Gestion autonomique de la QoS au niveau middleware dans l'IoT

Yassine Banouar

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *21/09/2017* **par :**

Yassine BANOUAR

Gestion Autonomique de la QoS au niveau Middleware dans l'IoT

JURY

Gladys DIAZ	Maître de conférences, HDR	Rapporteure
Didier DONSEZ	Professeur d'Université	Rapporteur
Olivier FOURMAUX	Maître de conférences, HDR	Examinateur
Thierry GAYRAUD	Professeur d'Université	Examinateur
Khalil DRIRA	Directeur de Recherche	Invité
Christophe CHASSOT	Professeur d'Université	Directeur de thèse
Thierry MONTEIL	Professeur d'Université	Co-directeur de thèse

---

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
LAAS - CNRS (UPR 8001)
Directeur(s) de Thèse :
Christophe CHASSOT et *Thierry MONTEIL*
Rapporteurs :
Gladys DIAZ et *Didier DONSEZ*

A ma mère, à Hasna.

Remerciements

Les travaux de cette thèse présentés dans ce manuscrit ont été réalisés au sein du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS - CNRS). Je tiens donc à remercier Messieurs Jean ARLAT et Liviu NICU respectivement, précédent et actuel directeurs du LAAS, pour leur accueil dans ce laboratoire.

Je remercie également Monsieur Khalil DRIRA, responsable de l'équipe Services et Architectures pour Réseaux Avancés (SARA) du LAAS-CNRS pour l'accueil dans son équipe depuis mon premier stage dans ce laboratoire.

Je sais gré à Madame Gladys DIAZ et Monsieur Didier DONSEZ pour leurs commentaires et pour avoir accepté de rapporter ce travail de thèse. Je remercie Messieurs Olivier FOURMAUX et Thierry GAYRAUD (président du jury) qui ont accepté d'examiner mon travail, ainsi que Monsieur Khalil DRIRA pour avoir accepté d'être membre invité de mon jury de thèse.

Mes sincères remerciements et ma gratitude à Christophe CHASSOT, directeur de ma thèse, pour tout ce qu'il m'a appris tout au long de ces années. Ses conseils techniques et scientifiques m'ont permis d'aller jusqu'au bout de ma thèse et ont amélioré mon savoir-faire. Ses conseils personnels ont participé à devenir ce que je suis aujourd'hui et ont amélioré mon savoir-être. Je le remercie pour son soutien inconditionnel depuis mon premier stage au LAAS jusqu'à l'aboutissement de ce travail de thèse, et j'espère, au-delà.

Merci à Thierry MONTEIL, co-directeur de cette thèse, pour son intervention technique sur la partie modélisation analytique.

Je remercie aussi tous mes collègues et amis du LAAS pour nos moments de soutiens conjoints ainsi que leurs conseils qui m'ont été d'une grande aide. Mes remerciements aussi aux personnes avec qui j'ai pu collaborer durant cette thèse, notamment à Yassine, Saad, Clovis et El-Fadel durant leurs stages.

Un remerciement à tout le personnel administratif du LAAS-CNRS, de l'école doctorale MITT et de l'UPS.

Un grand remerciement à ma famille, mon père et mon frère, surtout à ma mère qui m'a encouragé depuis le début et m'a épaulé durant toutes ces années pour poursuivre les chemins que je me traçais. Aucun mot ne serait suffisant pour lui montrer ma grande reconnaissance et ma gratitude. Je remercie également mes amis pour leurs encouragements, notamment, Bouchta, Codé, Ghada, Hajar, Naoufal et Nargisse.

L'aboutissement de ce travail de doctorat doit aussi beaucoup à mon épouse, Hasna, pour sa patience et sa compréhension inconditionnelles depuis le début de ma thèse. Mais aussi, surtout, pour sa présence, son soutien et ses encouragements durant les moments d'incertitude. Elle a su mettre de petites lampes pour éclairer mes idées et me réconforter durant ces moments. Avec ces quelques mots, je ne saurais lui donner sa vraie part de remerciement.

Je remercie enfin toute personne ayant été impliquée de près ou de loin dans l'aboutissement de ce travail de doctorat.

Yassine BANOUAR

Résumé

L'Internet connaît à nouveau une expansion drastique. En plus des terminaux classiques, il permet aujourd'hui d'interconnecter toute sorte d'*objets connectés* permettant la capture d'événements depuis l'environnement considéré, mais également le contrôle à distance de cet environnement. Plusieurs milliards de ces objets sont ainsi amenés à l'horizon 2020 à contribuer à l'avènement de l'*Internet des Objets* (IoT). Ce paradigme, qui étend le concept de *Machine-to-Machine* (M2M), ouvre la voie à de nouveaux usages tels que la domotique, la télésurveillance, ou encore les usines du futur.

Plusieurs architectures ont été proposées pour structurer l'IoT. Leur fondement est basé sur une vision en quatre niveaux : le niveau *Équipement,* qui comporte les objets connectés, le niveau *Réseau* contenant les différentes technologies nécessaires aux échanges, le niveau *Intergiciel* (ou *Middleware*) qui offre aux applications une couche d'abstraction des niveaux sous-jacents, et enfin le niveau *Application* qui consiste en l'ensemble des applications concourant, via leurs interactions avec les objets connectés, à la réalisation d'une activité métier. Nos travaux se positionnent au niveau Middleware sur la base de l'architecture définie dans le cadre des standards SmartM2M puis oneM2M. Plusieurs problématiques sont amenées à être (re)-posées dans ce contexte. Nous nous intéressons essentiellement à celle de la qualité de service (QoS - *Quality of Service*) exprimée par certaines applications métiers.

Les solutions proposées en réponse à cette problématique concernent principalement le niveau Réseau. Au niveau Middleware, les standards se focalisent essentiellement sur la proposition d'architectures et de services fonctionnels. Les besoins non fonctionnels, typiquement orientés QoS, ne sont que peu ou pas considérés. Parallèlement, les solutions propriétaires ne considèrent pas l'évolution dynamique du contexte et des besoins. Face à ces limites, nous proposons une approche de gestion *dynamique*, i.e. durant l'exécution du système, et *autonome* induisant un minimum d'intervention humaine. La gestion proposée, guidée par des modèles, porte sur des actions de reconfiguration comportementales et structurelles touchant au trafic applicatif et/ou sur les ressources de niveau Middleware.

La première contribution de cette thèse porte sur la spécification, la conception, l'implémentation et l'évaluation de mécanismes de gestion de la QoS. La deuxième contribution consiste en la spécification et la conception d'une architecture logicielle pour la gestion auto-adaptative de la QoS au niveau Middleware suivant l'un ou l'autre des standards SmartM2M et oneM2M. Cette architecture, intitulé IoT-Q, repose sur l'application de *politiques hiérarchiques* dans l'élaboration des actions d'adaptation, ainsi que sur le paradigme de l'*Autonomic Computing*. La troisième contribution consiste en la proposition et la validation d'un modèle basé sur la théorie des files d'attente du Middleware OM2M (implémentation open source du standard SmartM2M), permettant d'estimer les performances des entités Middleware impliquées. Son application à la phase de monitoring est étudiée sous deux approches, *réactive* et *proactive*, en couplage avec des techniques de CEP (*Complex Event Processing*) pour la première approche, et d'un modèle de prédiction ARMA (*Auto-Regressive Moving Average*) pour la deuxième. La quatrième proposition porte sur la planification des mécanismes orientés ressources, que nous proposons de guider par une approche couplant le modèle analytique avec un modèle du système à base de graphes. Ce modèle permet de représenter et de prendre en compte les configurations possibles des entités impliquées et de leurs interactions. Des règles d'appariement et de réécriture de graphes permettant d'assurer la conformité du système avec le style architectural sont également produites. En cinquième contribution, un cas d'étude portant sur une situation de crise dans un environnement de transport urbain est élaboré pour illustrer l'application de ces derniers modèles.

Mots clés : IoT, Middleware, QoS, Scalabilité, Autonomic Computing, Théorie des files d'attente, Théorie des graphes, réécriture de graphe, Modèles de prédiction, Cloud Computing.

Abstract

The Internet is experiencing a drastic expansion again. In addition to conventional terminals, it now allows to interconnect all kinds of *connected objects* allowing the capture of events from the considered environment, but also the remote control of this environment. Billions of these objects are thus led in 2020 to contribute to the advent of the *Internet of Things* (IoT). This paradigm, which extends the *Machine-to-Machine* (M2M) concept, paves the way for new uses such as home automation, remote monitoring, or even the factories of the future.

Several architectures have been proposed to structure the IoT. Their foundation is based on a vision in four levels: (1) *Equipment* level, which includes the IoT equipment, (2) *Network* level containing the various technologies for data exchanges, (3) *Middleware* level, which offers applications an abstraction layer for underlying levels, and finally, (4) *Application* level, which consists of the set of applications contributing, via their interactions with the connected objects, to the realization of a business activity. Our work is positioned at Middleware level and is based on the architecture defined in the SmartM2M and then oneM2M standards. Several challenges have to be (re)considered in this context. We are mainly interested in the Quality of Service (QoS) issue expressed by some business applications.

Proposals addressing this issue essentially target the Network level. For the Middleware level, standards focus mainly on the proposal of architectures and functional services. The non-functional requirements, typically QoS, are little or not considered. Meanwhile, proprietary solutions do not consider the dynamic evolution of the context and requirements. In response to these limitations, we propose a dynamic management approach, i.e. during the execution of the system, and autonomous, i.e. without human intervention. The proposed management, guided by models, focuses on behavioural and structural reconfiguration actions related to application traffic and/or Middleware resources.

The first contribution of this thesis deals with the specification, design, implementation and evaluation of QoS management mechanisms. The second contribution is the specification and design of a software architecture for the self-adaptive QoS management at the Middleware level according to one of the SmartM2M and oneM2M standards. This architecture, called IoT-Q, is based on the application of *hierarchical policies* in the development of adaptation actions, as well as on the paradigm of Autonomic Computing. The third contribution proposes and validates an analytical model, based on the queuing theory, for the OM2M Middleware (open source implementation of the SmartM2M standard). This model allows to estimate the performance of the involved Middleware entities. The application of this model to the monitoring phase is studied under two approaches, reactive and proactive, coupled with Complex Event Processing (CEP) techniques for the first approach, and a ARMA (Auto -Regressive Moving Average) prediction model for the second one.

The fourth contribution concerns the planning of resources-oriented mechanisms that we propose to guide following an approach coupling the analytical model with a model of the system based on graphs. This model allows to represent and consider the possible configurations of the involved entities and their interactions. The rules of graphs matching and rewriting that ensure the conformity of the system with the architectural style are also produced. As a fifth contribution, a case study dealing with a crisis situation in an urban transport environment is also proposed to illustrate the application of the proposed models based on graphs.

Keywords: IoT, Middleware, QoS, scalability, autonomic computing, queueing theory, graphs theory, graph rewriting, prediction models, cloud computing.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Contexte et Problématique

Les prémices de l'Internet datent des années 1960 [kle61]. Basiquement, il permet d'interconnecter des réseaux de machines (ordinateurs et autres) à des fins de communication dans le cadre de logiciels distribués entre ces machines. De par l'évolution de ses différents piliers (infrastructure, protocoles, applications, usages, etc.), l'Internet constitue l'une des révolutions technologiques majeures de ces dernières décennies. Différentes voies d'évolution de l'Internet sont aujourd'hui considérées. La plus récente est relative à son expansion au monde physique à travers tous les types possibles d'équipements connectés, c'est-à-dire dotés de moyens de communication informatique. Plusieurs paradigmes ont ainsi vu le jour en fonction des périmètres couverts, des parties prenantes et des scénarios métiers considérés. Actuellement, le paradigme de l'Internet des objets (IoT - *Internet of Things*) en constitue l'un des plus importants.

Ce paradigme, qui étend le concept de M2M (*Machine-to-Machine*), permet d'interconnecter toute sorte d'équipements de la vie quotidienne, pour permettre la capture à distance d'événements (température, humidité, présence, etc.) via des *capteurs*, mais également le contrôle de l'environnement via des *actionneurs*. Ce paradigme ouvre la voie à de nouveaux usages tels que la domotique, la télésurveillance des biens et des patients, ou encore le suivi des états des machines au travers du concept d'usine du futur. L'utilisation de l'IoT dans ces contextes est amenée à apporter une véritable valeur ajoutée tant du point de vue du consommateur que du producteur de service.

Plusieurs visions architecturales sont proposées pour la structuration de l'IoT. Celle que nous retenons pour notre contexte est constituée de quatre niveaux. Le premier est le niveau *Équipements* qui est constitué de l'ensemble des équipements IoT (capteurs, actionneurs, tags, etc.). Il est suivi du niveau *Réseau* qui comporte l'ensemble des technologies réseaux d'interconnexion, nécessaires aux différentes interactions. Le niveau *Intergiciel* (*Middleware* en anglais*)* qui offre aux applications une couche d'abstraction pour, notamment, faciliter leur interaction avec les niveaux sous-jacents. Enfin, le niveau *Application* qui consiste en l'ensemble des applications logicielles concourant, via leurs interactions avec les objets connectés, à la réalisation d'une activité métier.

Plusieurs approches permettent d'implémenter le Middleware. Nous retrouvons par exemple des approches telles que RPC (*Remote Procedure Call*), SOA (*Service Oriented Architecture*) ou plus récemment ROA (*Resource Oriented Architecture*), cette dernière approche étant déclinée au travers des standards SmartM2M [ets690] et oneM2M [one15], dont plusieurs implémentations open source existent, notamment la plateforme OM2M [ben14]. Dans la suite, nous utiliserons le terme *entité Middleware* pour désigner toute instance logicielle de niveau Middleware.

Les spécificités de l'IoT conduisent à la reconsidération de nombreuses problématiques déjà abordées dans d'autres contextes *plus* traditionnels. Dans cette thèse, nous nous intéressons à la qualité de service (QoS - *Quality of Service*). Dans le domaine des télécommunications, l'ITU-T définit la QoS [itu08] comme l'ensemble des caractéristiques d'un service de télécommunication conduisant à sa capacité à satisfaire les besoins indiqués et implicites de l'utilisateur du service. Dans le contexte de l'IoT, elle se réfère à la capacité du système IoT et ses différentes couches à garantir les besoins non fonctionnels correspondants aux exigences des applications métiers.

Si le problème de la QoS a été largement traité dans l'Internet classique, il doit être reconsidéré pour l'IoT et ses applications. En effet, en fonction du scénario métier, les applications IoT peuvent présenter plusieurs profils définis en termes de types de données (brutes, audio, vidéo ou image), de type d'interactions (requête / réponse, publication / souscription, etc.) et de besoins en QoS qui peuvent évoluer de façon dynamique, c'est-à-dire durant l'exécution de l'application. Ces besoins peuvent être exprimés en termes de temps de réponse de bout en bout, de taux de pertes, de disponibilité, etc.

Concernant le niveau Equipements, plusieurs approches ont été proposées pour répondre aux besoins de QoS ; elles reposent notamment sur un principe d'adaptation (ou reconfiguration) des équipements en fonction du besoin [ala16, pan13]. Au niveau Réseau, plusieurs approches existent déjà pour la gestion de la QoS dans l'Internet. Ces approches sont notamment inspirées des propositions IntServ [rfc1633] et DiffServ [rfc2475], ainsi que des travaux réalisés autour des protocoles de Transport [ame94, van13, oul15].

En revanche, au niveau Middleware pour l'IoT, la majorité des propositions ont été focalisées sur l'aspect fonctionnel du Middleware via la proposition de cadres architecturaux et d'implémentations. Les travaux traitant la gestion de la QoS restent cependant insuffisants. La majorité d'entre eux ne s'attachent qu'au besoin en configuration des réseaux et des équipements, mais ne considèrent pas le Middleware en tant que partie prenante, alors que celui-ci peut également engendrer une dégradation de la QoS au niveau Middleware. Dans d'autres contextes que l'IoT, la plupart des solutions de gestion de la QoS proposent des approches qualifiables de *statiques*, en ceci qu'elle ne permettant pas de prendre en compte l'évolution des besoins en QoS durant l'exécution du système. Cependant, le contexte IoT est fondamentalement *dynamique* en ceci que les équipements, les réseaux, les types d'interactions, les types de données et les besoins en QoS peuvent varier au cours de l'exécution du système. Le besoin d'un système pour l'*adaptation dynamique* du Middleware en fonction des besoins en QoS de l'application se présente donc. En outre, pour un administrateur humain, la prise en compte de cette dynamicité peut s'avérer difficile, voire impossible. Il s'agit donc autant que possible de doter le système de capacité d'*auto-adaptation*, lui permettant de s'autogérer en minimisant (et idéalement supprimant) l'intervention humaine.

Contributions

Dans cette thèse, nous proposons un système de gestion de la QoS de bout en bout au niveau Middleware de l'IoT que nous intitulons IoT-Q. La gestion visée se traduit par la configuration et l'activation, lorsque c'est nécessaire, de mécanismes ayant un impact sur la QoS des applications. Dans notre approche, cette gestion est assurée de manière dynamique et auto-adaptative suivant le paradigme de l'*Autonomic Computing* proposé par IBM [kep03]. Ce paradigme est basé sur le principe d'une boucle dite *MAPE-K* enchaînant successivement des phases de supervision (M - *Monitoring*), d'analyse (A - *Analysis*), de planification (P - *Planning*) et d'exécution (E - *Execution*) pour aboutir à l'autogestion du système, sur la base

de règles, des modèles et/ou d'algorithmes maintenus dans une base de connaissance (K - *Knowledge Base*).

Les contributions proposées dans cette thèse portent globalement sur la spécification, la conception et l'implémentation de l'architecture structurelle et comportementale du système IoT-Q. Ces contributions se déclinent comme suit :

1. Nous proposons tout d'abord des mécanismes de gestion de la QoS amenés à constituer les actions d'adaptation du Middleware. Ces mécanismes se divisent en deux familles. La première famille de mécanismes, *orientée trafic*, est inspirée des approches utilisées aux niveaux Réseau et/ou Transport de l'architecture TCP/IP ; elle repose sur des techniques de marquage, de rejet, de retardement et d'ordonnancement du trafic en fonction de sa priorité. La deuxième famille de mécanismes, *orientée ressources,* est inspirée des solutions envisageables dans des environnements de virtualisation de type Cloud Computing, portant notamment sur les ressources informatiques dont bénéficient les entités Middleware.

2. Nous proposons ensuite notre architecture IoT-Q pour une gestion de la QoS de bout en bout au niveau Middleware, qui intègre et étend les spécifications des standards SmartM2M et oneM2M. L'architecture IoT-Q est basée sur une approche de gestion hiérarchique inspirée, notamment, de certaines approches pour la gestion des réseaux [fle01]. Elle repose sur deux niveaux de politiques de gestion appliquées au Middleware. Le premier niveau consiste en des *actions opérationnelles* basées sur les mécanismes précédents pour conduire l'adaptation dynamique du comportement et/ou des ressources des entités Middleware, ceci dans le but d'atteindre un objectif de QoS local à l'entité considérée. Le deuxième niveau porte sur des *actions stratégiques* conduisant à l'adaptation dynamique des objectifs de QoS locaux à atteindre par chacune des entités Middleware afin de répondre à l'objectif de bout en bout de la QoS requise par l'application. Chaque niveau repose sur l'utilisation du paradigme de l'Autonomic Computing pour élaborer la gestion de manière auto-adaptative.

3. Notre troisième classe de contribution porte sur la proposition de modèles pour guider la gestion *locale* de la QoS (c'est-à-dire, vis-à-vis de l'objectif de QoS locale fixé pour chaque entité Middleware). Ces modèles fournissent des représentations des différents éléments du système géré. Les composants de gestion se basent ainsi sur ces représentations afin de réduire les interactions avec le système géré. De façon synthétique, ces modèles sont les suivants :

    - basé sur la théorie des files d'attente, le premier modèle porte sur les entités Middleware de la plateforme OM2M basée sur le standard SmartM2M. Ce modèle permet de donner une représentation mathématique des métriques de performance, notamment le temps de réponse, en fonction de paramètres d'entrée tels que le taux de requêtes / réponses amenées à transiter par l'entité Middleware considérée ;
    - le deuxième modèle est basé sur le modèle de prédiction ARMA. Ce modèle permet de comprendre et éventuellement de prédire les valeurs futures en se basant sur les données actuelles et passées. Dans notre cas, le modèle est dédié à la représentation du comportement du trafic en terme de taux d'arrivée dans un environnement de déploiement réel ainsi qu'à la prédiction des arrivées futures des requêtes vers l'entité Middleware ;
    - un troisième modèle proposé repose sur le formalisme des graphes et l'utilisation des règles d'appariement et de réécriture de graphes. Ce modèle permet de représenter et

de prendre en compte les configurations possibles des entités impliquées et de leurs interactions, tout en garantissant la conformité avec le style architectural du système.

4. Nous proposons également l'application des modèles précédents à certains des composants de gestion de la QoS impliqués dans la gestion autonomique de la QoS locale, typiquement les composants de *monitoring* et de *planification* :

   - nous proposons deux approches de gestion du monitoring : *réactive* et *proactive*. La première approche vise à détecter les dégradations effectives de la QoS. Elle se base sur un ensemble de règles de supervision élaborées en couplant le modèle analytique de l'entité Middleware avec des techniques de traitement d'événements complexes (CEP - *Complex Events Processing*). La deuxième approche, proactive, vise à prédire la dégradation de la QoS. Elle étend les règles précédentes en y intégrant le modèle du trafic à des fins de prédiction de son taux d'arrivée ;
   - nous proposons également de guider la planification des mécanismes orientés ressources par une approche couplant le modèle à base de graphes pour garantir le respect du style architectural et le modèle analytique pour vérifier la disponibilité des ressources informatiques.

5. Finalement, un cas d'étude illustratif portant sur la supervision d'un réseau de lignes de métro et de gestion de la foule en cas d'incident est proposé. L'objectif de ce cas d'étude est d'illustrer la contribution portant sur le composant de planification pour la gestion de la QoS.

Structure du manuscrit

Ce manuscrit est divisé en cinq chapitres structurés comme suit :

Le **chapitre 1** présente l'état de l'art, la problématique spécifique et l'approche générale de notre contribution incluant son positionnement. Après avoir présenté le paradigme de l'IoT et ses apports attendus, un état de l'art est établi sur les solutions Middleware, dans l'IoT et hors IoT, pour la gestion de la QoS. Cet état de l'art permet à la fois d'identifier les limites des solutions existantes mais également de s'en inspirer et les étendre dans le cadre de notre approche. Nous précisons par la suite notre problématique spécifique ainsi que notre approche générale de gestion auto-adaptative (dynamique et autonome) de mécanismes orientés QoS pour le Middleware de l'IoT.

Le **chapitre 2** détaille notre proposition de mécanismes pour la gestion de la QoS au niveau des entités Middleware. Il présente tout d'abord les sources de trafic IoT. Les mécanismes orientés trafic, ainsi que les scénarios d'évaluation de performances permettant de valider leur pertinence, sont présentés dans un second temps. Cette validation est basée sur une plateforme d'émulation de trafic. Enfin, les mécanismes orientés ressources sont également présentés et validés via la même plateforme d'émulation et dans un environnement virtualisé.

Le **chapitre 3** décrit la spécification et la conception de l'architecture de gestion dynamique et autonomique de la QoS de bout en bout (architecture IoT-Q). Il présente tout d'abord l'approche d'élaboration de l'architecture basée sur le paradigme de l'Autonomic Computing et sur le principe de politiques hiérarchiques. Ensuite, la méthodologie d'élaboration, basée sur le concept de MDA [sol00], ainsi que les besoins fonctionnels et non fonctionnels du système. Cette méthodologie amène également à identifier les cas d'utilisation, les diagrammes de structure composite de l'architecture et de ses principaux composants, ainsi que les interactions entre ces composants, suivant le formalisme UML 2.0

Le **chapitre 4** présente le modèle analytique de l'entité Middleware basé sur la théorie des files d'attente et le modèle du trafic basé sur le processus ARMA. Pour chaque modèle, nous détaillons la démarche adoptée pour son élaboration et sa validation. Nous proposons également l'application de ces modèles à la phase de supervision via le composant de monitoring pour la gestion locale de la QoS. A cette fin sont ainsi présentés les niveaux de supervision, les métriques collectées, la structure interne du composant de monitoring et les approches de supervision considérées (réactive et proactive). Pour chacune de ces approches, nous donnons une représentation à titre illustratif des règles de supervision (pattern) combinant les modèles précédents et les techniques de CEP, et permettant de lever des symptômes de dégradation de la QoS.

