition Environment. Technical report, PECOS Project Deliverable D2.2.9-1, 1999.

[84] Nicolas Le Sommer. *Contractualisation des ressources pour les composants logiciels : une approche réflexive*. PhD thesis, Université de Bretagne Sud, 2003.

[85] Marco Spuri and Giorgio Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Journal of Real Time Systems*, 1996.

[86] John Stankovic, Tarek Abdelzaher, Zhimin He, Prashant Nagarradi, and Zhendong Yu. Vest : Virginia embedded systems toolkit. pages 390–402, 2001.

[87] BEA Systems. *BEA WebLogic Server - http ://www.bea.com/products/weblogic/server*, 2002.

[88] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[89] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach : Towards Predictable Real-Time Systems. In *Proceedings of the USENIX 1990 Mach Workshop*, 1990.

[90] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. The Qinna Experiment, a Component-Based QoS Architecture for Real-Time Systems. In *Workshop Architectures for Cooperative Embedded Real-Time Systems in conjunction with the 25th real-Time System Symposium*, 2004.

[91] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. An Evaluation of Qinna, a Component-Based QoS Architecture for Handheld Systems. In *Proceedings of the ACM Symposium on Applied Computing*, 2005.

[92] International Telecommunication Union. ITU-T Recommendation E.800 - Terms and definitions related to quality of service. Technical report, ITU Telecommunication Standardization Sector (ITU-T), 1994.

[93] David Urting, Yolande Berbers, Stefan Van Baelen, Tom Holvoet, Yves Vandewoude, and Peter Rigole. A tool for component based design of embedded software. In *CRPITS '02 : Proceedings of the Fortieth International Conference on Tools Pacific*, pages 159–168, 2002.

[94] Rob van Ommering. Building product populations with software components. *22rd International Conference on Software Engineering (ICSE'2002)*, pages 255–265, May 2002.

[95] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 3(33) :78–85, 2000.

[96] Greg Voss. Java Beans : Introducing Java Beans. Technical report, Sun Microsystems, http ://java.sun.com/developer/onlineTraining/Beans/Beans1/, 1996.

[97] Kurt C. Wallnau, Scott A. Hissam, and Robert C. Seacord. *Building systems from commercial components*. Addsion Wesley, 2002.

[98] Jiacun Wang. *Timed Petri Nets*. Kluwer Academic Publishers, 1998.

[99] J. Warmer and A. Kleppe. *"The Object Constraint Language : Precise Modelling with UML"*. Addison-Wesley, 1999.

[100] IBM WebSphere. *Middleware, application server, e-business, infrastructure software - http ://www.ibm.com/websphere*, 2004.

Table des figures

2.1 Méta modèle d'un composant UML2.0. 7

2.2 Représentation graphique de composition de composants Fractal. 8

2.3 Architecture d'exécution des EJB. 13

2.4 Cycle de vie d'un bundle OSGi. 16

2.5 Méta-modèle du modèle de composants PECOS. 20

2.6 Représentation graphique du mécanisme mis en oeuvre par Think pour l'utilisation d'interface. . . 22

2.7 Propriétés d'un contrat de QdS. 26

2.8 Architecture de l'ISO de gestion de QdS. 27

2.9 Architecture QoS-A de gestion de QdS. 28

2.10 Architecture QuA de gestion de QdS. 29

3.1 Vue globale de l'architecture Qinna. 36

3.2 Représentation graphique du composant *Nom_Composant1* lié au composant *Nom_Composant2* par la liaison *L* et fournissant une interface non liée (*i1*) et requérant une interface non liée (*i2*). . . 36

3.3 Représentation d'un QoSComponent. 39

3.4 Représentation d'un QoSComponentBroker. 39

3.5 Représentation d'un QoSComponentManager. 40

3.6 Représentation d'un QoSComponentObserver. 41

3.7 Représentation d'un QoSDomain. 42

3.8 Réussite lors de la mise en place du contrat lié à l'activation du composant QoSX utilisant un composant QoSY. 44

3.9 Dégradation du contrat lié au composant QoSX utilisant le composant QoSY. 45

3.10 Amélioration réussie du contrat lié au composant QoSX utilisant le composant QoSY. 46

3.11 Échec de l'amélioration du contrat lié au composant QoSX utilisant le composant QoSY. 47

3.12 Notification au QoSDomain de la violation du contrat lié à QoSX, dans le cas d'une observation initiée par QoSXObserver. 47

3.13 Notification au QoSDomain de la violation du contrat lié à QoSX, dans le cas d'une observation initiée par QoSX. 48

3.14 Mise à jour des données du contrat lié au composant QoSX utilisant le composant QoSY. 48

3.15 Étapes de mise en place de Qinna. 50

3.16 Représentation du système initial. 53

3.17 Représentation du système présenté à la figure 3.16 après intégration de Qinna. 55

3.18 Réservation réussie du composant QoSVideo de niveau BON. Afin de ne pas surcharger le diagramme, les requêtes liées au QoSMemBroker ont été omises. 58

4.1 Représentation graphique d'un composant A utilisant les services de n ressources R_i. 62

4.2 Intégration de Qinna au cas type d'utilisation de n ressources par un composant tel que présenté à la figure 4.1. 62

4.3 Cas type de partage de composant. 65

4.4 Représentation des notations de QdS pour le cas type de partage de composant. 65

4.5 Représentation du partage d'un composant sécable C entre les composants A et B. 65

4.6 Intégration de Qinna dans le cas d'un partage de composant sécable. 66

4.7 Partage d'un composant C insécable. 68

4.8 Intégration de Qinna pour le partage d'un composant insécable. 69

4.9 Représentation sous forme d'arbre du framework d'ordonnancement hiérarchique défini par [37]. . 71

4.10 Intégration de Qinna au framework d'ordonnancement hiérarchique. 72

4.11 Représentation des tâches et des ordonnanceurs d'un système temps-réel exécutant une machine virtuelle Java. 72

4.12 Intégration de Qinna au système de la figure 4.11 représentant des tâches et des ordonnanceurs d'un système temps-réel exécutant une machine virtuelle Java. 73

4.13 Représentation arborescente d'une composition et de ses liaisons. 77

4.14 Représentation arborescente des liaisons des QoSComponentManagers après intégration de Qinna du système représenté à la figure 4.13. 77

4.15 Enchaînement des opérations lors de l'activation d'un QoSComponent. 79

4.16 Profil de QdS requise variable. 82

4.17 Évolution du niveau de QdS requis contractualisé par QoSA. 85

4.18 Représentation arborescente des n_i niveaux de QdS de N contrats. 86

4.19 Évolution de la capacité du réseau en fonction du temps. 88

4.20 Renégociation des contrats lors d'une modification de relation d'ordre. 90

4.21 Système multimédia intégrant l'architecture Qinna et exécutant le jeu Doom et une vidéo. 93

4.22 Système multimédia intégrant l'architecture Qinna et exécutant une vidéo. 96

Liste des tableaux

3.1 Interfaces définies par l'architecture Qinna. 38

3.2 Table de mapping du composant VIDEO. 53

3.3 Table de mapping du composant DECO. 53

3.4 Évolution de l'état du système au travers des contraintes globales courantes lors de la réservation d'un QoSVideo de niveau BON. 57

4.1 Quantité de mémoire requis par les composants A,B et C pour chaque niveau de QdS. 80

4.2 Table de mapping de QoSVideo. 94

4.3 Table de mapping de QoSDoom. 94

4.4 Taille mémoire des composants du système intégrant Qinna. 94

4.5 Taille mémoire des composants du système n'intégrant pas Qinna. 94

4.6 Taille mémoire des composants. 97