Le **chapitre 5** est consacré à l'étape de planification des mécanismes opérationnels mis en œuvre au niveau des entités Middleware pour satisfaire un objectif de QoS local, et à la valorisation de l'approche proposée via un cas d'étude de gestion de la foule dans un réseau de lignes de métro. Ce chapitre présente tout d'abord notre approche d'adaptation dynamique guidée par les modèles. Ensuite le cas d'étude envisagé en deux grandes phases : une de supervision et une d'intervention. Pour chacune des phases, les entités physiques sont ensuite modélisées sur deux niveaux d'abstraction (ou points de vue) exprimant leurs rôles dans le cadre du scénario de gestion, ainsi que les entités logicielles les implémentant. Pour chacun des niveaux, nous donnons la description des styles architecturaux caractérisant les contraintes et les caractéristiques communes des configurations acceptables en termes d'entités et d'interactions. Enfin, le chapitre propose des règles de reconfiguration pour la gestion dynamique issues de la composition du modèle analytique de l'entité Middleware et des règles de réécriture de graphes.

Le manuscrit expose finalement les conclusions relatives à notre travail et les perspectives attenantes aux contributions à court, moyen et long termes.

Chapitre 1

Etat de l'art, problématique, positionnement et approche générale

Contenu

1.1. INTRODUCTION

De par la largeur de son spectre et les attendus qu'il suscite, l'IoT fait actuellement l'objet de nombreuses études de recherche, relevant tant de ses applications que de la reconsidération de problématiques initialement abordées dans l'Internet "classique". Dans ce cadre, l'objectif final de ce chapitre est de présenter la problématique de nos travaux et d'y positionner notre approche générale de solution. Pour cela, nous exposons tout d'abord le contexte de nos travaux, à savoir l'IoT, ses applications et leurs besoins en QoS, ainsi que les limites actuelles de l'IoT à y faire face. Nous dressons ensuite un état de l'art des solutions envisagées au niveau Middleware pour l'IoT. Nous fournissons en particulier une analyse des capacités et des limites de ces Middlewares face aux besoins en QoS. Nous introduisons également les propositions de

gestion de la QoS faites au travers de Middleware dédiés à d'autres domaines que l'IoT. Enfin, nous précisons le contexte et la problématique spécifiques de nos travaux et nous y positionnons notre approche pour une gestion de la QoS de bout en bout au niveau Middleware de l'IoT.

La suite de ce chapitre est structurée comme suit. La section 1.2 présente le paradigme de l'IoT selon les visions qu'en donnent les principaux organismes de standardisation. Elle donne ensuite un exemple de trois applications métier dans l'IoT à savoir le smart metering, l'eHealth et le smart factory pour illustrer l'apport de l'IoT. La section se termine par la présentation des visions architecturales de l'IoT, dont celle que nous retenons dans nos travaux, ainsi que les challenges majeurs relatifs à l'IoT. Cette thèse adressant la problématique de la QoS dans l'IoT, la section 1.3 décrit les différents types de données, d'interactions et de besoins en QoS inhérents aux applications de l'IoT. Nous nous focalisons ensuite sur le niveau Middleware. La section 1.4 expose tout d'abord les différentes approches de style architectural qui permettent d'implémenter un Middleware. Ensuite, elle dresse un état de l'art des propositions existantes au niveau Middleware issues des efforts d'organismes de standardisation et d'alliances. Elle donne enfin les différentes propositions de gestion de la QoS au niveau Middleware dans le contexte de l'IoT et en dehors de ce contexte. Face aux limites de l'état de l'art vis-à-vis du besoin en gestion de la QoS au niveau Middleware, la section 1.5 détaille la problématique spécifique que nous adressons, ainsi que le positionnement et l'approche générale que nous proposons. Finalement, nous terminons le chapitre par les conclusions attenantes en section 1.6.

1.2. PARADIGME DE L'INTERNET DES OBJETS

Le qualificatif d'*Internet des Objets* (IoT) est apparu pour la première fois en 1999 [ash09]. Ce paradigme est considéré comme étant la prochaine génération de l'Internet [atz10, alf15]. Plusieurs définitions ont été données à l'IoT. Gartner [gar17] le définit comme étant "*le réseau d'objets physiques qui contiennent des technologies intégrées pour communiquer et détecter ou interagir avec leurs états internes ou l'environnement externe*". L'IETF considère l'IoT comme étant une extension des technologies de l'actuel Internet et affirme qu'un véritable IoT exige que les objets doivent être capables d'utiliser les protocoles de l'Internet [ish13]. Une définition plus détaillée a été donnée par l'IERC (European Research Cluster on the Internet of Things) [ierc14] dans laquelle l'IoT est "*une infrastructure de réseau globale dynamique avec des capacités d'auto-configuration, où les objets physiques et virtuels ont des identités, des attributs physiques et des personnalités virtuelles, et utilisent des interfaces intelligentes pour se connecter entre elles et au réseau de données*". Ce paradigme permet de couvrir un large éventail d'applications dans plusieurs domaines tels que la santé, la domotique, les services publics, les transports, ou encore les usines du futur.

Dans cette section, nous illustrons trois applications *métier* dédiées à l'IoT, l'objectif étant d'illustrer la valeur ajoutée qu'apporte l'IoT aux différentes parties prenantes. Nous présentons enfin les différentes visions architecturales pour l'IoT ainsi que ses principaux challenges.

1.2.1. Applications métier de l'IoT

Une application métier dédiée à l'IoT (*application IoT*) se caractérise par l'*autonomie* ajoutée à une activité métier, telle que la surveillance de patients à distance et l'intervention en cas d'urgence, à travers l'utilisation d'équipements de toutes sortes accessibles via l'Internet. L'autonomie qu'offre ces applications est destinée à permettre, par exemple, des économies de

temps ou à réduire les risques d'erreurs, en minimisant autant que possible l'intervention humaine. Ces applications sont basées sur un ensemble d'échanges réalisés de manière essentiellement autonome entre les équipements, les entités intermédiaires et au final les humains via des moyens de communication, de nouvelles capacités de traitement de données offertes par le cloud computing, et des outils de gestion et d'analyse de ces données.

Les applications IoT couvrent un large éventail de domaines. Elles sont qualifiées *d'intelligentes* (*smart* en anglais) par l'autonomie du service qu'elles fournissent aux utilisateurs finaux. Nous parlons à présent de Smart City, Smart Home, eHealth, Smart Energy, etc. Dans cette section, nous étudions trois catégories d'applications IoT, relevant respectivement du Smart Metering, du eHealth et du Smart Factory, dans le but de montrer la valeur ajoutée de l'utilisation de l'IoT du point de vue du consommateur et du fournisseur de services.

Smart Metering

L'économie d'énergie est une priorité mondiale pour la préservation de notre planète. Le contrôle de la consommation énergétique permet d'aller vers cet objectif. L'une des applications clés du Smart Grid est la mesure intelligente. Elle est basée sur des compteurs dits intelligents offrant une gamme de services tels que la mesure et l'enregistrement de la consommation (électricité, gaz, eau, etc.), le relevé de la consommation locale ou distant, la fixation du seuil de consommation maximale et l'extinction à distance de l'électricité par le consommateur, ainsi que l'échange d'informations sur la consommation avec les services publics.

Les compteurs intelligents sont considérés comme les successeurs des compteurs d'électricité mécaniques classiques dits muets. La différence est que, pour les compteurs intelligents, il n'y a pas besoin d'intervention humaine (Figure 1.1). Toutes les fonctionnalités se font automatiquement et à distance. D'ici fin 2020, le ministère de l'énergie et du changement climatique du gouvernement britannique prévoit d'équiper chaque maison du royaume avec des compteurs intelligents [uk17].

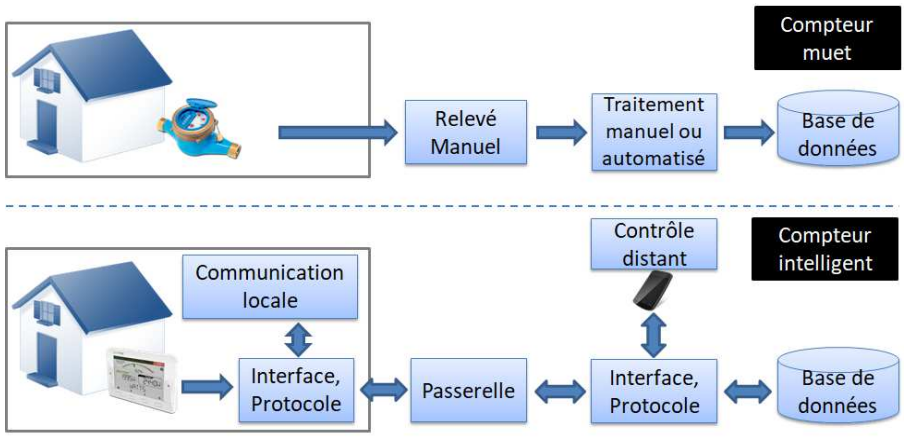

Figure 1.1 - Différence entre le compteur traditionnel et intelligent ([ger06])

Au regard des avantages qu'offre une telle application, le consommateur et le fournisseur de services sont bénéficiaires. Le consommateur de service (le client final) a l'opportunité de réaliser des économies d'argent en ayant différentes vues de sa consommation, par exemple, la consommation en temps réel et son équivalent en euro, l'historique de la consommation et la visualisation des graphiques connexes, le coût quotidien, mensuel ou annuel, ainsi que sa capacité à contrôler à distance l'utilisation de ses ressources. Du point de vue du fournisseur de services (l'opérateur électrique), les coûts d'acquisition de données sont minimisés. Les agents,

dont le rôle était auparavant dédié au déplacement pour l'acquisition manuelle de la consommation, peuvent maintenant être affectés à d'autres tâches. En outre, le fournisseur de services peut déterminer le comportement des utilisateurs liés à leur consommation énergétique pour une meilleure gestion des ressources. Notons que les questions liées à la protection de la vie privée sont sujettes à de multiples études qui ne sont introduites dans ce manuscrit.

eHealth

Le monde de la médecine (ou en général la santé) est en constante progression. Il commence à intégrer les méthodes et les ressources des technologies de l'information et de la communication. L'avènement des technologies de l'IoT permet d'envisager un saut substantiel. L'eHealth est l'un des sous-domaines les plus prometteurs de la télémédecine basée sur l'IoT. La Commission européenne [ehe17] le définit comme l'utilisation de technologies d'information et de communication modernes pour répondre aux besoins des citoyens, des patients, des professionnels de la santé, des fournisseurs de soins de santé, ainsi que des décideurs. Les applications IoT pour l'eHealth permettent [ets102] la surveillance à distance des informations sur la santé et la condition physique des patients, le déclenchement d'alarmes dans des conditions critiques et, dans certains cas, la maîtrise à distance de certains traitements ou paramètres médicaux.

Dans le passé, il n'y avait pas de surveillance à distance. Le patient devait être présent à l'hôpital, et dépendait des médecins et des équipements hospitaliers pour suivre et enregistrer les informations sur sa santé. Maintenant, les soins aux patients sont améliorés avec la surveillance à distance (Figure 1.2). La personne surveillée utilise des capteurs physiologiques portables formant un réseau BAN (*Body Area Network*). Ces capteurs recueillent et enregistrent différents signes vitaux tels que la fréquence cardiaque, la température corporelle, la pression artérielle, le taux respiratoire, les chutes et d'autres informations sur le patient. Les données collectées sont envoyées à un agrégateur (par exemple, le téléphone cellulaire du patient) qui est responsable de la transmission d'informations au centre de contrôle [wu11]. Grâce à ce système, les agents de santé peuvent exécuter des plans prédéfinis et intervenir lorsque des conditions critiques sont détectées en fonction des alarmes déclenchées.

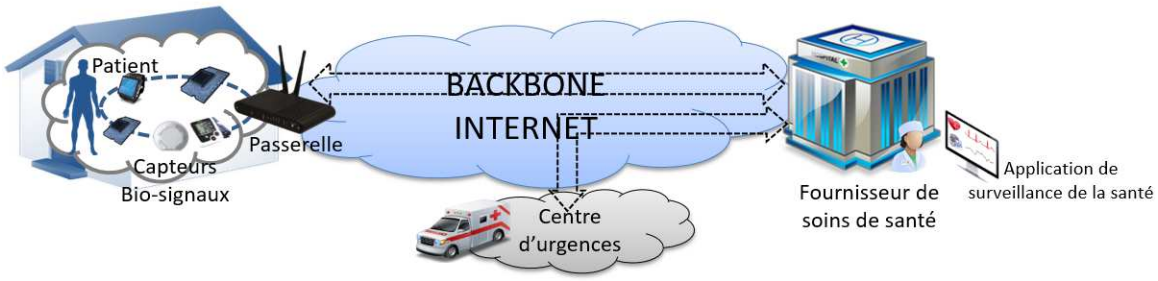

Figure 1.2 - Exemple d'architecture de télésurveillance de patients

De cette façon, plusieurs avantages peuvent être énoncés. Du point de vue du consommateur (ici, le patient), il y aura une réduction du temps et des risques. Le patient peut rester et guérir à la maison tout en bénéficiant d'une supervision et d'une assistance en temps réel. Pour le fournisseur de services (ici, le centre hospitalier), il y aura une réduction considérable des coûts grâce à la diminution des services nécessaires. Les hôpitaux seront en outre en mesure de traiter plus de patients sans avoir besoin de plus de lits et de personnels.

Smart Factory

Le temps est une clé importante dans le contexte industriel. Tout temps d'arrêt signifie perte de revenus. Les usines recherchent toujours l'amélioration de l'efficacité de la production grâce à la réduction de coûts et de temps. L'intervention des technologies de l'IoT dans le processus de fabrication dans les usines peut apporter une vraie valeur ajoutée ; nous parlons ainsi d'*usines intelligentes (*ou *smart factories)*. Dans [wey14], l'usine intelligente se réfère à la connexion des machines, des dispositifs et de la logistique, et au final des humains pour effectuer la coordination nécessaire de manière omniprésente et ponctuelle. L'usine sera "intelligente" dans différents processus. Elle pourra suivre les produits qui sont dotés d'étiquettes RFID sur la chaîne d'approvisionnement mondiale. Elle sera capable d'obtenir la charge et l'état de chaque machine, et de détecter la panne de la machine et la changer rapidement et pourquoi pas d'une manière autonome sans aucune intervention humaine. Le coût et la qualité de production seront optimisés en utilisant des capteurs intelligents et des actionneurs permettant d'avoir plus de souplesse, une adaptation rapide aux exigences changeantes et une gestion plus efficace des ressources.

Du point de vue consommateur de services (le client), les bénéfices induits par l'usine intelligente se traduisent par la possibilité de suivre l'état d'avancement de sa commande tout au long du processus de fabrication et de livraison, et aussi la possibilité de personnaliser sa commande qui sera prise en compte directement par le fabricant de la machine. Pour le producteur (usine), le coût de gestion et d'intervention en cas de panne sera plus bas grâce à la rapidité de détection de la panne. Il y aura aussi une optimisation du processus de fabrication grâce au suivi en temps réel et distant de l'état du produit, ainsi que la réduction du temps de fabrication en prenant rapidement en compte les changements d'exigences ou de processus.

1.2.2. Visions architecturales et challenges de l'IoT

Plusieurs visions architecturales ont été proposées pour l'IoT. L'architecture basique est constituée de trois niveaux : le niveau *Application* incluant les services métiers associés, le niveau *Réseau* qui inclut les différents types de réseaux, et enfin le niveau *Équipement* comportant l'ensemble des objets impliqués dans la capture d'événements et le contrôle depuis l'environnement (capteurs, actionneurs, etc.).

Dans [ida14], l'architecture proposée reprend le modèle précédent en l'étendant à quatre niveaux. Elle intègre un niveau additionnel entre les niveaux Application et Réseau appelé niveau de *Service de gestion*. Ce niveau inclut le traitement de l'information par l'analyse, le contrôle de sécurité, la modélisation des processus et la gestion des périphériques. La proposition [dua11] considère l'architecture de base à trois niveaux mais en subdivisant le niveau Application en deux sous-niveaux. Le premier est le sous-niveaux *Application* qui implémente l'application métier en se basant sur les informations fournies par les niveaux sous-jacents. Le deuxième est le sous-niveau *Service* qui intègre et stocke les informations depuis le réseau et réalise la gestion de ces informations, par exemple, l'analyse de données et la prise de décision.

Dans cette thèse, nous adoptons une vision architecturale considérant quatre niveaux (Figure 1.3) pour la séparation des différents rôles. Cette vision est proposée par le standard SmartM2M [ets690]. Elle considère les niveaux *Application IoT*, *Middleware*, *Réseau* et *Équipements*. Le rôle principal du niveau Middleware est d'assurer l'interopérabilité des applications et des objets en masquant les détails des technologies sous-jacentes aux applications [raz16]. Dans l'IoT, le Middleware abstrait l'hétérogénéité des réseaux et des équipements en fournissant une

représentation homogène facilitant leur manipulation par les applications IoT. Ce rôle peut étendu via la capacité du Middleware à fournir d'autres services tels que la gestion des informations (par exemple, l'agrégation, le traitement et l'analyse des données), la gestion des services (par exemple, la découverte des objets et la configuration des services), ou encore la gestion des utilisateurs et des équipements.

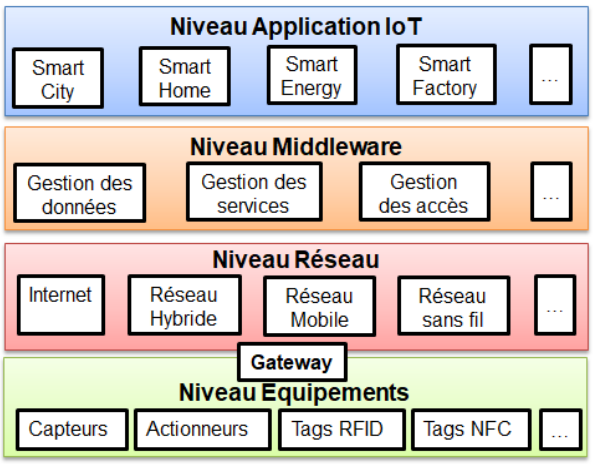

Figure 1.3 - Modèle architectural pour l'IoT

Sur les différents niveaux de l'architecture de l'IoT, plusieurs challenges traditionnels doivent être reconsidérés. En effet, la spécificité de ce paradigme, liée notamment à la multitude des technologies utilisées au niveau Réseau et Équipements, rend inadaptées ou sous efficaces les solutions proposées par ailleurs pour ces différents challenges. Nous citons trois challenges parmi les plus importants :

- **Fragmentation verticale et interopérabilité :** de nombreux industriels ont conçu et développé des solutions propriétaires dans le but d'amener de nouvelles applications IoT sur le marché. Cependant, cet état de fait a rapidement créé un problème de compatibilité entre les solutions des différents constructeurs, que ce soit au niveau architectural pour le lien entre les applications IoT et les équipements, ou au niveau interactionnel, où chaque application possède une API spécifique à la solution, créant ainsi un problème d'interopérabilité [alf15] entre les équipements et applications de différentes solutions ;
- **Sécurité et confidentialité :** l'IoT est caractérisé par l'utilisation d'équipements dits intelligents. Ces équipements génèrent des données à destination des applications ou reçoivent des commandes des applications pour contrôler leur environnement. En fonction du domaine applicatif, ces équipements doivent être de taille réduite et mobile, ceci à bas coût. Ces exigences peuvent rendre ces équipements ainsi que les réseaux vulnérables vis-à-vis des attaques externes. Ces attaques ciblent, par exemple, l'accès aux données, leur intégrité, le vol d'identité, ou le cambriolage [cha09], surtout lorsque les données communiquées sont susceptibles de contenir informations personnelles sur les utilisateurs et leurs habitudes [nin13]. De ce fait, de nouvelles techniques plus légères (pouvant être supporté par ces équipements) mais efficaces doivent être établies en adéquation avec le contexte de l'IoT ;
- ***Scalabilité* et Qualité de Service :** le terme de *scalabilité* désigne ici la capacité du système à s'adapter à la charge qu'il subit en maintenant les mêmes fonctionnalités et les mêmes performances indépendamment de cette charge. Dans l'IoT, le système peut avoir des problèmes de scalabilité vu la multitude des équipements et des applications qui échangent du trafic en passant par le cœur du système. La QoS requise par une

application se traduit par plusieurs métriques telles que le temps de réponse, le taux de pertes ou encore la disponibilité qu'il s'agit de maintenir à un certain niveau pour la bonne exécution de l'application. Dans un contexte IoT et en fonction du domaine ciblé, les applications métiers peuvent présenter des besoins en QoS plus ou moins critiques qui doivent être pris en considération par les niveaux sous-jacents. Par exemple, dans le domaine du eHealth, la supervision et l'intervention en urgence dans les plus brefs délais peut être primordiale face à la survie du patient. Nous parlons ainsi d'applications *critiques*.

Dans cette thèse, le focus est essentiellement mis sur le challenge de la QoS, Ce besoin en QoS est exprimé par les applications IoT critiques. Nous évoquerons dans certaines parties la scalabilité qui est un facteur impactant la gestion la QoS des besoins applicatifs.

Dans la section qui suit, nous présentons une caractérisation des applications métiers dans l'IoT. Elle concerne le type de données générées, le type d'interactions entre ces applications et les entités sous-jacentes du système, ainsi que les métriques de QoS que peuvent être exprimer ces applications.

1.3. APPLICATION MÉTIER DANS L'IOT ET BESOINS EN QoS

Dans les différents domaines, plusieurs familles d'applications IoT peuvent être considérées en fonction de leurs profils (types de données, types d'interactions et besoins en QoS) et de leurs rôles (supervision, intervention, etc.). Nous retrouvons par exemple les applications dédiées à la télésurveillance en temps réel, la génération d'alarme et l'intervention en cas de problème. De ces différents types d'applications IoT, plusieurs spécificités peuvent être extraites telles que le type de données générées, le type d'interactions entre les applications et les entités Middleware, ou encore les besoins en QoS qui reflètent la sensibilité de l'application vis-à-vis de métriques telles que le délai, les pertes, la disponibilité, etc. Dans cette section, nous présentons tout d'abord les différents types de données qui peuvent être générés dans le contexte de l'IoT. Nous présentons ensuite les types d'interactions entre les applications et les entités Middleware. Enfin, nous décrivons les besoins en QoS qui peuvent être exprimés par les applications IoT.

1.3.1. Types de données générées par les entités IoT

Dans un contexte IoT, les données sont échangées entre les différentes entités afin de mettre en place un service métier. Ces données peuvent être de différents types de par la diversité des équipements connectés (capteurs, actionneurs, caméras, microphones, cardiographe, etc.). Nous divisons les données échangées en deux types. Le premier concerne les données brutes issues d'équipements basiques (des relevés de température par exemple). Le deuxième type fait référence aux données multimédia (audio, vidéo) qui peuvent être issues d'équipements plus sophistiqués.

Les données brutes

Les données brutes sont transmises sous forme de texte simple transporté via un protocole de transfert. Dans un contexte IoT, ces données peuvent être issues d'équipements tels que des capteurs pour reporter par exemple la température, le degré d'humidité, la localisation, le niveau d'un gaz, ou des signaux bio pour le diagnostic médical (fréquence cardiaque,

respiration, etc.). Ces données sont véhiculées via des protocoles de transfert tels que HTTP [ets118] basé sur le protocole de transport TCP, CoAP [rfc7252] qui s'appuie sur UDP, ou encore MQTT [bg15] qui repose sur TCP. La description de ces données se fait, par exemple, via des langages de balisage tels que XML [rfc7303] ou JSON [cro09]. Ces données peuvent être simples telles qu'une mesure d'un capteur, ou composées de plusieurs valeurs simples afin de constituer une information utile telle que la pression artérielle. Les données brutes peuvent aussi être issues des applications IoT. Elles sont dédiées aux commandes et envoyées sous forme de requêtes par les applications et peuvent servir, par exemple, à envoyer une requête vers un équipement tel qu'un capteur pour récupérer son état, ou pour contrôler à distance un actionneur ou un moteur.

Les données multimédia

Les données multimédia sont plus riches. Elles peuvent, par exemple, être constituées d'image, d'audio, et/ou de vidéo. Les données de type Image [ets690] peuvent, par exemple, être utilisées dans le domaine de la télésurveillance de patients pour le visionnage et l'analyse par le médecin d'une blessure, d'un symptôme, ou pour la prise d'une radio à distance.

Le trafic Audio [sta10] permet d'échanger du son (de type voix ou autre) entre les entités impliquées. Ce type de données est très utile dans le cas d'intervention en urgence, par exemple, lors d'un diagnostic de l'état d'un patient en situation critique, ou lors d'un accident routier (projet eCall [ecall]). Un flux audio de type voix est véhiculé via des réseaux privés ou d'opérateurs (LTE, 3G, etc.) passant par l'Internet, par le biais de techniques de VoIP [boo10] ou AoIP [chu10]. Ces techniques peuvent être basées sur des protocoles propriétaires ou standardisés). Des exemples de protocoles VoIP sont en majorité représentés par la figure suivante et peuvent être H.323, MGCP, SIP, H.248 (Megaco), RTP, RTCP, SRTP, SDP, IAX, etc.

Concernant le trafic vidéo dans l'IoT, il peut essentiellement être généré à des objectifs de surveillance à distance [pet12]. Il peut s'agir par exemple de télésurveillance du domicile ou bien la téléconsultation d'un patient auprès d'un médecin. Le flux peut être fait transmis en *unicast* (de 1 en 1), en *multicast* (de 1 vers N ou de N vers M) ou en *broadcast* (de 1 vers tous). Il peut être basé sur les protocoles RTP, RTCP (standards de l'IETF), MMS (propriété de Microsoft), Adobe RTMP (propriété d'Adobe Systems) ou de nouveaux formats non propriétaires qui commencent à émerger tels que MPEG-DASH pour le streaming à débit adaptatif [saa11] sur HTTP.

1.3.2. Types d'interactions entre les entités IoT

La génération des données précédentes se fait suite à l'interaction entre les entités du système IoT. Plusieurs types d'interactions [ets690, nik13] peuvent avoir lieu entre ces entités en fonction de la source et de la destination de la requête, mais aussi des informations que contiennent les données échangées.

Un premier type d'interaction concernent les données des équipements. Ces requêtes peuvent émaner des équipements (capteurs ou actionneurs) pour la sauvegarde de leurs données l'IoT (liaison montante). Ces requêtes sont émises de façon périodique ou événementielle suite à un changement d'état (Figure 1.4 - p1). Elles peuvent aussi provenir de l'application en vue de la récupération d'une valeur de capteur ou de l'état d'un actionneur. Deux cas de figure se posent alors : (1) la requête est adressée au système IoT pour la récupération d'une donnée stockée

localement dans la base de données du Middleware (Figure 1.4 - p2), ou (2) la requête passe par le système IoT qui la redirige vers l'équipement concerné (Figure 1.4 – p3).

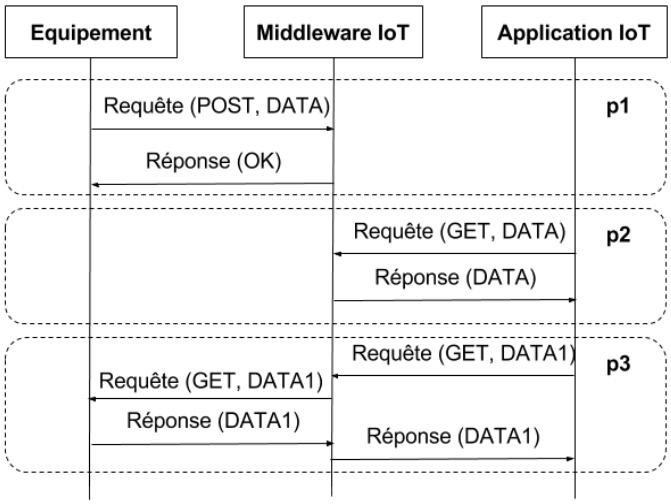

Figure 1.4 - Interactions de types requête / réponse pour le cas de données d'équipements

Un second type d'interaction se présente quand la requête provient d'une application pour le contrôle d'un équipement dans le cadre d'un scénario métier (Figure 1.5). La réponse confirmant la prise en compte de cette commande par l'équipement peut, par exemple, être son état courant (ON / OFF) s'il contrôle une lampe, un ventilateur, etc., ou une autre forme de flux de données si la requête est dédiée au déclenchement d'une caméra ou d'un micro.

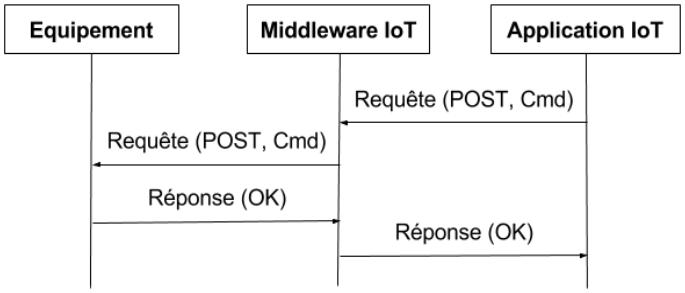

Figure 1.5 - Interaction pour le contrôle d'un actionneur

Un troisième type d'interaction concerne la signalisation (Figure 1.6). Une requête de signalisation provient soit de l'application soit du Middleware pour la réalisation d'une authentification, d'un enregistrement ou d'une mise à jour du micro-logiciel (firmware) d'un équipement.

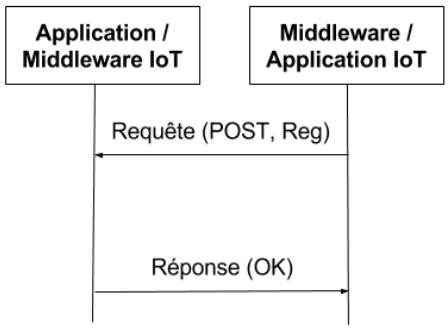

Figure 1.6 - Interaction de signalisation entre l'entité Middleware et l'Application IoT

Un autre type d'interaction entre l'application et le Middleware concerne la procédure de souscription / notification (Figure 1.7). Dans ce cas de figure, la souscription de l'application se fait sur toute nouvelle donnée issue d'un équipement. A la réception de cette donnée par le Middleware, ce dernier envoie une notification à l'application comportant la donnée qui vient d'être créée. Cette donnée peut, par exemple, être une donnée d'un capteur ou un nouvel état d'un actionneur.

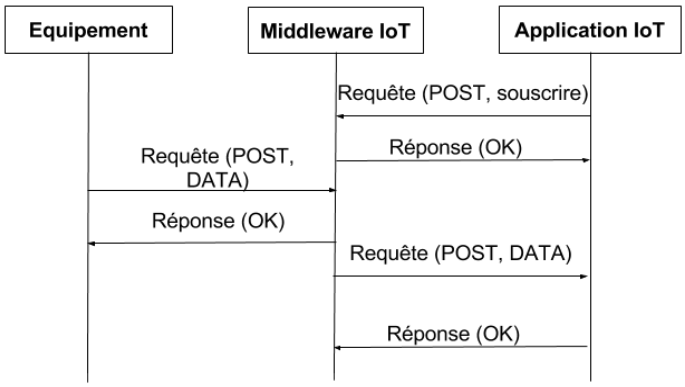

Figure 1.7 - Interaction de type souscription / notification entre les entités IoT

D'autres types d'interactions plus poussées existent. Nous retrouvons par exemple le *long-polling* [kat13]. Dans ce cas, l'application envoie une requête à laquelle le serveur répond par une notification quand la donnée devient disponible. Après la réception de la donnée, l'application doit envoyer une nouvelle requête de long-polling vers le serveur et attendre la réponse de ce dernier. Du fait que la requête de l'application ne peut pas rester indéfiniment pendante, le serveur envoie une réponse vide avant l'expiration du temps de vie de la requête. Cette réponse informe l'application du fait qu'il doit envoyer une nouvelle requête long-polling si elle veut continuer à recevoir les notifications. Il existe aussi le type *time-based pull* [kat13] où l'application envoie une requête de type pull pour une nouvelle donnée. Quand la donnée attendue devient disponible (par exemple, la mesure d'un capteur), le serveur envoie cette donnée à l'intérieur d'une réponse de notification. Sinon, il envoie une réponse vide. La fréquence des requêtes périodiques de l'application dépend de la définition de la fenêtre temporelle du *pull* (*pull time window*) qui doit être ajustée à la génération des données. Quand la valeur est trop petite, les requêtes de l'application peuvent arriver au serveur plus rapidement alors que les données ne sont pas encore prêtes (les ressources sont inutilement dépensées). Quand la fenêtre est trop large, l'application ne va pas acquérir toutes les données intéressantes.

1.3.3. Besoins en QoS des applications IoT

En fonction du contexte et du rôle d'une application IoT, celle-ci peut présenter un caractère critique et avoir des besoins en QoS. Ces besoins doivent être considérés et gérés par les niveaux sous-jacents du système IoT.

Parmi les besoins les plus importants, nous retrouvons [ieee90] :

- **la disponibilité :** définie comme le taux de disponibilité opérationnelle d'un système ou d'un composant lorsqu'il est requis pour utilisation. Pour les applications IoT critiques (par exemple, la surveillance des signes vitaux des soins de santé), la disponibilité est une exigence importante que l'on souhaite être très proche de 100% ;
- **la fiabilité :** se réfère à la capacité d'un système ou d'un composant à exécuter ses fonctions dans des conditions données et pour une période de temps spécifiée. Dans

l'IoT, les composants doivent être les plus fiables possible pour la surveillance continue et la gestion des requêtes ;

- **la priorité :** peut être considérée comme l'importance attribuée à une requête ou sa réponse par rapport à une autre. Le trafic de l'entité IoT ayant une priorité plus élevée doit être traité en priorité par le système, suivant une politique de gestion appropriée ;
- **le délai** : nous distinguons deux types de délai. Le premier est le délai d'aller-retour (RTT - *Round Trip Time*) qui correspond au temps nécessaire pour répondre à une demande. Il est considéré comme la somme du temps d'envoi de la demande et de la réponse, ainsi que du temps de traitement de différents composants parcourus dans le système IoT. Le deuxième type est le délai de bout en bout correspondant, par exemple, au transfert de données entre l'équipement et l'application IoT. Par exemple, [sko10] a fixé un délai maximal de 300 ms pour le transfert d'une électrocardiographie en temps réel ;
- **la périodicité :** définie en rapport avec l'intervalle temporel d'exécution des actions. Au-delà de cet intervalle, la donnée devient obsolète. La périodicité est une caractéristique spécifique des applications IoT de surveillance. Les données collectées peuvent avoir différentes périodes selon l'activité métier ;
- **l'intégrité des données :** garantit la non altération des données, que ce soit par accident ou par des parties non autorisées. Les données collectées provenant des équipements ne doivent pas être modifiées pour assurer la réalisation de l'action correcte par l'application IoT ;
- **la confidentialité :** liée à l'assurance que les données ou le système IoT ne peuvent être accessibles que par ceux qui en ont l'autorisation. Cette propriété est essentielle dans ces systèmes qui doivent garantir que les informations des équipements ou machines ne sont accessibles qu'aux parties autorisées.

Les applications IoT Micro ainsi être classifiées selon plusieurs critères. Le Tableau 1.1 illustre une classification de quelques applications IoT relatives aux domaines du eHealth et Smart Grid. Cette classification prend en considération le débit exigé, la sensibilité au temps, ainsi que d'autres besoins en QoS.

Domaine	**eHealth**			**Smart Grid**	
Application	**Télédiagnostic**	**Téléconsultation**	**Télé-monitoring**	**Smart metering**	**SCADA**
Type de données	brut et multimédia (image)	multimédia (audio et vidéo)	brut et multimédia (image et vidéo)	brut	brut
Temps réel	~	OUI	~	NON	OUI
Débit	Elevé	Elevé	~	Faible	Moyen
Périodicité	NON	NON	OUI	OUI	~
Temps de réponse	OUI	OUI	~	NON	OUI
Fiabilité	OUI	OUI	OUI	OUI	OUI
Disponibilité	OUI	OUI	OUI	NON	OUI
Priorité	~	OUI	~	NON	OUI
Deadline	~	OUI	~	NON	~

Tableau 1.1 - Classification des Applications IoT

Dans ce tableau, nous considérons deux domaines (eHealth et Smart Grid).

Nous illustrons trois applications du domaine du eHealth [sko10, gal05]. La première est une application de télédiagnostic véhiculant des données brutes (texte) ou multimédia (images)

issues de capteurs et d'équipements (ECG). Elle peut être temps réel en transférant des images médicales à une localisation distance dans des situations d'urgence, ou non temps réel dans le cas, par exemple, d'une simple analyse de données par des spécialistes. La deuxième application est une application de téléconsultation se fait en temps réel sur la base de conversation entre docteur / patient ou entre docteur / docteur. Enfin, la troisième application est une application de télé-monitoring peut être temps réel en cas de transmission des signes vitaux du patient et vidéo streaming en cas d'urgence, ou non temps réel quand la transmission des signes vitaux se fait du patient vers le centre de supervision pour une analyse.

Pour le domaine du Smart Grid, nous illustrons deux applications. La première application, de smart metering, véhicule des données brutes qui n'ont pas d'exigences de temps réel, de débit, de temps de réponse, de disponibilité de priorité ou de deadline. La deuxième application, SCADA (*Supervisory Control and Data Acquisition*), a plus d'exigence notamment en temps réel puisqu'elle doit assurer le contrôle à distance des installations techniques dans des environnements contraints tels que l'industrie. La fiabilité est une exigence pour toutes ces applications. Par rapport au débit des données, trois valeurs sont considérées : faible, moyen ou élevé. Il est considéré comme étant faible lorsqu'il est inférieur à 12 kbps, moyen s'il est entre 12 kbps et 24 kbps, et élevé quand il est supérieur à 24 kbps [wen12].

Les besoins applicatifs en terme de QoS peuvent être différents en fonction du contexte et du rôle joué par l'application métier (supervision, intervention, etc.). Afin d'assurer la prise en compte de ces besoins, ils doivent être considérés par les différents niveaux sous-jacents du système IoT. Dans cette thèse, nous traitons la gestion de la QoS au niveau Middleware, qui constitue le premier niveau pour la prise en compte de ces besoins. Nous abordons dans ce qui suit, les différentes solutions Middleware, ainsi que les propositions faites pour la gestion de la QoS à ces niveaux.

1.4. SOLUTIONS AU NIVEAU MIDDLEWARE ET GESTION DE LA QOS

Dans le contexte de l'IoT, le Middleware permet d'abstraire les applications de la complexité des niveaux sous-jacents. Cette complexité est due à la diversité et l'hétérogénéité des technologies réseaux et équipements. Au sein de l'IoT, les modes d'interaction peuvent suivre plusieurs *styles architecturaux* selon l'implémentation du Middleware. Dans cette section, nous détaillons tout d'abord ces différents styles architecturaux. Nous exposons ensuite les solutions les plus abouties issues d'organismes de standardisation, d'alliances ou de contributions isolées. Enfin, nous nous focalisons sur les propositions de gestion de la QoS au niveau Middleware dans le contexte de l'IoT et au-delà.

1.4.1. Styles architecturaux pour le Middleware

Un système, en particulier logiciel, est défini par son architecture. Celle-ci comporte des éléments structurels (ici des composants logiciels), leur comportement (i.e. l'algorithmique interne des composants), leurs interfaces, leur composition via des liens d'interconnexion, les objets échangés via ces liens, ainsi que le *style architectural* qui va régir cette organisation [sha96]. Un style architectural définit une famille de systèmes en termes d'organisation structurelle et de vocabulaire de composants et de connecteurs [fie00]. Dans cette section, nous définissons les principaux styles architecturaux envisagés pour l'implémentation du Middleware. Chacun d'eux est analysé sous l'angle de ses performances et de ses capacités à répondre au besoin en gestion de la QoS.

Dans cette section, nous définissons les différentes approches des styles architecturaux permettant d'implémenter le Middleware.

Remote Procedure Call

Dans les systèmes distribués, le modèle RPC (*Remote Procedure Call*) permet la communication entre processus distants [**rfc1057**]. Il permet à une application d'exécuter une procédure offerte par un serveur distant [bir84]. Le serveur est vu comme un ensemble de procédures exécutables via des appels par un client distant afin de réaliser une tâche spécifique [bir84]. Parmi les protocoles implémentant le modèle RPC, le plus utilisé est le système NFS (*Network File System*) développé par SUN Microsystems [rfc1094] qui fait partie de la couche application.

Etant donné que la logique métier est centralisée au niveau d'un serveur central, le modèle RPC présente des limites. En terme de QoS, elle résulte de celle offerte par les protocoles des couches sous-jacentes à savoir les couches Transport et Réseau. Le modèle RPC ajoute aussi une complexité supplémentaire en utilisant le protocole HTTP comme simple protocole de transfert sans exploiter les méthodes qu'il offre, alourdissant ainsi le corps de la requête en y incluant les appels de procédure [ric08] et en impactant ainsi les performances du système.

Approche orientée objets

Cette approche se base sur l'utilisation d'un ORB (*Object Request Broker*) afin de permettre l'interaction distante avec des objets déployés dans un même espace mémoire local [orb13]. Ces objets communiquent entres eux par l'envoi et la réception de requêtes et de réponses, de manière transparente et portable. Cette approche représente ainsi une extension de l'approche RPC en offrant davantage d'interopérabilité. Plusieurs implémentations de cette approche existent. Nous retrouvons par exemple parmi les plus connues : CORBA [cor92] qui est un standard défini par le groupe OMG pour les ORB, .NET Remoting [mcl02] pour les plateformes Microsoft, et RMI [rmi17] qui est une API Java développée par SUN Microsystems et qui permet l'invocation distante de méthodes Java.

Malgré l'interopérabilité et l'intégrabilité que permet d'offrir l'approche orientée objets, les besoins en QoS ne sont pas adressés par les solutions basiques. Les solutions plus avancées (telle que CORBA) supportent une certaine QoS en terme de tolérance aux fautes [emm00].

Approche orientée messages

Les Middlewares orientés messages (MOM - *Message-Oriented Middleware*) supportent l'envoi et la réception de messages entre systèmes distribués. Cette approche est basée sur le concept de message et de *canal* de communication qui peut être de 'un vers un', de 'un vers plusieurs', de 'plusieurs vers un', ou de 'plusieurs vers plusieurs', ainsi qu'un agent appelé *Message broker* [bro03]. Parmi les implémentations les plus utilisées du *Message broker*, nous retrouvons : Apache ActiveMQ [sny10], Fuse [fus12], RabbitMQ [rab07], ainsi que le protocole AMQP (*Advanced Message Queuing Protocol*) pour traiter les problématiques d'interopérabilité [amq12].

L'approche MOM réduit la complexité des systèmes distribués hétérogènes. En ce qui concerne la QoS, plusieurs implémentations telles que DDS (*Data Distribution Service for Real-Time Systems*) [dds04] ou RabbitMQ [rab07] permettent de la gérer via, par exemple, des

mécanismes de persistance de données pour assurer la disponibilité des messages et la fiabilité de leur transfert.

Approche orientée services

L'approche orientée services permet d'élaborer des architectures dites SOA (*Service Oriented Architecture*) [soa06] pour les systèmes distribués. Cette approche se base sur l'échange de données entre les entités de l'architecture en tant que *services web*. Les solutions basées SOA garantissent l'interopérabilité en se basant sur des protocoles tels que HTTP, JMS, SMTP ou FTP pour la communication entre des plateformes hétérogènes. Les bus de services ou ESB (*Enterprise Service Bus*) [esb02] constituent l'une des implémentations les plus connues de SOA. L'ESB joue le rôle de Middleware entre producteurs et consommateurs de services. Il permet l'échange asynchrone via des interfaces telles que SOAP, JMS ou JBI.

Les solutions orientées SOA sont toutefois limitées. Les protocoles de communication (par exemple HTTP) ne sont exploités que pour véhiculer les messages. Ces messages contiennent la représentation ainsi que la méthode invoquée, ce qui nécessite à chaque fois la désencapsulation du corps du message pour déduire la méthode. Il ne tire donc pas partie de la méthode utilisée par le protocole de communication et peut engendrer des problèmes de QoS.

Approche orientée ressources

L'approche orientée ressources est basée sur les principes du style architectural REST (*Representational State Transfer*) [fie00]. Ce style architectural considère que toute entité physique ou logique est une ressource ayant une représentation accessible à distance. Chaque ressource est adressable de manière unique via un URI (*Uniform Resource Identifier*). Les systèmes RESTful se basent essentiellement sur des protocoles de transfert tels que HTTP ou CoAP tout en exploitant leurs méthodes pour alléger le corps de la requête.

REST offre donc une grande interopérabilité en exploitant les méthodes du protocole de transfert (par exemple GET, POST, PUT, DELETE pour le protocole HTTP). Il est aussi sans état, ce qui le rend scalable puisque le serveur ne gère plus l'état des sessions des utilisateurs.

Pour pallier au problème de fragmentation verticale des solutions IoT, plusieurs solutions Middlewares ont été proposées que nous présentons à présent. Dans ce qui suit, le focus est plus spécifiquement porté sur les propositions issues des organismes de standardisation et des alliances.

1.4.2. Propositions de niveau Middleware dans l'IoT

Plusieurs solutions ont été proposées au niveau Middleware afin de palier au problème de fragmentation verticale de l'IoT. Nous regroupons ces propositions en deux familles : celles issues des organismes de standardisation qui proposent les cadres architecturaux, les composants fonctionnels et les interactions entre eux ; et celles basées sur des alliances qui proposent directement des plateformes Middleware open source.

Dans cette section, nous commençons tout d'abord par la première famille en présentant les standards SmartM2M et oneM2M. Ensuite, nous passons aux alliances avec la présentation des projets open source IoTivity et FIWARE.

Standard SmartM2M

L'ETSI (*European Telecommunication Standard Institute*) a proposé une première version du standard SmartM2M en 2011 [ets690]. Il a établi un standard complet comprenant l'architecture fonctionnelle, les interfaces de communication, la représentation et la structuration des données, ainsi que toutes les procédures nécessaires entre entités du système d'une part, et entre ces mêmes entités et les entités externes au système. La couche d'abstraction, appelée SCL (*Service Capability Layer*), propose un ensemble de services de niveau Middleware, tels que la communication générique, l'accessibilité, l'adressage et le stockage, la sélection du canal de communication, la gestion des entités (équipements, données, droits d'accès, etc.), la sécurité, etc. Cette couche de service peut être déployée sur un serveur de concentration (NSCL - *Network SCL*) avec lequel interagissent les applications métier, ou sur des gateways (GSCL - *Gateway SCL*) reliées au serveur et permettant l'interconnexion des équipements non capables de communiquer avec l'extérieur.

L'interaction entre les entités de l'architecture se fait via des interfaces de communication basées sur le style architectural REST. Il donne une représentation sous forme de ressources des différents services. Ces ressources sont adressables via une URI et gérables via les méthodes CRUD (*Create, Read, Update et Delete*).

Le standard donne une structuration de ces ressources sous forme d'une représentation arborescente appelée *arbre de ressources*. Cet arbre vise à offrir des fonctions de médiation de données en tant que moyen de simplifier l'adressage et la maintenance des ressources, décrire comment les différents types de ressources sont liés entre eux et améliorer la performance globale du système grâce à l'utilisation de données minimalement structurées. Ce style permet ainsi aux applications IoT d'interagir indépendamment du type de technologies réseau et d'équipements sous-jacents. Le standard propose également l'intégration de protocoles tels que HTTP et CoAP pour la communication distante entre les différentes entités du système.

Vis-à-vis des besoins de QoS, le standard SmartM2M n'intègre aucun service au niveau Middleware permettant de prendre en considération ces besoins. Il considère que la QoS est celle résultante de la couche sous-jacente (réseau).

Initiative oneM2M

L'initiative oneM2M [one15] s'inscrit dans une vision globale d'organismes de standardisation pour offrir un unique standard. Elle propose également une couche de services RESTful horizontale. Cette couche permet l'enregistrement mutuel entre les entités, la découverte des services et des ressources, la gestion des équipements, la sécurité, etc. Nous la considérons généralement comme le successeur de SmartM2M.

L'architecture oneM2M est composée de trois couches. La couche Application est constituée des entités applicatives (AE - *Application Entity*) qui représentent les applications interagissant avec le serveur, les gateways ou les équipements. La couche Service, appelée CSE (*Common Service Entity*), représente la couche d'abstraction Middleware. La couche Réseau englobe tous les réseaux de communication et équipements sous-jacents. Le standard repose aussi sur la notion de ressources et propose le même arbre de ressources que SmartM2M, avec intégration de protocoles de communication tels que HTTP, CoAP, ou MQTT.

Projet IoTivity

IoTivity [lin15] est un projet open source financé par l'OCF (Open Connectivity Foundation) [ocf16]. Il propose un framework permettant la connectivité entre les équipements D2D (*Device-to-Device*) afin de répondre aux besoins de l'IoT en terme d'interopérabilité. Le framework offre des services aux applications IoT tels que la gestion et la découverte d'équipements, la transmission et la gestion de données. Plusieurs protocoles sont supportés, nous retrouvons par exemple CoAP, basé sur le style architectural REST, mais aussi Zigbee, Z-wave, Bluetooth, etc. En 2016 ce projet a été rejoint par le projet [all17] de l'alliance AllSeen. AllJoyn propose un framework basé sur l'approche producteur / consommateur pour l'échange de données.

Projet FIWARE

FIWARE [fiw17] est un projet conduit par l'union européenne dans le cadre de l'IERC [ier14]. Il propose une plateforme open source de niveau Middleware qui offre un écosystème pour la création de nouvelles applications et services pour la mise en place de scénarios métier et de fonctions de l'IoT. La plateforme offre des fonctionnalités Cloud améliorées basées sur le framework OpenStack (telles que l'allocation de ressources, le provisionnement, le stockage, l'orchestration, etc.) ainsi qu'un ensemble d'outils et de bibliothèques connus sous le nom de *Générique Enablers* (GEs). Ces GEs offrent différentes capacités [fiw17] telles que la délivrance des applications, services et équipements, la gestion des données et du contexte, fourniture d'interfaces avec les réseaux et équipements, le stockage en cloud, la sécurité, etc.

Un nouveau service qu'offre les GEs de l'architecture FIWARE est celui de la gestion du contexte. Ce service propose un mécanisme pour générer, collecter, publier, demander ou analyser des informations (potentiellement massives) de contexte de manière efficace. Ces informations sont utilisées par les applications afin de réagir à leur contexte. Elles peuvent par exemple être des informations de localisation, de présence, liées au profil de l'utilisateur ou aux terminaux. Ce processus est complexe car ces informations peuvent provenir de différentes sources, systèmes, applications, capteurs, etc.

1.4.3. Gestion de la QoS au niveau Middleware

Dans le contexte de l'IoT, en plus des besoins fonctionnels que doit remplir le Middleware en abstrayant la complexité des couches sous-jacentes, d'autres besoins non fonctionnels peuvent être exprimés par les applications IoT et doivent donc être pris en considération, notamment le besoin en QoS. Dans cette section, nous présentons dans un premier temps les travaux proposant des solutions au niveau Middleware pour la gestion de la QoS dans le contexte de l'IoT. Ensuite, nous présentons quelques-unes des principales solutions au niveau Middleware dédiées à d'autres contextes, toujours dans une perspective de gestion de la QoS.

Solutions Middleware orientées IoT pour la gestion de la QoS

Dans le cadre des efforts de standardisation, plusieurs couches de services de niveau Middleware ont été proposées dans le contexte de l'IoT. Ces couches intègrent plusieurs services fonctionnels et non fonctionnels pour la gestion des équipements IoT. Cependant, ces standards ne proposent aucune solution de gestion de la QoS au niveau Middleware [ban15]. Ils considèrent que la QoS résulte de celle fournie par les réseaux sous-jacents.

Plusieurs solutions spécifiques (i.e. hors standard) ont cependant été proposées.

[yu14] propose d'améliorer le Middleware WuKong [lin13] pour la gestion de la QoS. Il introduit la notion de *score de qualité* qui prend de multiples métriques de QoS (temps de réponse, fiabilité, etc.) en considération. Il sélectionne les équipements physiques adéquats et décide du déploiement des équipements le plus optimal, c'est-à-dire engendrant le plus grand niveau de score. La limite de l'approche est liée au fait que les besoins en QoS des différentes applications ne peuvent pas être prises en compte dynamiquement.

Dans le projet MiLAN [hei04], Heinzelman propose un Middleware qui gère le réseau et les nœuds. En fonction de la description de l'application et de ses besoins en QoS, le Middleware configure le réseau et les nœuds pour satisfaire ces besoins. Ceci étant, MiLAN a besoin d'un diagramme d'états spécifique à chaque scénario applicatif et au contexte de réseau de capteurs sans fil, ce qui est relativement plus compliqué vu les changements dynamiques dans le contexte de l'IoT.

D'autres solutions telles que [sir14, red15] reposent sur l'intégration du protocole MQTT [bg15] pour la gestion de la QoS. Ce protocole inclut une forme basique de gestion de la QoS. Il propose trois niveaux de garantie de l'arrivée du message. Le niveau QoS 0 offre un mode *best effort* sans aucun acquittement. Le niveau QoS 1 garantit la délivrance du message au récepteur au moins une fois. Le niveau QoS 2 garantit la délivrance unique du message au récepteur. Enfin, les dernières spécifications du standard oneM2M proposent l'intégration du protocole MQTT [one16]. Face au problème de la QoS, ce protocole offre une certaine garantie pour la délivrance du message. Cependant, dans un contexte IoT, les applications métier peuvent avoir des besoins différents et plus complexes que la simple garantie de délivrance du message.

Solutions Middleware non orientées IoT pour la gestion de la QoS

Dans d'autres contexte que l'IoT, plusieurs solutions ont été proposées pour la gestion de la QoS au niveau Middleware. Nous en citons ici quelques-unes parmi les principales.

DDS [dds04] est une norme de niveau Middleware spécifiée par l'OMG. Elle est dédiée aux environnements à fortes contraintes (orientés temps réel) et propose un Middleware suivant l'approche MOM. La description des données se fait via des langages tels que IDL (*Interface Description Language*) [idl02] pour permettre la communication entre entités hétérogènes du système distribué. Face au problème de QoS, DDS offre un contrôle de bout en bout à travers une multitude d'attributs. Il propose un niveau d'interfaces appelé DCPS (*Data-Centric Publish/Subscribe*) qui permet la configuration de la QoS requise.

Dans [zen04], le Middleware AgFlow est présenté. Il suit une approche SOA et permet la composition des web services guidée par la qualité. La QoS offerte par les web services est capturée par le Middleware et est évaluée par le biais d'un modèle multidimensionnel de QoS. Ce modèle prend en considération des propriétés de QoS telles que le coût de l'exécution, sa durée, la réputation, la fiabilité, et la disponibilité. La sélection des services est réalisée de sorte à optimiser la QoS des exécutions du service composite. Le Middleware prend aussi en considération les changements des services et modifie le plan d'exécution afin d'être conforme aux exigences en QoS de l'utilisateur.

Dans ses travaux de thèse, [dio15] s'inscrit dans le contexte des bus de service orientés SOA. L'auteur propose les principes architecturaux pour un bus de services doté de capacités de gestion de la QoS. L'approche de solution se base sur le paradigme de l'*Autonomic Computing* et propose une gestion dynamique et auto-adaptative d'actions de reconfiguration orientées QoS. Le bus de service proposé offre une prise en compte des exigences de QoS et une gestion

différenciée via des mécanismes orientés QoS. Ceci étant, du fait que la solution est orientée SOA, la désencapsulation du corps du message à chaque fois pour déduire la méthode utilisée engendre des problèmes de scalabilité et des délais de traitement supplémentaire pouvant impacter la QoS requise. Ceci est particulièrement vrai dans un contexte d'IoT où les Middlewares peuvent être déployés sur des équipements limités en termes de performances.

Pour conclure sur les solutions de gestion de la QoS au niveau Middleware, en particulier dans un contexte IoT, les principales solutions proposées dans le contexte de l'IoT ne portent en fait pas réellement sur le Middleware. Elles abordent principalement le sujet par la configuration des niveaux sous-jacents (réseau et équipements) à travers par exemple le redéploiement des équipements ou le changement de la topologie réseau.

Cependant, le niveau Middleware peut lui aussi constituer un goulot d'étranglement susceptible d'engendrer une dégradation notable de la QoS à travers des délais de traitement additionnels mais aussi des pertes potentielles. De manière générale, dans les différents contextes (IoT et hors IoT), la majorité des solutions reposent principalement sur une prise en compte statique (i.e. au démarrage du système) des besoins par le biais de mécanismes de gestion configurés au préalable. Dans un contexte aussi dynamique que celui de l'IoT, cette approche pose des limites significatives vis-à-vis la gestion de la QoS. En effet, le contexte peut être amené à changer en cours de déploiement via l'apparition ou disparition de nouveaux équipements, la modification des configurations des entités Middleware, mais aussi le changement des profils et des besoins applicatifs en fonction de l'étape du scénario métier.

Ainsi, et dans l'inspiration des travaux de [dio15], une approche de gestion de la QoS dotée de propriétés de gestion adaptative et dynamique s'avère nécessaire dans le contexte de l'IoT.

Dans ce qui suit, nous présentons tout d'abord la problématique spécifique que nous considérons dans cette thèse. Nous détaillons ensuite notre positionnement et notre approche générale de gestion de la QoS de bout en bout au niveau Middleware.

1.5. POSITIONNEMENT ET APPROCHE GENERALE DE GESTION DE LA QOS

De manière générale, la notion de *qualité* (en particulier de service) est définie par l'ISO 9000 [iso9000] comme étant "*la totalité des fonctionnalités et des caractéristiques d'un produit ou d'un service sur sa capacité à satisfaire les besoins indiqués ou implicites*". En informatique, la définition de la notion QoS est généralement plus spécifique. Il s'agit des performances globales (temps de réponse, taux d'erreurs, disponibilité, etc.) perçues par l'utilisateur du système.

Dans un contexte spécifique tel que celui de l'IoT, ces besoins non fonctionnels restent valables et doivent être reconsidérés. De par la spécificité des équipements, réduits en taille et en performances, mais aussi des réseaux de communication où plusieurs solutions sont proposées pour gérer la QoS, le niveau Middleware constitue aussi un niveau problématique pour la gestion de ce besoin, mais avec beaucoup moins de propositions de gestion.

Dans cette section, nous donnons tout d'abord notre positionnement spécifique pour la gestion de la QoS au niveau Middleware. Ensuite, nous spécifions la problématique considérée dans le cadre de cette thèse. Enfin, nous exposons notre approche générale de gestion de la QoS de bout en bout.

1.5.1. Contexte spécifique

Plusieurs propositions IoT au niveau Middleware ont été présentées précédemment. Elles se déclinent en deux catégorie : (1) les standards proposant des spécifications de Middleware avec une vue d'ensemble de l'architecture, des composants et de leurs échanges ; (2) les solutions spécifiques qui proposent directement des implémentations de Middleware.

Dans cette thèse, le focus est porté sur les standards qui permettent de donner un cadre formel à ses solutions Middleware. Pour le modèle contextuel, nous considérons une architecture inspirée de la vision que donne les standards SmartM2M et oneM2M. Elle constitue notre infrastructure globale et se décline en les composants suivants (Figure 1.8) : *Applications IoT*, *Serveur IoT* (Middleware), *Gateways IoT* (Middleware), et au final *Equipements IoT*. Dans notre contexte, l'interconnexion entre les applications IoT et le serveur IoT, ou entre les entités Middleware elles-mêmes (serveur-gateway ou gateway-gateway), est supposée être basée sur des réseaux IP. L'interconnexion entre les gateways d'extrémités et les équipements terminaux est supposée s'appuyer sur des réseaux dédiés (Zigbee, Bluetooth, etc.).

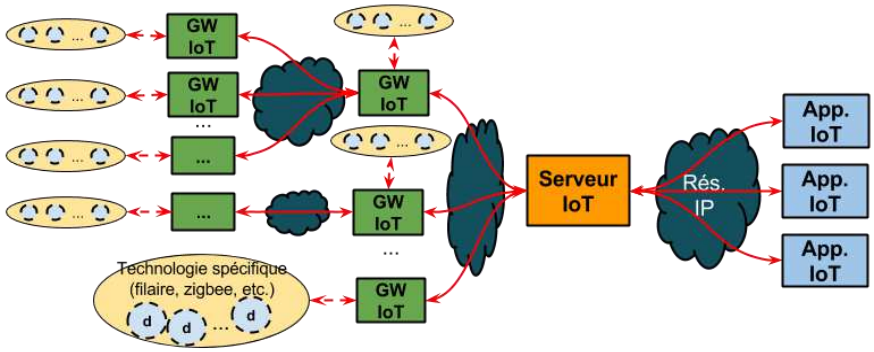

Figure 1.8 - Modèle contextuel considéré

Dans ce contexte, les interactions que nous considérons suivent le style architectural REST [fie00]. Ainsi, les requêtes sont envoyées via un protocole de transfert (par exemple HTTP) utilisant l'une des méthodes CRUD. En fonction de la méthode utilisée, la requête peut comporter un corps contenant la représentation de la donnée à créer ou la mise à jour d'une donnée existante. Dans ce qui suit, nous donnons le modèle considéré de chaque entité de l'architecture.

Modèle d'une application IoT

Les applications IoT diffèrent des applications traditionnelles de l'Internet. En effet, alors que les applications conventionnelles (transfert de fichiers, Web ou même VoIP) impliquent habituellement deux hôtes finaux et des routeurs intermédiaires, les applications IoT se réfèrent à un ensemble d'activités impliquant de nombreuses entités matérielles / logicielles réparties. L'interconnexion de ces entités nécessite l'utilisation de Middleware de communication pour cacher la complexité sous-jacente aux développeurs d'applications. Dans notre cas, l'API de communication avec ces entités Middleware se fait via une interface REST.

Vis-à-vis de la QoS, ces applications peuvent être divisées en deux types. Le premier type concerne les applications qualifiables de *QoS-unaware*. Ces applications ne sont pas capables d'exprimer leurs besoins en QoS. Dans ce cas, c'est le système qui doit gérer le trafic de ces applications en fonction de leur rôle (supervision, intervention, etc.) et de leur profil (type de données et type d'interactions). Le deuxième type est celui des applications *QoS-aware* qui

sont, contrairement au premier type, capables d'exprimer leurs besoins en QoS et éventuellement de participer à leur gestion en interagissant avec le système.

Modèle d'un serveur IoT

Un serveur IoT est une entité Middleware. Le modèle du serveur que nous considérons dispose d'un ensemble de propriétés. Tout d'abord, il constitue le point d'entrée des communications entre les applications métier et au final les équipements via des entités Middleware de type gateway. Le nombre de serveurs dans l'architecture est égale à 1 et seulement 1. Son déploiement se fait sur une plateforme de Cloud et l'accès se fait via Internet. Il implémente la couche NSCL du standard SmartM2M.

Modèle d'une gateway IoT

Les gateway que nous considérons constituent le point d'entrée aux équipements et potentiellement aux autres gateways. Une gateway joue le rôle d'un *proxy* pour les équipements afin de les interfacer avec le réseau de cœur. Elle est donc déployée sur une machine physique proche des équipements, généralement limitée en termes de ressources informatiques. Dans le cas du standard oneM2M, des gateways peuvent être attachées entre elles en séquence et de manière hiérarchique jusqu'au serveur IoT. Le nombre de gateways varie de 1 à plusieurs. Elle peut être attachée à 1 ou plusieurs équipements, et à 0 ou plusieurs autres gateways. Enfin, la gateway implémente la couche GSCL du standard SmartM2M.

Modèle d'un équipement IoT

Les équipements IoT que nous considérons sont supposés *non-ETSI compliant*, i.e. ne respectant pas le standard SmartM2M. Leur intégration au système IoT impose donc l'utilisation de gateway. Le nombre d'équipements varie de 1 à plusieurs pour chaque gateway. Chaque équipement peut réaliser des opérations de capture de métriques (température, humidité, battements de cœur, etc.) et/ou d'actionnement (caméra, moteur, etc.). Il peut être limité en terme d'énergie et disposer d'une batterie. Il communique avec la gateway via une technologie réseau spécifique (éventuellement dans le cadre d'un réseau IP). Ces technologies incluent des réseaux PAN (*Personal Area Network*), telles que IEEE 802.15.1, Zigbee, Bluetooth, IETF ROLL ou ISA100.11a), ou des réseaux LAN (*Local Area Network*) telles que IEEE 802.11, PLC, M-BUS, Wireless M-BUS ou KNX. Enfin, ces équipements ne peuvent pas être déployés dans le Cloud étant donné qu'ils doivent reporter des données / modifier leur environnement physique.

1.5.2. Problématique considérée

Telle qu'introduit précédemment, il existe de nombreuses propositions de gestion de la QoS pour les niveaux équipements et réseau de l'IoT. Cependant, le besoin d'une gestion de la QoS au niveau Middleware est toujours présent. Du fait de la particularité des applications IoT, les approches traditionnelles de gestion de la QoS doivent être reconsidérées pour le niveau Middleware de l'IoT.

Dans la transmission d'une requête et la réception de sa réponse, plusieurs goulots d'étranglement sont susceptibles d'impacter la QoS de bout en bout, notamment en terme de temps de réponse (RT - *Response Time*) global.

Le premier goulot se situe au niveau des réseaux (induisant un temps NT - *Network Time*) qui interconnectent, pour un chemin donné, l'application émettrice avec le serveur (S), le serveur avec la gateway (G) et au final la gateway avec les équipements (d) dont elle a la charge. Le deuxième goulot d'étranglement est lié au traitement des requêtes au sein des différentes entités S, G ou d induisant un temps PT - *Processing Time*.

Ainsi, pour une requête issue d'une application IoT :

- si la requête est destinée au serveur, alors :

$$RT_A = NT_S + PT_S \quad (1.1)$$

- si la requête est destinée à un équipement d'une gateway interconnectée au serveur, alors :

$$RT_A = NT_S + PT_S + NT_G + PT_G + NT_d + PT_d \quad (1.2)$$

- en général, pour le cas d'une requête issue d'une application à destination d'un équipement, et ayant à traverser le serveur et *n* gateways, le temps de réponse total est égal à :

$$RT_A = NT_S + PT_S + \sum_{i=1}^{n}(NT_{Gi} + PT_{Gi}) + NT_d + PT_d \quad (1.3)$$

Le temps réseau de l'équipement dépend de la technologie utilisée (filaire ou non filaire), ainsi que de ses propriétés en terme de vitesse de transfert, par exemple 20-250 kb/s pour Zigbee, 1-3 Mb/s pour Bluetooth, etc.

Les délais NT_i et PT_i induits sur le chemin parcouru par une requête et sa réponse sont variables en fonction de la charge instantanée des entités (S, G, d) et des réseaux impliqués. Il n'y a donc pas de respect systématique des contraintes de QoS requises de bout en bout (ici le temps de réponse) pour assurer le bon fonctionnement de l'application métier.

La problématique adressée dans cette thèse se situe au niveau Middleware et se pose de la façon suivante (en supposant le besoin en QoS réduit au délai) : comment *maîtriser* les délais PT_S et PT_{Gi} de chacune des entités Middleware impliqué dans un chemin de donnée, pour contribuer au respect de la contrainte de QoS de bout en bout sur ce chemin. Le souhait est alors de respecter autant que possible de l'inégalité suivante :

$$PT_S + \sum_{i=1}^{n} PT_{Gi} \leq RT_{MWMAX} \quad (1.5)$$

1.5.3. Approche générale de solution

Dans l'optique d'aboutir à la préservation autant que possible d'une QoS de bout en bout au niveau Middleware, l'approche développée dans cette thèse se traduit tout d'abord par la mise en œuvre d'actions d'adaptation portant sur les entités Middleware, suivant une approche basée *politiques hiérarchiques.*

Nous considérons ainsi deux niveaux d'actions pour le niveau Middleware :

- au plus haut niveau : des actions *stratégiques* conduisant à l'élaboration et l'adaptation *d'objectifs de QoS locale* à atteindre par chacune des entités S et G impliquées dans le chemin de données, afin de répondre aux objectifs de QoS de bout en bout de l'application ;

- au plus bas niveau : des actions *opérationnelles* conduisant à l'adaptation du *comportement* ou des *ressources* des entités S et G pour atteindre l'objectif de QoS locale fixé par le niveau supérieur.

Concernant le niveau stratégique, la Figure 1.9 illustre la décomposition du besoin en QoS de bout en bout (de niveau Middleware) en objectifs de QoS locale à atteindre au niveau de chacune des entités Middleware (gateway et serveur) impliquées sur le chemin reliant une application IoT à (suivant le cas de figure) l'une des entités suivantes : serveur, gateway ou équipement.

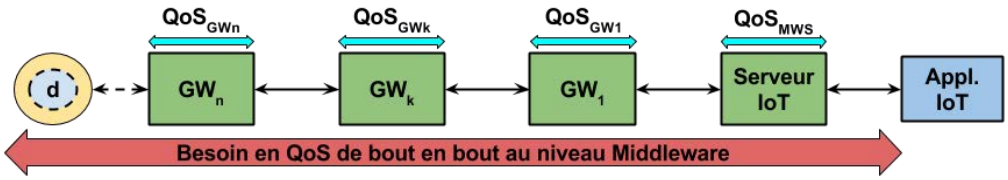

Figure 1.9 - Décomposition du besoin en QoS de bout en bout en objectifs de QoS locale

Concernant le niveau opérationnel, les actions d'adaptation que nous considérons sont inspirées de celles proposées au niveau des couches réseau et transport pour la gestion de la QoS dans l'Internet. Elles sont également inspirées d'actions envisagées dans des environnements de virtualisation. Ces actions sont décrites en détails dans le chapitre 2 de ce mémoire.

Au-delà de cette approche basés politiques hiérarchiques, la proposition que nous développons dans cette thèse vise à répondre aux limites que présentent la plupart des solutions Middleware orientée QoS en termes de capacité d'adaptation face à la dynamicité des environnements. Par exemple, c'est le cas de DDS où les "bons" attributs de QoS sont d'une part difficile à choisir du point de vue du développeur [alf152], mais surtout restent fixes tout au long de l'exécution du système alors l'environnement est susceptible d'évoluer en termes de ressources et/ou de besoin.

Notre approche de solution vise ainsi à assurer deux propriétés quant à l'exécution des actions stratégiques et opérationnelles :

- la première propriété est *l'autonomie* qui signifie que le système doit être capable d'autogérer ses choix pour satisfaire le besoin en QoS des applications. Cette propriété impose une capacité du système à collecter les métriques d'évaluation de la QoS au niveau Middleware, à analyser les causes de dégradation de la QoS, ainsi qu'à élaborer et à exécuter des plans d'actions aux deux niveaux de politiques considérés. Tel que développé au chapitre 3, le modèle architectural sur lequel nous basons cette autonomie est celui de l'*Autonomic Computing* proposé par IBM en 2003 [kep03] ;
- la deuxième propriété visée est que la réalisation du processus d'adaptation dans son ensemble soit effectuée *durant l'exécution du système* et *sans rupture du service* fourni aux applications. Dans nos travaux, nous qualifions cette propriété par la capacité du système à gérer une adaptation *dynamique* des entités Middleware. Notions que notre définition englobe la notion de service *sans couture* évoquée dans certains travaux.

1.5.4. Contributions de cette thèse

Tel que présenté dans l'Introduction générale de ce mémoire, les contributions se déclinent ainsi de la façon suivante.

Nous proposons tout d'abord (chapitre 2) des mécanismes de gestion de la QoS pour l'élaboration des actions d'adaptation relevant du niveau opérationnel. Ces mécanismes se divisent en deux familles. La première famille de mécanismes, *orientée trafic*, repose sur des techniques de marquage, de rejet, de retardement et d'ordonnancement du trafic en fonction de sa priorité. La deuxième famille de mécanismes, *orientée ressources,* portent sur les ressources dont bénéficient les entités Middleware (mémoire, processeurs, etc.).

Nous proposons ensuite (chapitres 3) une architecture de gestion de la QoS de bout en bout permettant : 1) de décliner l'approche basée politiques hiérarchiques précédemment introduite, et 2) de mettre en œuvre le paradigme de l'Autonomic Computing pour guider l'adaptation dynamique et autonome des actions considérées aux deux niveaux de politiques.

Une troisième classe de contribution (chapitres 4 et 5) porte sur la proposition de modèles basés sur les files d'attentes, les séries temporelles et les graphes, pour guider l'élaboration et l'adaptation des actions opérationnelles, c'est-à-dire visant à répondre aux objectifs de QoS locale au niveau des entités Middleware impliqués.

Nous montrons également dans les chapitres 4 et 5 comment appliquer les modèles précédents à certains des composants de gestion de la QoS du gestionnaire de l'autonomie, à savoir le composant de monitoring et le composant de planification, notamment par le biais d'un cas d'étude portant sur la gestion crise dans un métro.

1.6. CONCLUSION

Ce chapitre a présenté le cadre général de notre travail. La première partie a été consacrée à l'IoT et ses applications. Nous avons démontré l'apport de l'IoT dans des contextes traditionnels. Nous avons ensuite donné les différentes visions architecturales de l'IoT. Celle que nous retenons est constituée de quatre niveaux : Application Métier, Middleware, Réseau et Équipements. Nous avons ensuite posé la problématique de la QoS. Le besoin provient d'une analyse des caractéristiques des applications IoT au travers des types de données qu'elles manipulent (brutes et multimédia), des types d'interactions (requête / réponse, commande, souscription / notification, etc.) qu'elles engendrent, et enfin des besoins non fonctionnels qu'elles suscitent, notamment en termes de temps d'exécution.

Face à ces besoins en QoS, l'analyse des goulots d'étranglement que présentent un système IoT nous a conduit à identifier plusieurs enjeux d'amélioration, dont en particulier au niveau Middleware. Au travers d'un état de l'art des solutions Middleware envisagées pour l'IoT, nous avons justifié le fait que ces solutions étaient insuffisantes dans leur façon d'appréhender le problème de la QoS. En effet, les solutions proposées dans le contexte de l'IoT sont conçues dans la seule optique de satisfaire des besoins fonctionnels liés à la (re)configuration des niveaux sous-jacents (réseau et équipements). En revanche, l'impact du Middleware sur la QoS perçue par les application IoT n'est pas considéré alors qu'il induit un délai supplémentaire de traitement voire des pertes potentielles dues à sa propre congestion. Notons cependant que dans d'autres contextes de Middleware tels que les bus de service (ESB) ou les services de distribution de données (DDS) par exemple, les solutions proposées abordent l'impact du Middleware sur la QoS. La majorité d'entre elles considère une prise en compte statique de ce besoin, ce qui est restrictif dans le contexte d'un IoT très évolutif de par ses équipements qui peuvent être ajoutés ou supprimés lors de l'exécution du système, les réseaux qui peuvent changer par exemple de topologie dans le cas de réseaux sans fil, les entités Middleware qui peuvent changer de configuration, ainsi que les applications qui peuvent changer de phases en fonction du scénario et induire des variations des besoins initiaux de QoS.

En réponse aux limites actuelles de l'état de l'art, nous avons enfin positionné notre approche de gestion dynamique et auto-adaptative d'actions orientées QoS au niveau Middleware de l'IoT, basée sur le concept de politiques hiérarchiques. Notre approche considère le contexte spécifique proposé par les standards SmartM2M et oneM2M avec des entités Middleware déployées sur des gateways et serveurs IoT. Pour faire face aux besoins en QoS exprimés par les applications IoT, le système de gestion considère deux niveaux d'actions, *opérationnelles* et *stratégiques*. L'autogestion de l'élaboration et de l'adaptation de ces actions est basée sur le paradigme de l'Autonomic Computing. Enfin, nous avons resitué les contributions principales de la thèse présentées dans l'introduction générale de ce mémoire.

Chapitre 2

Mécanismes de gestion de la QoS au niveau Middleware

Contenu

2.1. INTRODUCTION

Notre approche de gestion de la QoS se traduit (au final) par la mise en œuvre de mécanismes ayant un impact sur les performances du système considéré. Ces mécanismes sont activés dans le cadre d'une architecture de gestion conduisant les entités de niveau Middleware à être adaptées de façon dynamique (c'est-à-dire durant leur exécution) pour qu'elles répondent au mieux aux contraintes de QoS des applications et de l'opérateur du système.

Le premier besoin concerne donc l'élaboration de mécanismes pertinents vis-à-vis du contexte de l'IoT. Ce contexte est dynamique à la fois par les types d'entités déployées mais aussi leur environnement de déploiement, qui peut être physique ou virtuel. Tels que présentés dans l'état de l'art, de nombreux travaux ont été menés aux niveaux IP et Transport, ainsi qu'au niveau Middleware notamment dans des bus de service. Ces mécanismes peuvent porter sur le trafic lui-même, comme dans le cas des solutions de niveaux IP et Transport, ou sur les ressources des machines comme il a été proposé pour les bus de service et dans le cloud computing. Dans

notre proposition, nous considérons ces deux familles de mécanismes, *orientés trafic* et *orientés ressources*. Pour en évaluer les bénéfices, nous proposons un ensemble de scénarios permettant de tester les performances de ces mécanismes sur une plateforme réduite à une entité Middleware, confrontée à des requêtes applicatives généré par un émulateur de trafic IoT. Cette campagne d'évaluation vise à valider les mécanismes à travers la démonstration de leurs bénéfices mais aussi le coût lié à leur activation.

La suite de ce chapitre est structurée de la manière suivante : la section 2.2 introduit les entités Middleware qui vont être gérées ainsi que les types de sources de trafic considérées. La section 2.3 décrit de manière détaillée les mécanismes orientés trafic élaborés. Elle présente également les scénarios de validation et les résultats issus de la campagne d'évaluation des performances induites par ces mécanismes. Alors que la section 2.4 décrit les mécanismes orientés ressources pour l'amélioration des performances du Middleware ainsi que leur validation. Nous terminons le chapitre par une conclusion sur le travail mené.

2.2. ENTITÉS MIDDLEWARE ET SOURCES DE TRAFIC

L'élaboration des mécanismes de gestion de la QoS doit tenir compte des caractéristiques des entités intervenantes. Les premières entités sont les entités Middleware qui, en fonction de leur performance, peuvent induire une dégradation de la QoS. La gestion et l'intégration de ces mécanismes se fait donc au niveau de ces entités ou bien en amont. Les deuxièmes entités sont les sources de trafic. Ce sont elles qui peuvent être critiques et exprimer des besoins en QoS. Dans cette section, nous décrivons ces deux types d'entités.

2.2.1. Entités Middleware

Le modèle contextuel du système IoT considéré (Figure 2.1) se base sur des entités Middleware (*gateway* et *serveur* dans le standard SmartM2M [ets690]) intermédiaires entre les applications et les équipements contribuant au traitement du trafic. Ce trafic est appelé de manière générique trafic WTP (*Web Transfer Protocol*) indépendamment du protocole de transfert (par exemple, HTTP ou CoAP). Une *gateway IoT* est une passerelle permettant d'interconnecter les équipements IoT (par exemple, capteurs, actionneurs) au reste du système. Une gateway est contrainte d'être au même emplacement physique que ces équipements. Elle est (dans notre contexte) déployée sur une plate-forme physique et donc limitée en terme de ressources informatiques. Un *serveur IoT* constitue le point de concentration permettant l'interconnexion des différentes applications IoT avec les différentes gateway pour l'accès aux équipements IoT. Un serveur peut être déployé sur un environnement virtualisé tel que le cloud.

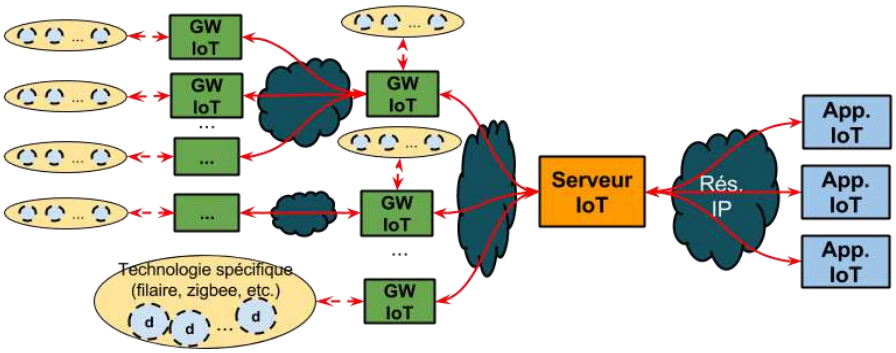

Figure 2.1 - Modèle Contextuel du système IoT

En fonction de leur état, ces entités sont susceptibles de conduire à la non satisfaction des besoins en QoS du trafic issu des sources. C'est donc au profit de ces deux entités que les mécanismes (en tout ou partie, et en fonction de l'entité) vont être déployés afin de respecter les contraintes QoS des sources tout en assurant la délivrance des services IoT.

La gestion de ces entités Middleware se passe par la proposition de mécanismes orientés QoS. Dans notre approche, cette gestion est assurée de manière dynamique et auto-adaptative suivant le paradigme de l'*Autonomic Computing* proposé par IBM [kep03]. Ce paradigme propose une boucle MAPE-K d'autogestion enchaînant des phases de supervision (M - *Monitoring*) de l'entité gérée (entité Middleware), d'analyse (A - *Analysis*) des causes de dégradation, de planification (P - *Planning*) et d'exécution (E - *Execution*). Ce paradigme met en œuvre des politiques, des règles, des modèles et/ou des algorithmes maintenus dans une base de connaissance (K). Dans ce chapitre, nous allons placer un gestionnaire autonomique pour la configuration des entités Middleware et des composants implémentant les mécanismes QoS. La spécification et la conception de l'architecture du système de gestion va être fournie en détail dans le chapitre 3.

2.2.2. Sources de trafic

Les *sources de trafic* sont les entités génératrices de trafic. En fonction du service métier fourni, le trafic émanant de ces sources peut être critique et présenté des besoins en QoS (délai, pertes, etc.). Le système doit donc répondre à ces besoins. Nous pouvons distinguer ces sources en fonction de l'entité Middleware destinataire (Tableau 2.1), c'est-à-dire via laquelle passe le trafic.

Middleware destinataire	Source de trafic
Serveur IoT	Application IoT
	Gateway IoT
	Equipement IoT implémentant la couche de service Middleware
Gateway IoT	Application IoT
	Serveur IoT
	Equipement IoT non capable d'implémenter la couche de service

Tableau 2.1 - Sources du trafic en fonction du Middleware destinataire

Vis-à-vis du besoin en QoS, nous distinguons deux types de sources :

- **source QoS-unaware :** source qui n'est pas dotée de moyens pour exprimer ses besoins en QoS. Le système IoT doit donc, en fonction des caractéristiques de la requête (source, destination, méthode, etc.), identifier la priorité à lui accorder pour son traitement ;
- **source QoS-aware :** contrairement au premier type, les sources de ce type sont capables d'exprimer leurs besoins en QoS au système, et éventuellement de participer à la gestion de la QoS en interagissant avec le système IoT, par exemple, en adaptant le débit de leurs requêtes.

Les types de mécanismes que nous considérons dépendent du type de l'entité Middleware gérée. Une gateway, qui est déployée sur une machine physique, ne peut tirer parti que des

mécanismes orientés trafic. Un serveur IoT, déployable / déployé dans un environnement cloud, peut exploiter les mécanismes orientés ressources en plus de ceux orientés trafic.

Dans la suite de ce chapitre, nous définissons ces deux types de mécanismes, leur instanciation dans le contexte considéré, ainsi que leur validation dans le cadre d'un certain nombre de scénarios.

2.3. MÉCANISMES DE GESTION ORIENTÉS TRAFIC

Les mécanismes de gestion orientés trafic sont inspirés de ceux utilisés au niveau des couches Transport et IP de l'architecture de l'Internet pour la gestion de la QoS. Ces mécanismes se positionnent au niveau Middleware et sont, dans notre travail, placés en amont de l'entité Middleware. Ils agissent directement sur le trafic WTP entrant. Dans la gestion proposée, nous nous basons sur un principe de différenciation de service analogue à celui proposé dans le modèle DiffServ de l'IETF [rfc2475]. Nous distinguons deux composants essentiels : le composant de classification et de marquage qui permet d'attribuer des priorités au trafic WTP, et le proxy orienté priorité qui différencie le traitement du trafic en fonction de sa priorité.

2.3.1. Classification et Marquage du trafic

Inspirée de la terminologie DiffServ, la fonction de classification et de marquage fournit la base pour la mise en œuvre des techniques de gestion de la QoS. Cette fonction se base sur la notion de priorité. Ces priorités correspondent au marquage du trafic en fonction de sa classe d'appartenance. La classification du trafic WTP se base sur des critères tels que l'adresse de la source, le type de la ressource destinataire, etc.

En-têtes WTP

Le composant de classification et de marquage se base sur certains des champs d'en-tête du trafic pour d'abord le classer et ensuite lui attribuer une priorité. Par défaut, les protocoles WTP (HTTP ou CoAP par exemple) ne comportent pas de champ permettant de spécifier une priorité. Ceci étant, ils permettent d'ajouter des en-têtes personnalisés. Nous exploitons donc cette propriété en intégrant un en-tête que nous appelons TOS_WTP (*Type of Service for WTP*), analogue au champ TOS des paquets IP.

Afin de garantir que l'en-tête TOS_WTP soit toujours à jour pour les sources QoS-aware, nous ajoutons un autre en-tête TOS_WTP_V qui reflète sa version. En cas de mise à jour des politiques de classification et de marquage, le TOS_WTP_V sert à vérifier si le TOS_WTP est à jour avec la nouvelle politique. S'il ne l'est pas, l'en-tête TOS_WTP ne sera pas pris en considération et le trafic sera traité comme un trafic émanant d'une source QoS-unaware. Suite à cette action, une notification est envoyée à la source émettrice de ce trafic pour l'avertir.

Classification du trafic

L'objectif de cette étape est de faire une distinction entre les différents trafics WTP. En fonction des caractéristiques du trafic, une classe lui sera attribuée. Cette classification concerne le trafic en provenance des sources QoS-unaware qui ne sont pas capables d'ajouter directement l'en-tête TOS_WTP pour exprimer leurs besoins en QoS.

Usuellement, un trafic applicatif est souvent défini par cinq éléments : adresse IP de la source, adresse IP destination, port source, port destination et protocole de Transport. Dans notre cas,

nous élaborons la classification de façon plus complexe sur la base des champs suivants (Tableau 2.2) :

- **Adresse source** : si c'est source est une application, étant donné que sur une même machine, plusieurs applications peuvent y être hébergées, la classification se fait sur la base de son adresse IP et son numéro de port. Par contre, si la source est l'entité Middleware (serveur ou gateway), la classification se fait seulement sur la base de l'adresse IP ;
- **Adresse destination :** idem que dans le cas précédent à ceci près que la classification se base sur l'adresse de la destination au lieu de celle de la source ;
- **Type de source :** la distinction du trafic WTP est faite sur la base du type de source : application IoT, serveur IoT, gateway IoT ou équipement ;
- **Type de destination :** la distinction du trafic WTP est faite sur la base du type de destinataire : application IoT, serveur IoT (base de données locale), gateway IoT (base de données locale) ou équipement ;
- **Type de données :** la distinction est faite en fonction du type de trafic entrant : données brutes, audio temps réel, vidéo temps réel, image, etc. ;
- **Type d'interaction :** la distinction est faite entre les données de signalisation (requête, souscription, etc.) et les données utiles (réponse, notification, données périodiques, alarme, etc.) ;
- **Méthode d'interaction :** la distinction est faite à partir des méthodes WTP utilisées : création, récupération, mise à jour, exécution ou suppression.

Base	**@SRC ou @DST**		**Type SRC ou DST**	**Type de données**	**Type d'interaction**	**Méthode d'interaction**
Critère(s) de classification	**APP**	**MW**	Contexte de la requête WTP (APP, SRV ou GW)	Brutes Image Audio Vidéo	Signalisation ou donnée utile	GET POST PUT DELETE
	@IP+n° port	@IP (Srv ou GW)				

Tableau 2.2 - Critères de classification du trafic

A l'issue de cette classification, chaque requête est ainsi affectée à une classe donnée. A titre d'exemple, pour une classification sur la base du type de trafic, nous pourrions avoir les classes illustrées par le Tableau 2.3.

Type de données	**Classe**
Brutes	classe 1
Vidéo	classe 2
Audio	classe 3
Images	classe 4

Tableau 2.3 - Exemple de politique de classification de trafic basée sur le type de données

Cette classe est ensuite utilisée par le composant de marquage afin d'attribuer au trafic la priorité correspondante via l'en-tête TOS_WTP.

Marquage du trafic

Cette étape consiste en l'attribution d'une priorité à chaque trafic en fonction de sa classe d'appartenance. Le marquage se base sur une correspondance entre les classes et les priorités fournies en amont par le gestionnaire autonomique (Tableau 2.4). L'en-tête TOS_WTP est ajouté et se voit attribuer la valeur correspondant à la priorité du trafic.

Classe	Priorité
classe 1	PRIOR_1
classe 2	PRIOR_2
classe 3	PRIOR_3
classe 4	PRIOR_4

Tableau 2.4 - Exemple de politique de marquage de trafic

Soit l'exemple du domaine de la télésurveillance d'un patient à domicile. Dans ce domaine, nous considérons trois applications de criticités différentes (hautement critique, moyennement critique et non critique) :

- L'application de télésurveillance postopératoire du patient est la plus critique. Elle est à la fois très sensible au délai (inférieur à 300 ms [sko10]) et aux pertes (aucune perte) ;
- L'application du suivi de la pression artérielle est moyennement critique (sensibilité moyenne au délai (10 s) et aux pertes (50%)) ;
- L'application du suivi du pourcentage de calorie n'est pas critique.

Le Tableau 2.5 suivant illustre les priorités attribuées au trafic (http par exemple) échangé dans le cadre des trois classes d'applications IoT pour la satisfaction de leurs besoins en délai ou en pertes.

Application IoT de télésurveillance	délai seuil	Pertes Seuil	TOS_WTP (besoin en délai)	TOS_WTP (besoin en pertes)
Postopératoire	300ms	0%	PRIORITY_DELAY_HIGH	PRIORITY_LOSSES_HIGH
Pression artérielle	10s	50%	PRIORITY_DELAY_MEDIUM	PRIORITY_LOSSES_MEDIUM
Calories brûlées	-	-	PRIORITY_DELAY_LOW	PRIORITY_LOSSES_LOW

Tableau 2.5 - Exemple d'attribution de priorités selon la sensibilité du trafic

Architecture fonctionnelle du CCM

Le composant de classification et de marquage (CCM) constitue le premier élément par lequel transite le trafic. Ce composant offre une interface de gestion permettant de l'activer, de le désactiver, ou de le configurer en termes de politiques de classification et de marquage.

La Figure 2.2 représente l'architecture fonctionnelle du CCM. Elle est constituée des composants suivants :

- **Gestionnaire CCM :** reçoit l'ordre d'activation / désactivation de la part de l'Autonomic Manager ainsi que les politiques pour guider la classification et de marquage ;
- **Récepteur WTP :** responsable de la réception du trafic WTP issue de la source. Ce récepteur est capable de recevoir le trafic quel que soit le protocole utilisé (HTTP ou CoAP par exemple) ;
- **Contrôleur CCM :** reçoit le trafic de la part du Récepteur WTP. En cas d'activation, il dirige le trafic soit vers le Classificateur WTP si l'en-tête TOS_WTP n'existe pas, sinon directement vers le Client WTP si l'en-tête existe déjà. En cas de désactivation, le trafic est directement dirigé vers le Client WTP ;
- **Classificateur WTP :** classifie le trafic entrant en fonction de la politique de classification. Ce composant réalise tout d'abord l'analyse du trafic en fonction de la politique de classification (adresse IP source, adresse IP destination, l'URL de la ressource, la méthode, etc.). Ensuite, il identifie la classe d'appartenance du trafic en

fonction de ses caractéristiques. Il redirige ensuite le trafic vers le composant Marqueur de priorité ;

- **Marqueur WTP :** attribue une priorité pour chaque classe en se basant sur la politique communiquée par le Gestionnaire de composants. Il ajoute l'en-tête TOS_WTP au trafic et lui attribue une valeur correspondant à la priorité du trafic ;
- **Client WTP :** redirige le trafic vers la prochaine entité. Cette dernière peut être soit le Proxy orienté priorités, soit le système destinataire. L'adresse de redirection est communiquée par le Gestionnaire Autonomique.

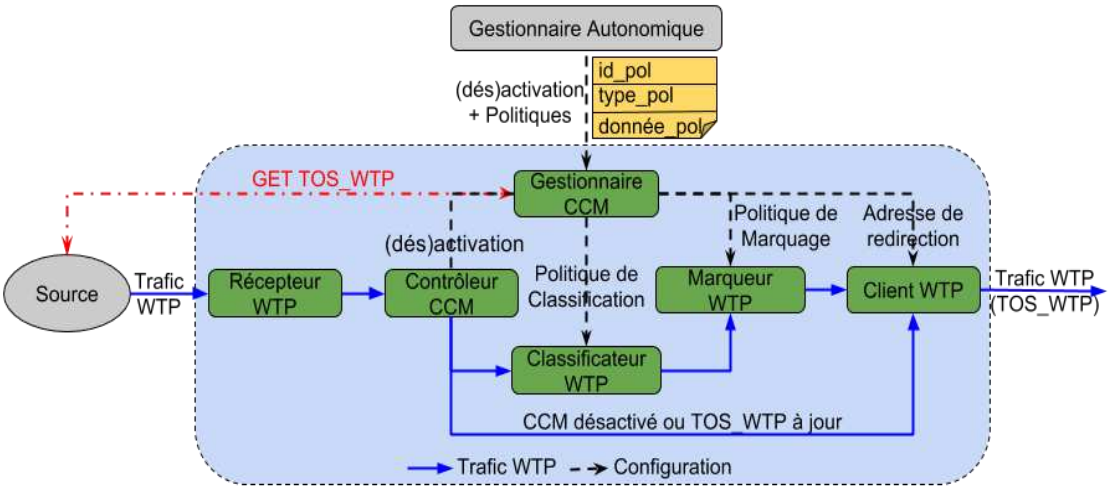

Figure 2.2 - Architecture fonctionnelle du CCM

La procédure d'ajout de l'en-tête TOS_WTP contenant une priorité est illustrée comme suit (Figure 2.3) :

1. initialement, le CCM est en état de désactivation. Le trafic passe par le composant Récepteur WTP ensuite par le Contrôleur CCM" qui le redirige directement vers le Client WTP ;
2. le CCM est activé par le Gestionnaire Autonomique. Ce dernier lui communique la politique de classification au Classificateur WTP et la politique de marquage au Marqueur WTP ;
3. à chaque réception d'un trafic WTP, le Récepteur WTP l'oriente vers le Contrôleur CCM qui vérifie l'existence de l'en-tête TOS_WTP et la validité du TOS_WTP_V. Si c'est le cas, il l'oriente directement vers le Client WTP. Sinon, il le dirige vers le Classificateur WTP qui le classe selon sa politique de classification et l'oriente vers le Marqueur WTP ;
4. le Marqueur WTP effectue la correspondance entre la classe et la priorité et ajoute l'en-tête TOS_WTP avec la priorité correspondante ;
5. enfin, le trafic est acheminé vers le Client WTP pour le rediriger. L'adresse de redirection est fournie par l'AM.

Les sources QoS-aware peuvent communiquer via une autre interface avec le Gestionnaire CCM pour récupérer les politiques de classification et de marquage existantes. Elles pourront donc ajouter par elles-mêmes l'en-tête TOS_WTP de leurs trafics WTP en fonction de leurs besoins en QoS.

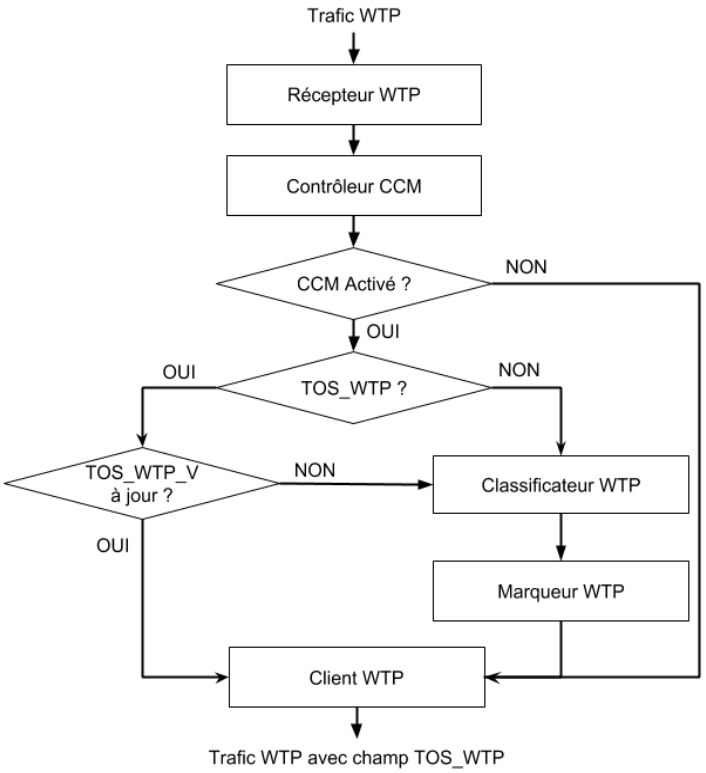

Figure 2.3 - Organigramme de programmation du CCM

2.3.2. Proxy orienté priorités

Le proxy orienté priorités (POP) est équivalent à un PEP (*Performance-enhancing Proxy*) [rfc3135] qui est utilisé pour l'amélioration des performances globales de certains protocoles de communication. Le POP est utilisé au niveau Middleware et se base sur la priorité attribuée au trafic (TOS_WTP) issu du CCM (Figure 2.4) afin d'élaborer sa gestion en fonction de la politique.

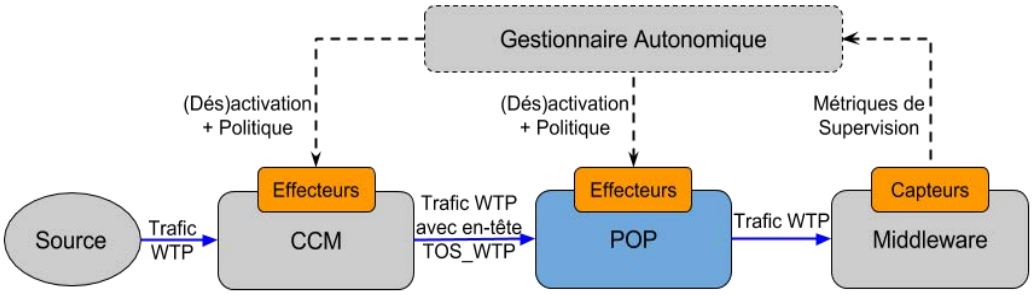

Figure 2.4 - Proxy orienté priorités dans l'architecture

Le fonctionnement du POP est inspiré des techniques de gestion des flux au niveau des couches IP et Transport. Nous retrouvons, par exemple, le rejet du trafic le moins sensible aux pertes, le retardement du trafic le moins sensible au délai, l'ordonnancement du trafic selon sa priorité avant envoi au Middleware, ou le contrôle de congestion en interagissant avec les sources QoS-aware. Certaines de ces techniques sont intégrées dans le POP dont l'architecture fonctionnelle est proposée ci-après.

Architecture fonctionnelle du Proxy orienté priorités

Le POP intègre des techniques de gestion basées sur l'en-tête TOS_WTP. La Figure 2.5 représente son architecture fonctionnelle. Elle est constituée des éléments suivants :

- **Gestionnaire POP :** reçoit l'ordre d'activation / désactivation du Proxy de la part du gestionnaire autonomique ainsi que la politique à appliquer par les composants. Cette politique comporte le(s) mécanisme(s) à déclencher (rejet, retardement ou ordonnancement) ainsi que les paramètres éventuels de ce(s) mécanisme(s) ;
- **Récepteur POP :** reçoit le trafic WTP (HTTP, CoAP, etc.) et l'oriente vers le Contrôleur POP ;
- **Contrôleur POP :** en fonction de la politique, il oriente le trafic vers les mécanismes impliqués dans la politique de gestion ensuite l'envoie vers le Client WTP. Si le POP est désactivé, il oriente le trafic directement vers le Client WTP ;
- **Rejeteur POP :** responsable du rejet de trafic selon la politique de rejet communiquée par le composant Gestionnaire du Proxy ;
- **Retardateur POP :** permet le retardement de trafic selon la politique de retardement communiquée par le composant Gestionnaire du Proxy ;
- **Ordonnanceur POP :** responsable de l'ordonnancement du trafic selon la politique d'ordonnancement communiquée par le composant Gestionnaire du Proxy ;
- **Client POP :** envoie le trafic WTP vers le système destinataire.

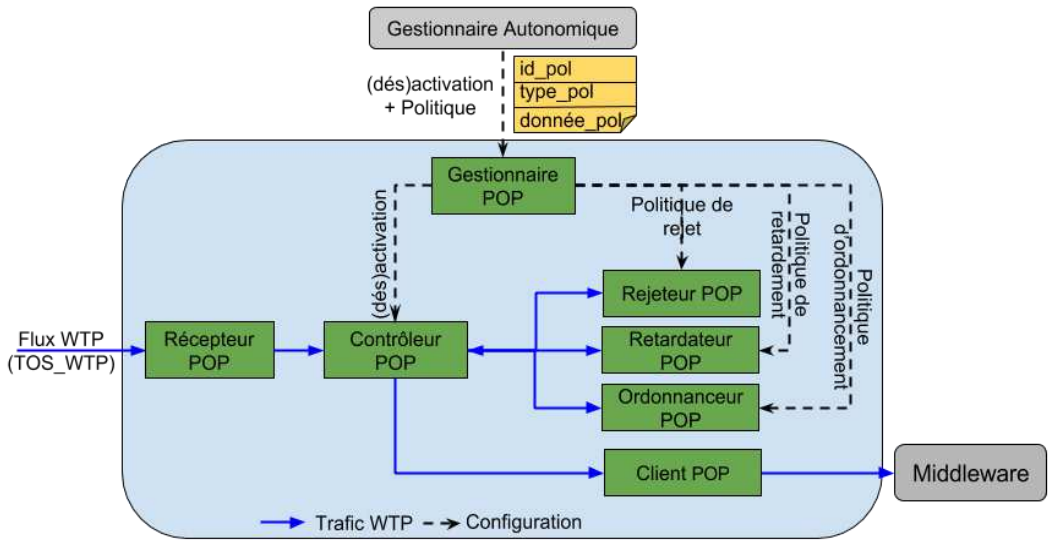

Figure 2.5 - Architecture fonctionnelle du Proxy orienté priorités

La configuration des composants (rejet, retardement et/ou ordonnancement) dépend de la politique de gestion. Elle est constituée des paramètres suivants :

- **id_pol :** identifiant de la politique ;
- **type_pol :** mécanisme(s) concerné(s) dans la gestion orientée priorité qui peut être soit du rejet, retardement, ordonnancement ou une combinaison d'eux ;
- **donnée_pol :** comporte les paramètres du/des mécanisme(s).

Politique POP avec une gestion orientée rejet

La gestion orientée rejet, réalisée par le composant *Rejeteur POP*, consiste en la possibilité de rejeter du trafic à hauteur d'un pourcentage donné en fonction de sa priorité. La politique de

rejet dépend de la sensibilité du trafic aux pertes et fait correspondre la priorité du trafic et le pourcentage de son rejet. Le Tableau 2.6 donne à titre d'exemple une politique de rejet.

TOS_WTP	Pourcentage de rejet
PRIORITY_LOSSES_HIGH	0%
PRIORITY_LOSSES_MEDIUM	40%
PRIORITY_LOSSES_LOW	80%

Tableau 2.6 - Exemple de politique de gestion orientée de rejet

Dans cet exemple, différents pourcentages de rejet sont utilisés en fonction de la priorité. Pour le trafic le plus sensible aux pertes (PRIORITY_LOSSES_HIGH), aucune requête n'est rejetée. Le trafic moyennement sensible au rejet (PRIORITY_LOSSES_MEDIUM) est rejetable à hauteur 30%. Le trafic le moins sensible aux pertes (PRIORITY_LOSSES_LOW) peut être rejeté jusqu'à 80%.

Politique POP avec une gestion orientée retardement

La gestion orientée retardement est réalisée par le composant *Retardateur WTP*. Elle consiste en la possibilité de retarder du trafic à l'intérieur du POP en fonction de sa priorité. La politique de retardement dépend de la sensibilité du trafic au délai.

TOS_WTP	Délai de retardement (ms)
PRIORITY_DELAY_HIGH	0
PRIORITY_DELAY_MEDIUM	5000
PRIORITY_DELAY_LOW	10000

Tableau 2.7 - Exemple de politique de retardement

Le Tableau 2.7 illustre un exemple de politique de retardement. Pour le trafic le plus sensible au délai (PRIORITY_DELAY_HIGH), aucun délai de retardement n'est autorisé. Le trafic moyennement sensible au délai (PRIORITY_DELAY_MEDIUM) peut être retardé de 5000 ms. Le trafic le moins sensible au délai (PRIORITY_DELAY_LOW) peut avoir un retard allant jusqu'à 10000 ms.

Politique POP avec une gestion orientée ordonnancement

Au sein du Proxy, la gestion orientée ordonnancement est réalisée par le composant *Ordonnanceur POP*. Elle repose sur deux actions principales : (1) la classification des requêtes sur une/des file(s) d'attente en fonction de la politique choisie de classification, et (2) l'application d'une politique d'ordonnancement (par priorité, Round Robbin, WFQ, etc.). Par défaut, si aucune politique d'ordonnancement n'est configurée, l'ordonnancement se fait suivant la technique FIFO. La politique d'ordonnancement de ce composant comporte les éléments suivants :

- **nombre_files :** nombre de files d'attente ;
- **technique_classification :** technique de classification du trafic dans les files en fonction de la priorité ;
- **technique_ordonnancement :** technique d'ordonnancement des files d'attente, avec les paramètres adéquats.

Le Tableau 2.8 illustre un exemple de politique de gestion orientée ordonnancement. Cette politique est basée sur la technique WFQ où un poids est attribué à chaque priorité. Ce poids

traduit le nombre de requêtes à exécuter au niveau d'une priorité donnée avant le passage à la priorité la plus basse.

TOS_WTP	Nombre de requêtes par cycle
PRIORITY_DELAY_HIGH	10
PRIORITY_DELAY_MEDIUM	5
PRIORITY_DELAY_LOW	1

Tableau 2.8 - Exemple de politique de retardement

Dans un cycle, pour le trafic le plus sensible au délai (PRIORITY_DELAY_HIGH), l'ordonnanceur va exécuter 10 requêtes de ce trafic. L'ordonnanceur passe ensuite au trafic de priorité moyenne (PRIORITY_DELAY_MEDIUM) et en exécute 5. Enfin, l'ordonnanceur ne va exécuter qu'une seule requête du trafic le moins prioritaire (PRIORITY_DELAY_LOW) avant d'achever le cycle et revenir au trafic le plus prioritaire.

2.3.3. Scénarios de validation des mécanismes orientés trafic

Dans cette section, nous présentons quelques scénarios visant à valider le bien fondé des mécanismes présentés précédemment. Ces scénarios ont été mis en œuvre sur une plateforme d'émulation (présentée ci-après) qui permet de confronter les composants proposés pour la gestion de la QoS à du trafic IoT émulé. La validation de ces composants passe tout d'abord par l'évaluation des bénéfices apportés sur la QoS (ici temps de réponse) des applications critique par l'ajout des composants CCM et POP. Nous mesurons ensuite le coût de l'ajout de ces composants en termes de ressources consommées (CPU, RAM, etc.) et de temps de traitement additionnel. Dans la structuration de cette section, nous présentons tout d'abord l'architecture d'émulation. Nous décrivons ensuite la spécification du scénario de validation de chaque mécanisme (rejet, retardement et ordonnancement).

Architecture de validation

L'architecture de validation est représentée par la Figure 2.6. Elle est basée sur un émulateur de trafic dont le rôle est de générer du trafic HTTP ou CoAP. Il permet de générer différents types de trafic (stochastique, périodique et rafale) depuis *n* injecteurs générant chacun *p* requêtes avec une fréquence de *f* requêtes/s.

Ce trafic est dirigé vers les composants implémentant la gestion orientée trafic (CCM + POP) avant d'être réorienté vers l'entité Middleware (ici réduit une entité de type gateway IoT). L'implémentation utilisée dans cette validation est le Middleware OM2M [ben14] dont les détails sont fournis dans le chapitre 4. Les métriques observées sont le temps de réponse de chaque injecteur (en millisecondes - ms), la CPU (%) et la RAM (octets) consommées par les différents composants.

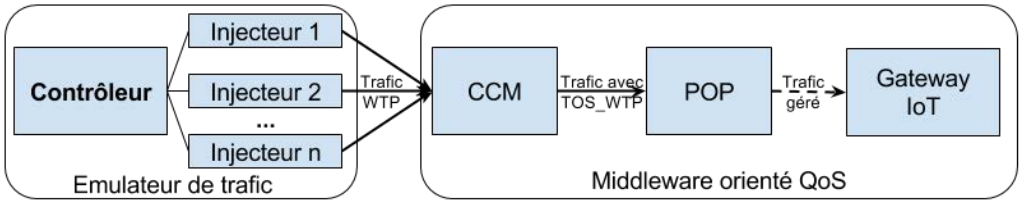

Figure 2.6 - Architecture de validation basée émulation

Chacun des composants (émulateur de trafic, CCM, POP et gateway IoT) est déployé sur une machine à part. Leurs caractéristiques sont représentées par le Tableau 2.9.

Ressource	Emulateur	CCM	POP	Gateway IoT
RAM (Mo)	1024	1024	2048	512
CPU (cœurs)	1	1	2	1

Tableau 2.9 - Caractéristiques des composants de l'architecture de validation

L'émulateur de trafic permet d'exécuter un nombre paramétrable d'injecteurs (*n* injecteurs sur la figure) gérés par un contrôleur. Il est conçu pour générer n'importe quel type de requêtes (HTTP ou CoAP) et de méthode (GET, POST, PUT et DELETE). Dans notre cas, nous déployons trois injecteurs ayant les caractéristiques exprimées par le Tableau 2.10. Les requêtes générées par chaque injecteur sont ici de type HTTP. Elles sont dédiées à la création, via la méthode POST, de ressources de type <*ContentInstance*>.

Injecteur	Taux d'arrivée en moyenne (reqs/s)	Application IoT de télésurveillance	Délai seuil (ms)	Pertes seuil (%)
1	6	Post-opératoire	300	0
2	6	Pression artérielle	10000	50
3	8	Calories brûlées	-	-

Tableau 2.10 - Caractéristiques de chaque injecteur et exigences

En fonction des caractéristiques des injecteurs, le CCM attribue des priorités aux trafics des différents injecteurs. Pour les différents mécanismes, l'observation et la validation seront focalisées sur le trafic issu de l'injecteur 1 (représentant l'application de télésurveillance post-opératoire) étant donné qu'il est le plus sensible au délai et aux pertes. La métrique observée est le temps de réponse de ce trafic. L'injecteur 1 n'est lancé qu'après le lancement des deux autres injecteurs, l'objectif étant d'étudier la valeur ajoutée des composants CCM et POP sur un trafic critique en présence de trafic moins critique.

Scénario sans gestion de QoS

Dans le cas où il n'y a pas de gestion de la QoS, le temps de réponse du trafic de l'injecteur 1 suit l'évolution représentée par la Figure 2.7.

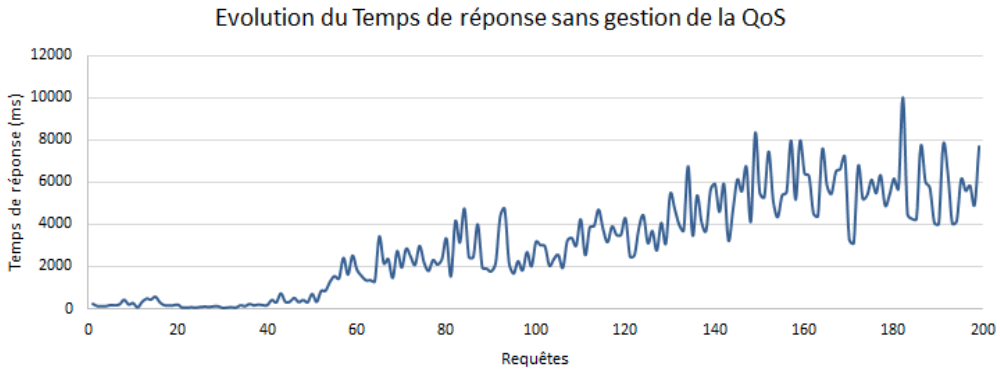

Figure 2.7 - Evolution du temps de réponse sans gestion de la QoS

Dans ce scénario de base, le seuil exigé par l'injecteur 1 (300 ms) est dépassé au bout de la 36ème requête (536 ms). Il évolue rapidement pour atteindre des temps de réponse supérieurs à 6000 ms. En moyenne, le temps de réponse est de 3067 ms qui est largement supérieur au seuil critique. Nous pouvons donc constater que par défaut, sans intégration de mécanismes de

gestion de la QoS, le Middleware n'est pas capable de garantir systématiquement les besoins en QoS des applications critiques.

Les scénarios qui suivent intègrent la gestion de la QoS. Afin de guider cette gestion, nous identifions trois catégories de symptômes problématiques en fonction de la valeur du temps de réponse (Tableau 2.11). Chaque symptôme est ici supposé être généré après *n* événements successifs du même état (*n* = 5 dans notre cas).

RTT (ms)	Etat	Symptôme
$RTT_{Inj1} < 300$	Normal	RTT_NORMAL
$300 \leq RTT_{Inj1} < 400$	Warning	RTT_WARNING
$RTT_{Inj1} \geq 400$	Critical	RTT_CRITITAL

Tableau 2.11 - Symptômes générés en fonction de la valeur du temps de réponse

Dans ce qui suit, nous présentons de façon analogue les scénarios de gestion de la QoS basée sur des mécanismes orientés rejet, retardement ou ordonnancement.

Scénario de gestion orientée rejet

Compte tenu du taux de pertes des différents injecteurs, le CCM attribue une priorité au trafic selon la politique de rejet (Tableau 2.12). Il ajoute l'en-tête TOS_WTP à chaque requête entrante en se basant sur l'adresse de la source (injecteur 1, injecteur 2 ou injecteur 3).

Injecteur	Priorité
Injecteur 1	PRIORITY_LOSSES_HIGH
Injecteur 2	PRIORITY_LOSSES_MEDIUM
Injecteur 3	PRIORITY_LOSSES_LOW

Tableau 2.12 - Politique de priorisation pour le scénario orienté rejet

Pour une gestion orientée rejet, la politique de rejet du trafic généré par les autres injecteurs tient compte du symptôme généré en fonction du RTT du trafic de l'injecteur 1.

Symptôme	RTT_NORMAL	RTT_WARNING	RTT_CRITICAL
PRIORITY_LOSSES_MEDIUM (%)	0	30	40
PRIORITY_LOSSES_LOW (%)	0	70	80

Tableau 2.13 - Politique de rejet pour le scénario orienté rejet

Le Tableau 2.13 décrit la politique de rejet. Tant que le symptôme RTT_NORMAL est généré, aucune politique de rejet n'est appliquée. Lorsque le symptôme RTT_WARNING est déclenché, le trafic de priorité PRIORITY_LOSSES_MEDIUM est rejeté à 30% alors que le trafic de priorité PRIORITY_LOSSES_MEDIUM est rejeté à 70%. Quand le symptôme est RTT_CRITICAL, le trafic de priorité PRIORITY_LOSSES_MEDIUM est rejeté à 40% alors que le trafic de priorité PRIORITY_LOSSES_MEDIUM est rejeté à 80%.

En appliquant ces règles, le RTT de l'injecteur 1 suit l'évolution présentée par la Figure 2.8. Dans ce scénario, le temps de réponse des premières requêtes génère un symptôme RTT_NORMAL. Lorsque le Middleware commence à se saturer, le RTT augmente pour dépasser 400 ms, ce qui génère un symptôme RTT_CRITICAL (41ème requête). La génération de ce symptôme induit l'activation de la politique critique en rejetant 40% du trafic de l'injecteur 2 et 80% du trafic de l'injecteur 3. Cette action conduit à la réduction de la charge vers la gateway IoT et permet au RTT de l'injecteur 1 de revenir à son état normal (inférieur à

300 ms) à partir de la 57ème requête, ce qui permet de lever la politique de rejet. Au bout d'un certain temps, le système retourne à l'état de saturation (179ème requête).

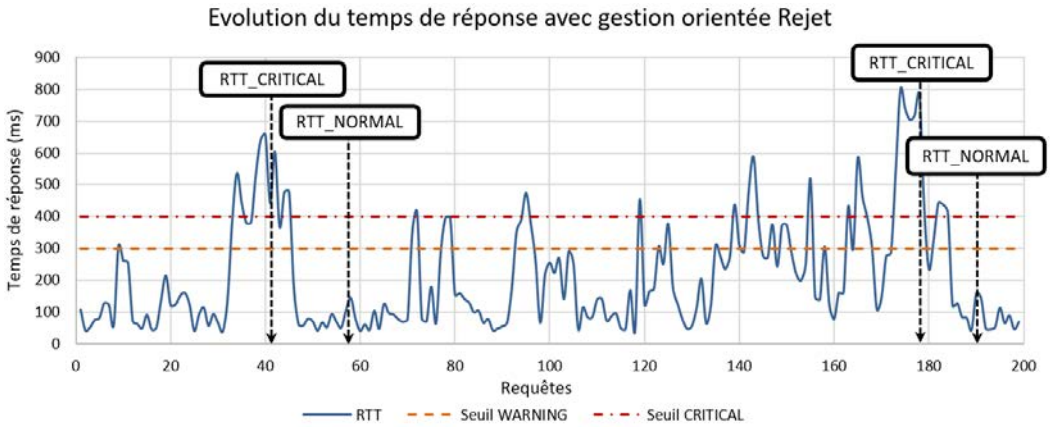

Figure 2.8 - Evolution du RTT avec une gestion orientée rejet

En terme de bénéfices, le RTT moyen du trafic critique de l'injecteur 1 est de 211 ms, ce qui est donc inférieur au seuil d'avertissement (300 ms). En ce qui concerne le coût d'intégration de cette gestion, le temps de traitement additionnel induit par le CCM et POP est de 7,992 ms. Pour la consommation des ressources, la CPU moyenne du CCM est de 24,82% et du POP est de 28,87%. La RAM est de 15,75 et 18,72 Mo pour le CCM et le POP respectivement.

Scénario orienté retardement

En prenant à présent en considération les contraintes en temps de réponse des injecteurs, le CCM attribue les priorités aux injecteurs selon la politique communiquée par le gestionnaire autonomique (Tableau 2.14).

Injecteur	Priorité
Injecteur 1	PRIORITY_DELAY_HIGH
Injecteur 2	PRIORITY_DELAY_MEDIUM
Injecteur 3	PRIORITY_DELAY_LOW

Tableau 2.14 - Politique de priorisation pour le scénario orienté retardement

Pour une gestion orientée retardement, le gestionnaire autonomique communique la politique de retardement au POP (Tableau 2.15) Cette politique tient compte du symptôme généré en fonction du RTT du trafic de l'injecteur 1. En situation normale, aucun des trafics des autres injecteurs n'est retardé. Une fois le symptôme RTT_WARNING est généré, le trafic de l'injecteur 2 est retardé de 1000 ms et celui de l'injecteur 3 de 2000 ms. Alors qu'ils sont retardés respectivement de 5000 ms et 10000 ms quand le symptômes RTT_CRITICAL est généré.

Symptôme	RTT_NORMAL	RTT_WARNING	RTT_CRITICAL
PRIORITY_DELAY_MEDIUM (ms)	0	1000	5000
PRIORITY_DELAY_LOW (ms)	0	2000	10000

Tableau 2.15 - Politique de retardement pour le scénario orienté retardement

En appliquant la politique décrite précédemment, le RTT de l'injecteur 1 suit l'évolution présentée par la Figure 2.9. Dans ce scénario, le RTT des premières requêtes génère un symptôme RTT_NORMAL. Après, lorsque le Middleware commence à se saturer, le RTT augmente pour dépasser 400 ms, ce qui génère un symptôme RTT_CRITICAL. La génération

de ce symptôme induit à l'activation par le POP de la politique critique en retardant de 5000 ms le trafic de l'injecteur 2 et de 10000 ms celui de l'injecteur 3. Le Middleware est donc déchargé et le RTT de l'injecteur 1 baisse pour atteindre l'état normal. Cet état dure un certain temps conduisant ainsi à la levée du retardement sur les injecteurs 2 et 3.

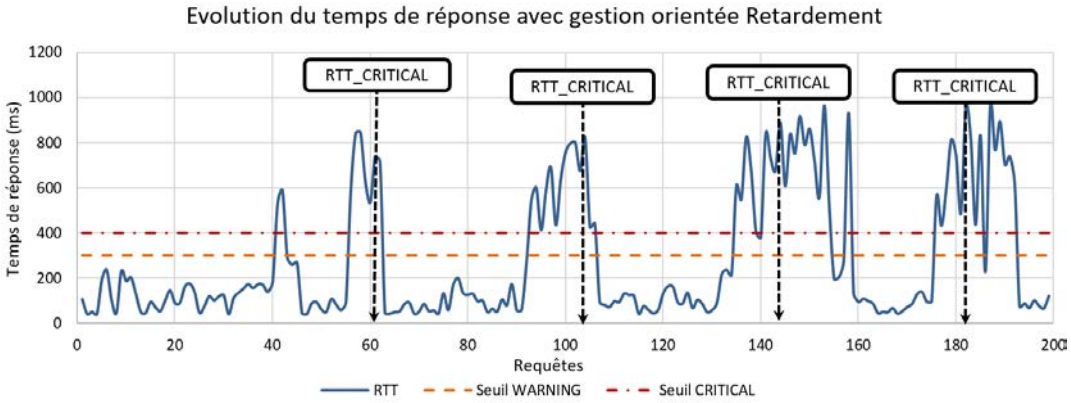

Figure 2.9 - Evolution du RTT avec une gestion orientée retardement

Dans ce scénario basé sur une politique de retardement, le RTT moyen est de 282 ms. Il est supérieur au RTT moyen obtenu dans le scénario basé rejet mais reste tout de même inférieur au seuil d'avertissement (300 ms). En termes de ressources, la consommation de la RAM par le CCM et le POP augmente très vite jusqu'à la saturation. Par contre, la consommation CPU du CCM est de 24,11 % et du POP de 28,84 %. LA RAM est alors de 17,57 et 18,67 Mo pour le CCM et le POP respectivement.

Scénario orienté ordonnancement WFQ

Dans ce scénario, la gestion est orientée ordonnancement en se basant sur des poids (WFQ) en fonction des priorités du trafic. Chaque poids reflète le nombre de requête à exécuter pour une priorité donnée avant de passer à la priorité la moins élevée. Le Tableau 2.16 représente les priorités attribuées au trafic des différents injecteurs.

Injecteur	Priorité
Injecteur 1	PRIORITY_HIGH
Injecteur 2	PRIORITY_MEDIUM
Injecteur 3	PRIORITY_LOW

Tableau 2.16 - Politique de priorisation pour le scénario orienté ordonnancement

La politique de gestion (Tableau 2.17) va dépendre du symptôme généré. Lorsque le symptôme est RTT_NORMAL, tous les injecteurs sont traités de manière similaire. L'ordonnanceur traite une requête de chaque priorité. Quand le symptôme RTT_WARNING est généré, l'ordonnanceur va traiter 5 requêtes de l'injecteur 1. Il passe ensuite à l'injecteur 2 pour traiter 3 requêtes. Par la suite, il ne traite qu'une requête de l'injecteur 3. Enfin, lorsque le symptôme RTT_CRITICAL est généré, l'ordonnanceur fait un traitement suivant les poids 10, 2 et 1 respectivement pour les injecteurs 1, 2 et 3.

Symptôme	RTT_NORMAL	RTT_WARNING	RTT_CRITICAL
PRIORITY_HIGH (requêtes)	1	5	10
PRIORITY_MEDIUM (requêtes)	1	3	2
PRIORITY_LOW (requêtes)	1	1	1

Tableau 2.17 - Politique d'ordonnancement WFQ pour le scénario orienté ordonnancement

La Figure 2.10 trace l'évolution du temps de réponse de l'injecteur 1. Comme dans les scénarios précédents, les différents symptômes générés déclenchent ainsi la politique de traitement WFQ.

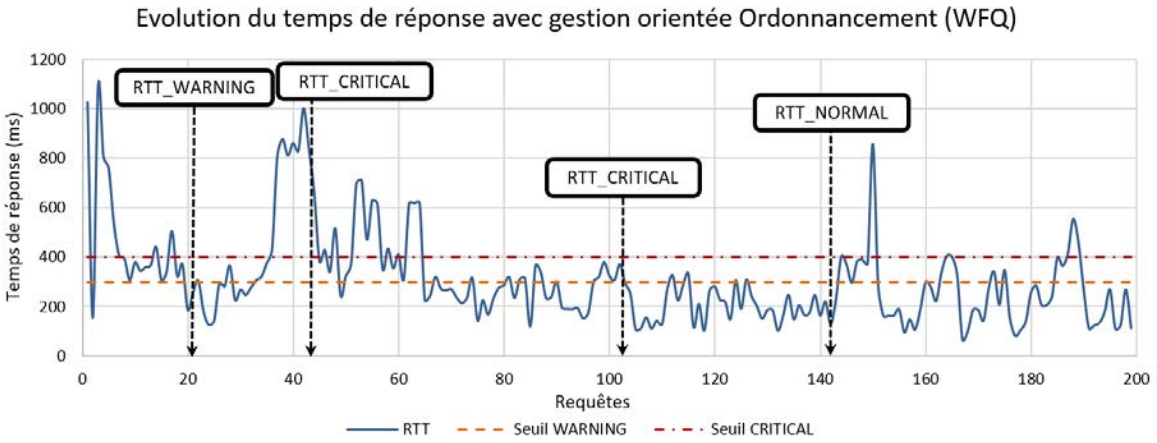

Figure 2.10 - Evolution du RTT avec une gestion orientée ordonnancement (WFQ)

En terme de bénéfices, contrairement aux scénarios précédents, le RTT moyen pour ce scénario est de 312 ms, ce qui est supérieur au seuil d'avertissement (300 ms) mais inférieur au seuil critique (400 ms). Donc, cette politique dans ce scénario donne de moins bons résultats que ceux précédents. En terme de coût, la consommation de la CPU par le CCM est de 31,75 % en moyenne. Celle du POP a une valeur moyenne de 42 %. En ce qui concerne la RAM, le CCM en consomme 20 Mo en moyenne, alors que le POP est de 14,5 Mo de consommation.

2.4. MÉCANISMES ORIENTÉS RESSOURCES

Le deuxième type de mécanismes pour la gestion de la QoS est dit orienté ressources. Les mécanismes de ce type sont inspirés de ceux utilisés dans des environnements de déploiement virtualisés tel que le cloud computing. Dans notre cas de gestion de la QoS, l'utilisation de ces mécanismes vise l'adaptation des ressources (processeurs, mémoire, disque, etc.) disponibles afin d'améliorer les performances du Middleware. Pour un système IoT, ce type de mécanismes est davantage envisageable pour le serveur IoT qui peut être déployé dans un environnement Cloud.

Nous distinguons ici entre deux approches de gestion orientée ressources, *verticale* et *horizontale*, correspondant aux approches de gestion de la *scalabilité* classiquement envisagées dans le Cloud.

2.4.1. Approche de gestion verticale

Le principe d'une gestion dite *verticale* des ressources d'une entité informatique consiste en l'ajout de nouvelles ressources informatiques (mémoire, processeurs, disques durs, etc.) afin d'améliorer ses performances. Dans notre cas, ce concept se traduit par la mise en place d'adaptateurs (Figure 2.11) permettant de gérer (ajouter ou supprimer) des ressources informatiques en fonction du besoin.

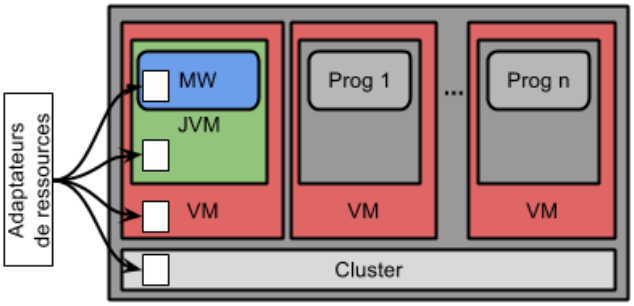

Figure 2.11 - Adaptation de ressources multi-niveaux

Par exemple, pour une plateforme Middleware basée sur le langage de programmation Java (par exemple, la plateforme OM2M), les adaptateurs gèrent les niveaux suivants :

- **ressources MW :** les composants internes de la plateforme Middleware dispose d'un certain nombre de ressources (par exemple les threads du serveur web). L'adaptateur va permettre la configuration dynamique de ces ressources afin d'améliorer les performances de la plateforme ;
- **ressources JVM :** où l'adaptateur alloue dynamiquement toute ou partie des ressources dont dispose la VM (mémoire, processeurs, disque, etc.) ;
- **Ressources VM :** à ce niveau, l'adaptateur ajuste dynamiquement les ressources informatiques allouées (mémoire, processeurs, disque, etc.) ;
- **ressources PM :** l'adaptateur gère le cluster constitué d'un ensemble de machines physiques disposant chacune de ressources telles que la mémoire, les processeurs, le disque dur, etc.

2.4.2. Approche de gestion horizontale

Le principe d'une gestion *horizontale* des ressources d'une entité informatique consiste en l'ajout de nouvelles entités ou la distribution d'une partie des composants de l'entité afin d'en améliorer les performances. Les modèles de déploiement en "Fédération-distribution" ou "Clustering-redondance" constituent deux modèles qui permettent de mettre en place cette approche. Plusieurs mécanismes peuvent être utilisés dans chaque modèle de déploiement.

Mécanismes de gestion par distribution

Le principe d'une gestion "par distribution" consiste en la subdivision de tout ou partie de l'entité considérée en plusieurs instances, et la migration d'une ou plusieurs instances sur une ou plusieurs autres machines virtuelles. Cette gestion peut, par exemple, concerner la distribution du serveur IoT en migrant dynamiquement (c'est-à-dire durant son temps d'exécution) la base de données (BD) sur une autre VM (Figure 2.12). Cette gestion est très utile dans le cas de plusieurs accès concurrents induisant des temps de réponse élevés, voire des pertes.

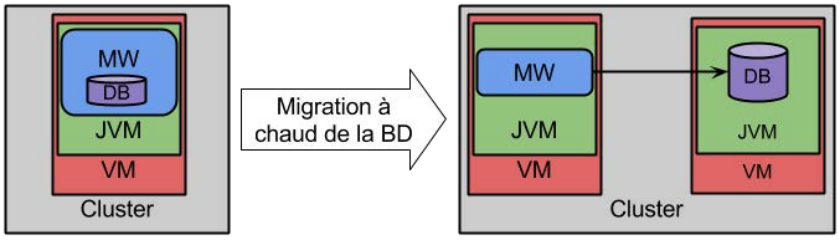

Figure 2.12 - Exemple de Migration de la BD du serveur vers une nouvelle VM

Mécanismes de gestion par redondance

Le principe d'une gestion par redondance ou par répartition de charge consiste en la création d'un ensemble d'instances du Middleware. Ces instances sont distribuées sur une ou plusieurs machines et se comportent comme une seule ressource virtuelle. Le composant de répartition de charge distribue la charge entrante sur les différentes instances (Figure 2.13).

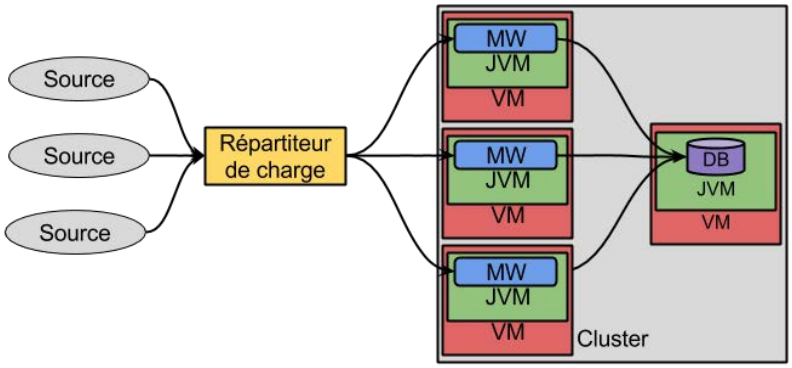

Figure 2.13 - Mécanisme d'adaptation basé sur la répartition de la charge

Tel qu'illustré la Figure 2.14, le répartiteur de charge (LB - *Load Balancer*) est constitué des éléments suivants :

- **Gestionnaire LB :** configure les composants internes en fonction de la politique de répartition ainsi que la liste des instances sur lesquelles faire la répartition ;
- **Récepteur WTP :** reçoit le trafic WTP issu des sources du trafic ;
- **Contrôleur LB :** réalise la répartition de charge issue des sources sur les entités Middleware de redondance. Il applique la politique de répartition en modifiant l'adresse destinataire par l'adresse de redirection ;
- **Redirecteur WTP :** envoie le trafic vers l'instance de redondance en fonction de l'adresse de redirection.

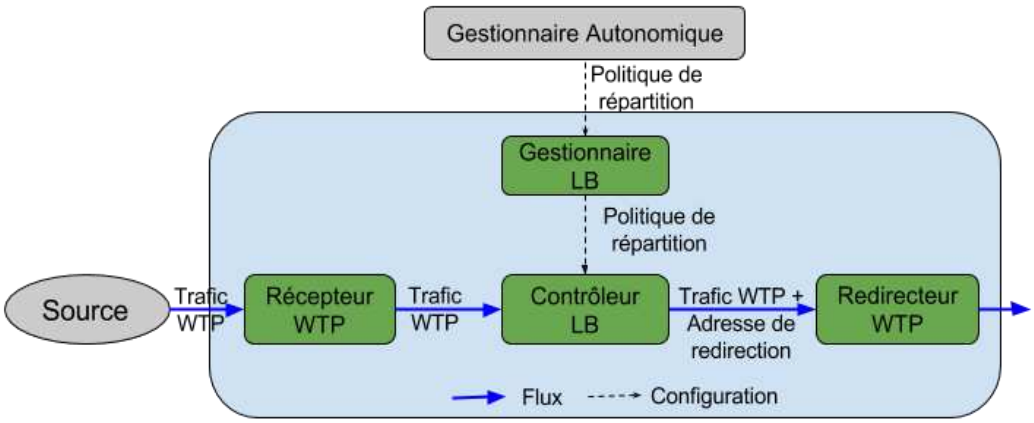

Figure 2.14 - Composition interne du composant répartition de la charge

La politique de répartition mise en œuvre par le contrôleur LB peut être guidée par plusieurs techniques, notamment :

- **équitable (ou round robin) :** la répartition de la charge (en terme de trafic) sur les instances de serveur est effectuée de façon équitable (chaque instance reçoit la même charge) ;

- **weighted round robin :** la répartition de la charge sur les instances de serveur est effectuée selon le poids attribué à chaque instance ; ce poids est susceptible d'être mis à jour durant l'exécution du système ;
- **orientée charge :** la répartition de la charge sur les instances de serveur est effectuée en fonction du pourcentage de charge courante (CPU, RAM, etc.) de ceux-ci, en visant à ne pas les saturer ;
- **orientée priorité :** la répartition se fait sur la base de la priorité du trafic. Chaque instance est donc responsable de gérer le trafic d'une priorité donnée.

2.4.3. Scénarios de validation des mécanismes orientés ressources

Dans cette étape de validation, nous nous focalisons sur l'approche de gestion horizontale par redondance basée sur la répartition de charge. Nous considérons le même émulateur que dans les scénarios orientés trafic précédents générant un trafic stochastique depuis *n* injecteurs (ici au nombre de 3), chacun *p* requêtes (ici au nombre de 900) avec une fréquence de *f* requêtes/s (ici au nombre de 60). Ce trafic est à destination d'une plateforme Middleware jouant le rôle de serveur IoT (basé sur OM2M) déployée dans un environnement Cloud.

Nous considérons trois scénarios visant à démontrer l'intérêt de la répartition de charge pour l'optimisation de la QoS :

- nous partons tout d'abord d'un premier scénario avec un seul serveur (sans répartition de charge donc) ;
- dans un deuxième scénario (illustrant à présent l'intérêt de la répartition de charge), nous considérons quatre serveurs ayant la capacité du serveur du premier scénario sur lesquels la répartition de charge se fait d'une manière équitable (25% chacun) ;
- un troisième scénario (illustrant là encore l'intérêt de la répartition de charge) est enfin joué avec trois serveurs traitant chacun un trafic de priorité donnée.

Nous nous intéressons au temps de réponse comme métrique de performance. Pour chaque scénario, nous évaluons les bénéfices induits par le répartiteur de charge mais aussi le coût en termes de temps de traitement additionnel, de consommation CPU et RAM.

Scénario 1 - Pas de répartition de charge

Dans ce scénario (Figure 2.15), nous prenons un serveur surdimensionné accueillant le trafic des 3 injecteurs.

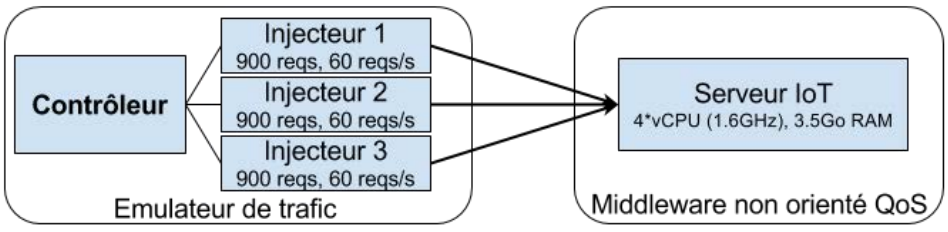

Figure 2.15 - Architecture du scénario sans répartition de charge

Le serveur dispose d'une capacité de 4*vCPU ayant chacun une fréquence de 1,6 GHz et 3,5 Go de RAM. La Figure 2.16 représente l'évolution de son temps de réponse.

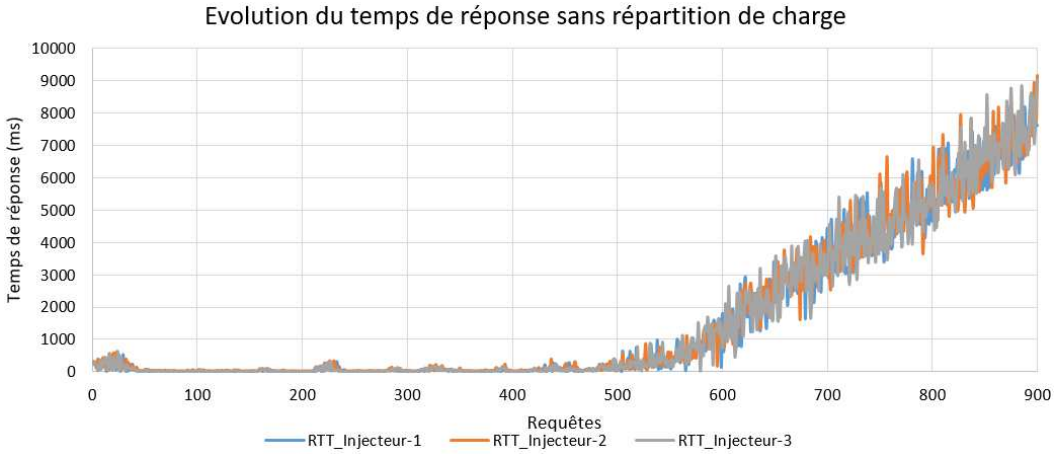

Figure 2.16 - Evolution du temps de réponse des injecteurs sans répartition de charge

En présence du trafic des autres injecteurs, le serveur IoT, même s'il est surdimensionné, sature au bout de 500 requêtes et conduit à une dégradation du temps de réponse de l'injecteur 1 (RTT_{Moy} = 1573,45 ms). Cette saturation est causée par la surcharge de la CPU (Figure 2.17) qui atteint en moyenne 73,2% (CPU_{Moy}) alors que la mémoire consommée reste faible (RAM_{Moy} = 355 Mo).

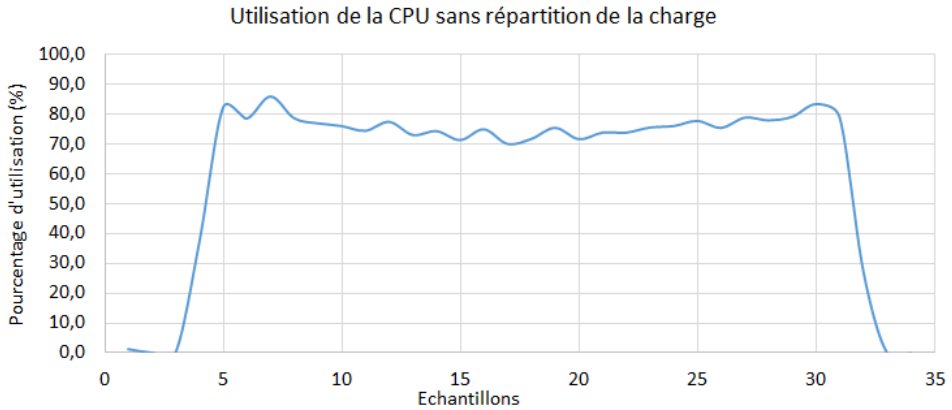

Figure 2.17 - Consommation de la CPU par le serveur sans répartition de charge durant le scénario

Pour ce scénario, nous pouvons conclure que, même en sur-dimensionnant le serveur, la présence d'autres trafic peut induire une dégradation de la QoS pour un trafic critique.

Scénario 2 - Répartition de charge équitable (Round Robin)

Dans ce scénario, nous prenons la capacité du précédent serveur réparti sur 4 serveurs (Figure 2.18). La répartition de charge entre ces serveurs se fait d'une manière équitable (25 % chacun) à l'aide d'un répartiteur de charge.

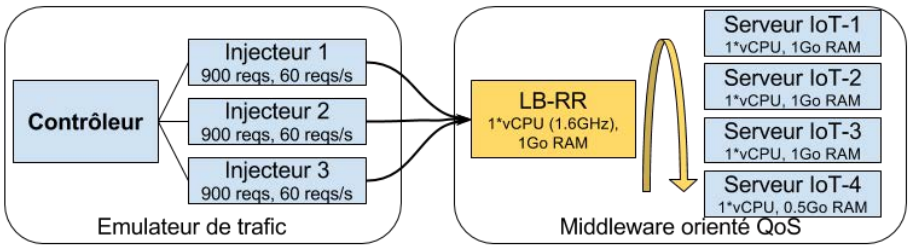

Figure 2.18 - Architecture du scénario de répartition de charge équitable

A l'aide de cette répartition de charge, le temps de réponse du trafic issu des trois injecteurs est amélioré comparativement au premier scénario (RTT_{Inj1} = 33,827 ms, RTT_{Inj2} = 34,314 ms, RTT_{Inj3} = 31,696 ms) et ne suit plus une évolution exponentielle (Figure 2.19).

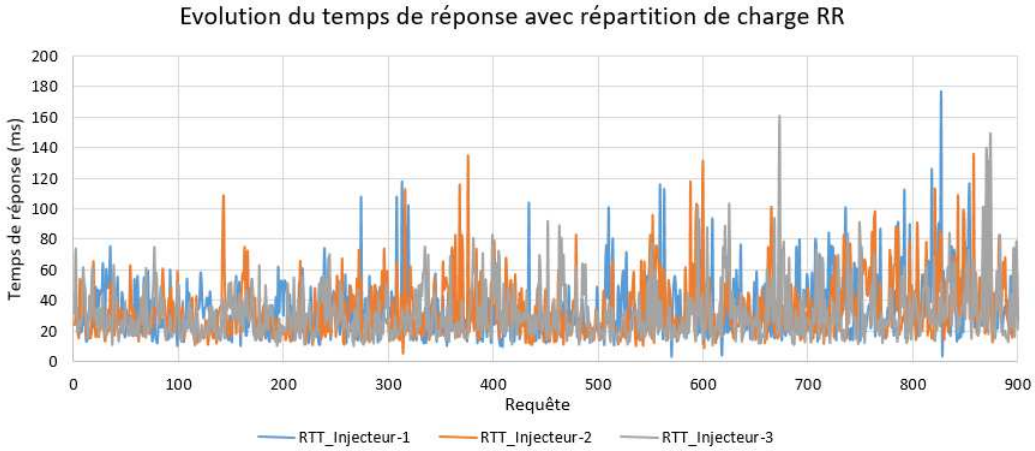

Figure 2.19 - Evolution du temps de réponse avec répartition de charge équitable

Ceci est dû au fait que le nombre de requêtes total que reçoit chaque serveur est moins élevé et ne conduit donc pas à une saturation. Le serveur 1 consomme 43,92% de sa CPU et 52,39 Mo de sa RAM. Le serveur 2 utilise 52,67% de sa CPU et 42,38 Mo de sa RAM en moyenne. Le serveur 3 est à 46,45% de CPU et 43,12 Mo de RAM. Enfin le serveur 4, qui est le moins puissant, consomme 52,68% de CPU et 37,93 Mo de RAM. Le coût induit par l'ajout du répartiteur de charge est de 0,89 ms en terme de temps de traitement additionnel, 31,1% de consommation CPU et 33,47 Mo de consommation de la mémoire.

Pour ce scénario, en se basant sur un répartiteur de charge basé Round Robin, le temps de réponse moyen subit une nette amélioration. Nous pouvons donc constater que la répartition conduit à de meilleurs résultats. Cependant, le trafic issu des différents injecteurs est traité de manière identique sans aucune distinction sur le RTT quelle que soit la priorité du flux ($RTT_{Inj1} \approx RTT_{Inj2} \approx RTT_{Inj3} \approx RTT_{MoyRR}$ = 33,279 ms).

Dans le scénario suivant, le trafic est traité par le répartiteur de charge selon sa priorité, l'optique étant d'observer comment ceci va impacter le temps de réponse de l'injecteur 1 face au trafic des autres injecteurs.

Scénario 3 - Répartition de charge orientée priorité

Dans ce scénario, le répartiteur de charge redirige le trafic vers un serveur donné en fonction de sa priorité. Comme représenté par la Figure 2.20, nous disposons de 3 injecteurs ayant chacun une priorité donnée (Injecteur 1 = HIGH, Injecteur 2 = MEDIUM, Injecteur 3 = LOW). Il y a donc 3 serveurs de capacité différente (serveur 1 (HIGH, 2 vCPU, 2 Go RAM), serveur 2 (MEDIUM, 1 vCPU, 1 Go RAM) et serveur 3 (LOW, 1 vCPU, 521 Mo RAM)) mais ayant au total la même capacité du serveur du premier scénario.

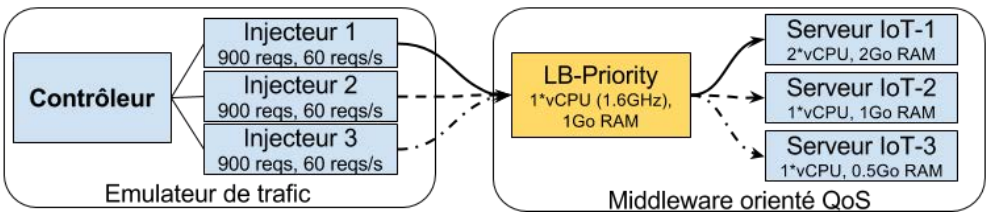

Figure 2.20 - Architecture du scénario de répartition de charge orientée priorité

En adoptant cette politique de répartition, le temps de réponse moyen (Figure 2.21) du trafic de l'injecteur 1 est de 25,4 ms, celui de l'injecteur 2 est de 33,11 ms, et celui de l'injecteur 3 est de 58 ms. Comparativement au scénario précédent, nous observons donc une amélioration du temps de réponse de l'injecteur 1, celui de l'injecteur 2 reste presque identique, et celui de l'injecteur 3, le moins prioritaire, se détériore.

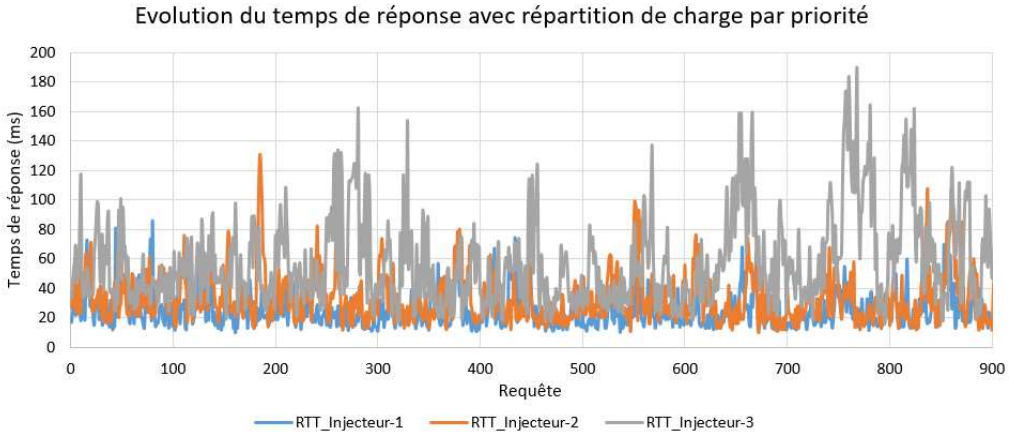

Figure 2.21 - Evolution du temps de réponse des injecteurs avec répartition de charge par priorité

En terme de coûts, le répartiteur de charge induit un temps de traitement additionnel de 0,48 ms, ainsi qu'une consommation de 23,9% pour la CPU et de 38,84 Mo pour la RAM en moyenne. Les serveurs consomment successivement en moyenne : CPU_{MoyS1} = 49,8%, RAM_{MoyS1} = 152,23 Mo, CPU_{MoyS2} = 75,7%, RAM_{MoyS2} = 39,59 Mo, CPU_{MoyS3} = 93%, RAM_{MoyS3} = 54,31 Mo.

L'adoption d'une politique basée priorité par le répartiteur de charge permet donc une meilleure prise en considération des priorités de chaque trafic sur les ressources disponibles ; elle permet donc de favoriser le trafic le plus prioritaire au détriment du trafic qui l'est moins ($RTT_{Inj1} < RTT_{Inj2} < RTT_{Inj3}$).

2.5. CONCLUSION

Dans ce chapitre, nous avons présenté deux principaux types de mécanismes pour la gestion de la QoS, orienté trafic et orienté ressources qui sont adaptés aux différents environnements de déploiement, physique ou virtuel. Les mécanismes orientés trafic sont inspirés des approches utilisées au niveau des couches IP et Transport de l'architecture de l'Internet, notamment l'approche DiffServ. Ces mécanismes permettent de gérer le trafic en fonction de sa priorité. Ils sont basés sur deux composants placés en amont du Middleware destinataire. Le composant de classification et de marquage (CCM) permet de classer et de marquer le trafic avec une priorité donnée en fonction de plusieurs critères. Le Proxy orienté Priorité (POP) qui gère ce trafic en fonction de sa priorité. La gestion se base sur trois fonctionnalités essentielles : le rejet, le retardement et l'ordonnancement. La politique de gestion du trafic par ces composants est fournie par le gestionnaire autonomique.

Les mécanismes orientés ressources reposent sur deux approches de gestion. La première approche, dite *verticale,* cible une gestion des ressources (mémoire, processeurs, etc.) du Middleware par leur ajout ou suppression en fonction de la charge du trafic et des performances qui en résultent. La deuxième approche, dite *horizontale*, est basée sur un modèle de distribution ou duplication. La distribution se fait, par exemple, de certains composants du Middleware sur une ou plusieurs machines virtuelles. Alors que la duplication se fait en créant

plusieurs instances du Middleware et en réalisant une répartition de charge entre elles selon une politique donnée (round robin, WRR, orientée charge ou orientée priorité).

Pour chaque type de mécanismes, des scénarios de validation ont été réalisés. Ils sont basés sur une plateforme d'émulation qui permet de générer du trafic HTTP ou CoAP à destination du Middleware depuis une ou plusieurs sources. Cette plateforme donne le choix de la méthode à utiliser (GET, POST, PUT et DELETE) mais aussi le type de trafic à générer (en rafale, périodique et stochastique). Ces scénarios ont été réalisés pour illustrer les bénéfices induits par chaque type de mécanismes en contraste avec leurs coûts (temps de traitement additionnel, consommation de la CPU et de la RAM).

Dans le chapitre suivant, nous spécifions et concevons l'architecture de gestion de ces mécanismes. Cette architecture inclut notamment les gestionnaires autonomiques responsables du déploiement et la gestion des mécanismes présentés dans ce chapitre. Ces gestionnaires autonomiques prennent en considération le contexte de déploiement du Middleware (machine virtuelle ou physique) afin d'utiliser les mécanismes adéquats pour aboutir au respect de la QoS de bout en bout.

Chapitre 3

Architecture de gestion autonomique et hiérarchique de la QoS

Contenu

3.1. INTRODUCTION

Pour aboutir à une maîtrise de la QoS de bout en bout au niveau Middleware de l'IoT suivant l'approche exposée au chapitre 1, un système de gestion d'actions d'adaptation appropriées (telles que celles proposées via les mécanismes présentés au chapitre 2) est nécessaire. Ce système doit être capable d'exécuter ces actions sans interruption du service fourni aux utilisateurs finaux. Afin de minimiser les erreurs et d'intervenir dans des temps compatibles avec les besoins de l'activité métier, il doit également être capable de réaliser ces actions avec un minimum d'intervention de l'administrateur du système ; en d'autres termes, il doit idéalement être doté de capacités d'autogestion vis-à-vis des choix liés à l'exécution, au final, des actions d'adaptations appropriées. Concevoir l'architecture d'un tel système est une problématique à part entière que nous traitons dans ce chapitre.

Tel que présenté dans l'état de l'art du chapitre 1, les solutions orientées QoS au niveau Middleware ne portent pas directement sur le Middleware, mais s'en servent pour relayer les besoins exprimés vers les niveaux des entités réseaux et équipements sous-jacents. Pour autant, il existe, hors contexte IoT, des solutions de gestion de la QoS au niveau Middleware. Dans des domaines tels que l'aéronautique, la simulation interactive peut reposer sur le Middleware DDS [dds04]. Pour ce dernier, les attributs de QoS sont fixés au démarrage du système et sont difficilement modifiables au long de l'exécution du Middleware. La gestion de la QoS ainsi proposée revêt un caractère statique et en conséquence non adaptatif. Dans le domaine de

l'Internet des services (au sens SOA), un travail plus conséquent a été opéré, notamment au niveau des bus de service [esb02], pour aboutir à une gestion dynamique et auto-adaptative de la QoS.

Ces derniers travaux inspirent en partie notre approche de gestion, la différence de contexte induisant cependant des problématiques différentes : premièrement, au niveau de la structuration qui est différente, à savoir un seul bus de service contre un serveur IoT et plusieurs gateways IoT, induisant donc un caractère distribué des entités Middleware ; ensuite, de part une dynamicité plus forte dans l'IoT de par l'apparition / disparition d'objets / flux générés ; enfin, par les différences d'approches d'implémentations (orientée services vs. orientée ressources) induisant des conséquences différentes sur les performances du système géré.

Dans ce contexte, notre contribution consiste en la spécification et la conception de l'architecture (ici structurelle) d'un système de niveau Middleware de l'IoT. Cette architecture intègre et étend les fonctionnalités usuelles des Middlewares basés sur les standards ETSI M2M ou oneM2M, par des fonctionnalités de gestion de la QoS. Le système, intitulé IoT-Q, permet ainsi la configuration et l'adaptation dynamique des mécanismes de gestion de la QoS présentés dans le chapitre 2 précédent. IoT-Q intègre également le paradigme de l'*Autonomic Computing* [kep03] pour viser au final une capacité d'auto-adaptation dans la gestion des mécanismes précédents.

L'architecture du système IoT-Q (également appelé architecture IoT-Q dans la suite) est basée sur un principe de *politiques hiérarchiques* dans la gestion de ces actions, inspirée de propositions antérieures pour la gestion du réseau [fle01]. Elle intègre ainsi deux niveaux d'actions : des actions *stratégiques* conduisant à l'élaboration et l'adaptation *d'objectifs de QoS locale* à atteindre par chacune des entités Middleware impliquées dans le chemin de données ; et des actions *opérationnelles* conduisant à l'adaptation du *comportement* ou des *ressources* des entités Middleware pour atteindre l'objectif de QoS locale fixé par le niveau supérieur.

La suite de ce chapitre est structurée en deux sections. La section 3.2 précise l'approche d'élaboration de l'architecture IoT-Q, basée sur le paradigme de l'*Autonomic Computing* et sur le principe de politiques hiérarchiques. La section 3.3 présente la méthodologie adoptée pour conduire la conception de l'architecture ainsi que son application au sujet étudié. La conception proposée s'inscrit dans le cadre de l'approche *Model Driven Architecture* (MDA) [sol00], avec un focus sur le niveau *Platform Independent Model* (PIM), et suit une méthodologie basée sur le formalisme UML 2.0. Cette méthodologie conduit à identifier les besoins fonctionnels et non fonctionnels du système à concevoir, les acteurs et cas d'utilisation, les diagrammes de structure composite de l'architecture et de ses principaux composants, les interactions entre ses composants, et enfin les détails structurels et comportementaux (diagrammes de séquence) de certains de ces composants. Les conclusions et perspectives de ce chapitre sont finalement présentées en **section 3.4**.

3.2. APPROCHE D'ÉLABORATION

Notre approche de gestion de bout en bout de la QoS vise à doter le système géré de deux propriétés : la dynamicité et l'auto-adaptation. Plusieurs approches permettent d'aboutir à une gestion adaptative telles que la boucle MAPE-K proposée par IBM [act04], GANA [cha08, cpk09], 4D architecture [gre05], CONMan management model [bal07], FOCALE [str06] ou encore un plan de connaissance pour l'Internet [cla03]. Dans cette thèse, nous nous intéressons principalement au paradigme de l'Autonomic Computing proposée par IBM. Ce paradigme propose les composants fonctionnels permettant d'aboutir à l'autogestion du système. Il fournit

aussi les niveaux de maturité permettant de mener un système non géré à devenir un système autonomique. Dans cette section, nous commençons tout d'abord par la présentation de ce paradigme, ses composants fonctionnels et ses niveaux de maturité (section 3.2.1). Nous présentons ensuite notre approche pour la proposition d'une architecture autonomique et hiérarchique pour la gestion de bout en bout de la QoS au niveau Middleware (section 3.2.2).

3.2.1. Paradigme de l'Autonomic Computing

La gestion manuelle du besoin en QoS au sein d'un système de communication tel que celui considéré dans nos travaux est une tâche complexe pour son administrateur. Celui-ci doit faire face à une évolution dynamique, c'est-à-dire durant l'exécution du système, des besoins en QoS des utilisateurs du système et des ressources disponibles au niveau de son infrastructure de déploiement.

En réponse à cette difficulté, l'Autonomic Computing est un paradigme clé introduit en 2003 par IBM [kep03] pour prendre en compte cette dynamicité de manière autonome sur la base de stratégies (ou politiques) de haut niveau fournies par l'administrateur. Idéalement un système autonome a ainsi la capacité de définir le choix des règles opérationnelles à appliquer en fonction des politiques ciblées, des performances courantes du système et de son contexte d'exécution.

La gestion du système se fait par un *gestionnaire autonomique* (*Autonomic Manager*) via des points de contact (*touchpoints*). Un gestionnaire autonomique comporte cinq composants fonctionnels principaux identifiant ce qui est appelé la boucle MAPE-K [kep03]. Ces composants (Figure 3.1) sont :

- **le Moniteur** : supervise le système géré par le biais de capteurs logiques permettant d'observer leurs performances actuelles. Basée sur des règles de traitement (filtrage, corrélation et agrégation par exemple) des événements de supervision, la détection d'une infraction (valeur anormale) vis-à-vis par exemple d'un besoin en QoS, engendre la génération d'un/des symptômes (équivalent à des alarmes) vers le composant d'analyse ;
- **l'Analyseur** : analyse les symptômes dans l'objectif d'identifier les causes potentielles de leur génération. Cette analyse se base sur l'interrogation directe du système géré ou sur la base d'informations stockées dans la base de connaissance décrite ci-après. Si un changement est nécessaire, une requête de changement (RFC - *Request For Change*) est envoyée au planificateur ;
- **le Planificateur** : planifie les actions à réaliser pour atteindre les objectifs exprimés sous forme de politiques. Il établit ou sélectionne les plans à exécuter au niveau de l'entité gérée ;
- **l'Exécuteur** : exécute le plan élaboré par le planificateur via les *effecteurs* présents au niveau de l'entité gérée ;
- **la Base de connaissance** : elle permet de stocker les informations de gestion telles que les symptômes, les politiques, la représentation de l'entité gérée, les métriques à collecter, etc.

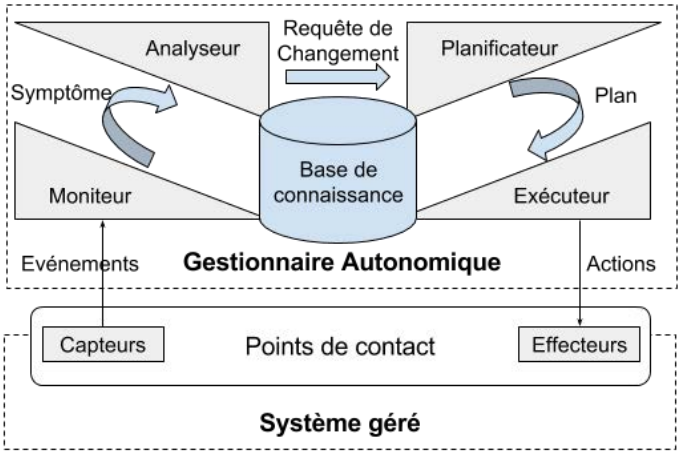

Figure 3.1 - Boucle MAPE-K du paradigme de l'Autonomic Computing

L'implémentation d'un système autonomique, implémentant la boucle MAPE-K, est une tâche complexe qui nécessite le passage par cinq niveaux de maturité [act04] (Figure 3.2) :

- **Niveau 1 – Basique** : la gestion et la configuration des éléments du système sont faits indépendamment par l'administrateur. Les compétences humaines sont alors nécessaires pour superviser le système, analyser les métriques observées et éventuellement exécuter des actions en fonction des anomalies détectées ;
- **Niveau 2 – Géré** : des techniques et outils de supervision sont utilisés afin de collecter les métriques depuis le système pour détecter les anomalies, aidant ainsi à réduire le temps de la collecte et la synthèse des informations. Les compétences humaines sont nécessaires pour l'analyse des anomalies détectées et l'exécution des actions correctives ;
- **Niveau 3 – Prédictif** : les capacités de diagnostic et d'analyse sont introduites dans le système afin d'analyser les situations et fournir les actions correctives possibles. Ici, la décision finale ainsi que l'activation des actions dépendent de l'administrateur ;
- **Niveau 4 – Adaptatif** : l'administrateur n'a plus à approuver les actions correctives et à les activer. Il doit simplement définir les politiques en se basant sur la corrélation entre les symptômes et les mécanismes afin de permettre à l'environnement adaptatif de décider automatiquement des actions adéquates en se basant sur les informations disponibles et sur la connaissance de ce qui se passe dans l'environnement ;
- **Niveau 5 – Autonomique** : à ce niveau final, les composants du système sont intégrés et autogérés de manière dynamique en accord avec les objectifs métier (business rules). Le niveau Autonomique permet ainsi à l'administrateur (au sens large) du système de ne se focaliser que sur les exigences métier requises. La politique métier devient le principal moteur de la gestion du système.

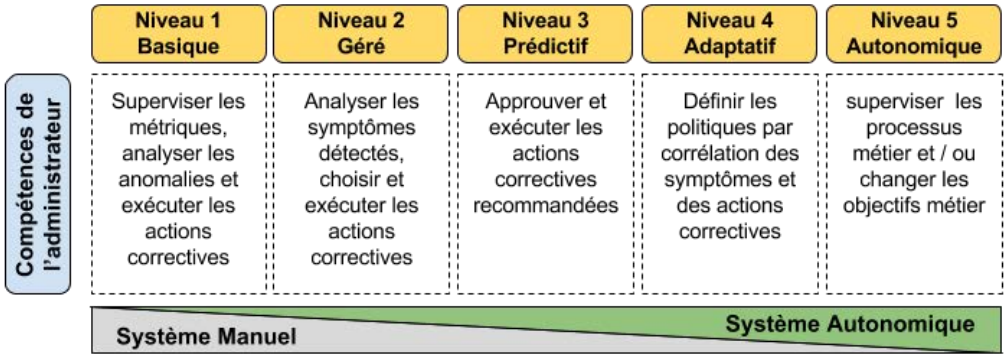

Figure 3.2 - Niveaux de Maturité pour un système autonomique

Pour le système de gestion de la QoS proposé dans le cadre de nos travaux, nous nous plaçons au niveau de maturité Adaptatif. Nous définissons les modèles, les règles et les mécanismes pour la mise en place du gestionnaire autonomique à ce niveau de maturité.

3.2.2. Approche de gestion autonomique et hiérarchique du système IoT

L'interconnexion des entités de niveau Middleware (typiquement le serveur et les gateways IoT) permet la communication entre les applications et les équipements en plus d'autres services (stockage, souscription et notification, etc.). Ces applications peuvent avoir des besoins en QoS qui doivent être pris en considération, notamment au niveau des entités Middleware. Ces entités peuvent en effet constituer des goulots d'étranglement qui dégradent le délai des requêtes applicatives, ce qui pose problème lorsqu'il s'agit d'applications IoT critiques, sensibles au temps de traitement des requêtes. Notre approche générale considère deux niveaux d'actions pour la gestion de bout en bout de la QoS au niveau Middleware : des *actions opérationnelles* (chapitre 2) qui vont conduire à l'adaptation du comportement ou des ressources des entités Middleware pour atteindre un objectif local de QoS ; et des *actions stratégiques* qui vont conduire à l'adaptation des objectifs de QoS locaux à atteindre par chacune des entités Middleware impliquées dans le chemin de données afin de répondre aux objectifs de bout en bout requis par l'application.

L'élaboration de ces actions se fait de manière autonome via le paradigme de l'Autonomic Computing. La conception et l'implémentation de la solution peut s'aborder de manière classique en proposant un gestionnaire autonomique (AM - Autonomic Manager) global pour la gestion de la QoS. Celui-ci assure cette gestion en collectant les métriques depuis l'ensemble des entités Middleware impliquées sur le chemin emprunté par les requêtes et leurs réponses. L'implémentation de ce gestionnaire global peut se faire d'une manière centralisée possédant une visibilité globale de l'ensemble des entités Middleware du système (Figure 3.3). Cependant, le nombre de métriques collectées, de diagnostic et de prise de décisions peut augmenter significativement et constituer en lui-même une source de problème si le nombre d'entités IoT augmente significativement dans un système IoT évolutif. Pour que le système de gestion de la QoS soit résistant à ce facteur d'échelle, une autre approche consiste en la distribution de la gestion sur plusieurs gestionnaires autonomiques. Avec une telle distribution, plusieurs questions se posent alors, relevant notamment de la politique de distribution, de son coût vis-à-vis des performances du système, des entités qui vont être gérées par chaque AM, de la façon de définir les objectifs pour respecter le besoin applicatif de bout en bout, et enfin, de la façon de garantir la coordination entre ces AM distribués. Une conception basée sur un principe de distribution des AM soulève donc plusieurs questions et fait apparaître le besoin de coordonner des politiques locales pour la mise en place des actions stratégiques.

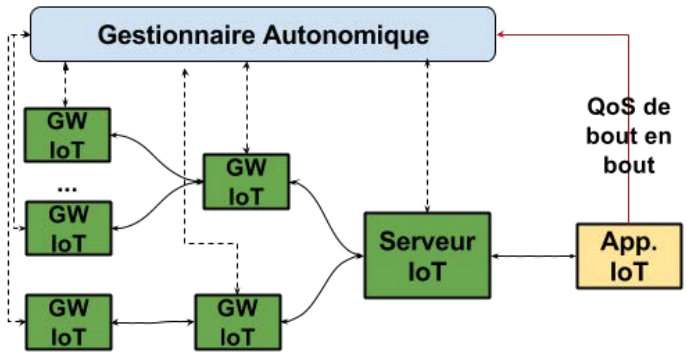

Figure 3.3 - Gestion globale avec implémentation centralisée

L'approche d'application de ce principe que nous proposons repose sur un principe de gestion hiérarchique de la QoS. Cette vision s'inspire de ce qui a été proposé pour la gestion du réseau [fle01]. Pour des systèmes complexes, la notion de hiérarchie permet d'appliquer un certain nombre de concepts clefs [mof93], tels que le raffinement d'objectifs, le partitionnement de pôles d'actions ou encore la délégation de responsabilité vis-à-vis de la façon de satisfaire chaque niveau d'objectif. Dans notre vision, nous distinguons deux niveaux de hiérarchie prenant en compte chacun des niveaux de politiques (Figure 3.4) :

1. le premier niveau est dédié aux actions opérationnelles. La gestion est assurée de manière autonomique via un AM dit *esclave* (AMS - *AM Slave)*. Chaque AM est spécifique à une entité Middleware pour la gestion locale de la QoS. Il assure la supervision et l'adaptation du comportement du Middleware vis-à-vis d'un objectif local à travers des actions de reconfiguration basées sur les mécanismes de QoS proposés dans le chapitre précédent ;
2. le deuxième niveau assure les actions stratégiques. Il est dédié à la coordination dans une optique de cohérence des objectifs à atteindre par les AM locaux. Dans notre vision, l'élaboration de ces politiques est envisagée de façon autonomique étant donné que nous considérons un modèle de gestion de QoS de bout en bout (dépendant donc de ce qui va se passer au niveau de chaque entité Middleware sur le chemin donné). La gestion se fait via un AM dit *maître* (AMM - *AM Master*) qui prend en compte les besoins applicatifs en QoS de bout en bout, définit les objectifs de QoS locaux à atteindre par chaque AM local, et adapte ces objectifs en fonction des résultats atteints par chaque AM.

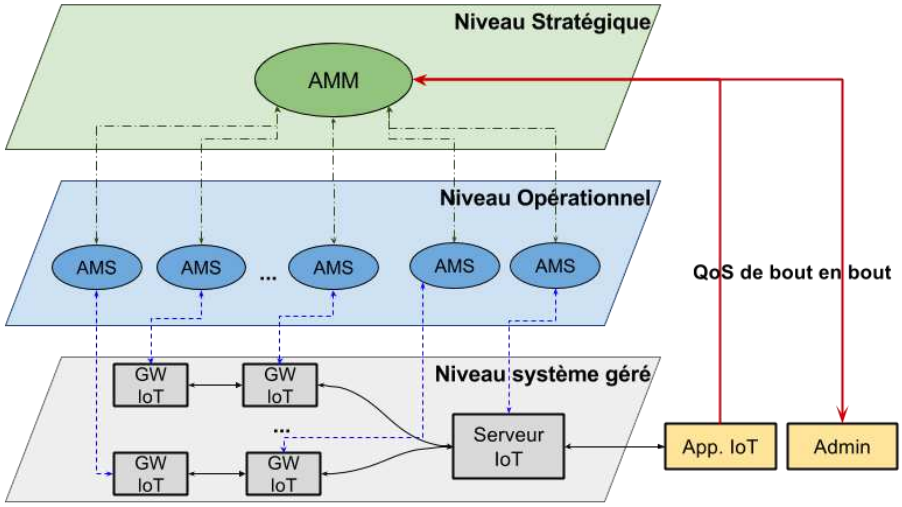

Figure 3.4 - Architecture de gestion autonomique hiérarchique

Dans ce qui suit, la spécification et la conception du système de gestion autonomique et hiérarchique de la QoS de bout en bout est traitée en détails. Ce système prend en compte les deux niveaux de politiques opérationnel et stratégique.

3.3. SYSTÈME DE GESTION AUTONOMIQUE ET HIÉRARCHIQUE

Nous proposons un système de gestion autonomique de la QoS de bout en bout intitulé IoT-Q. Ce système est défini de manière générique indépendamment des plateformes Middleware existantes. Il propose les services IoT traditionnels offerts par le Middleware ainsi que des services de prise en compte et gestion de la QoS de bout en bout. La section 3.3.1 présente tout

d'abord l'approche d'élaboration du système guidée par les modèles (MDA) ainsi que les exigences du système en termes de besoins fonctionnels et non fonctionnels. La section 3.3.2 expose les services fournis par le système, son architecture fonctionnelle de haut niveau et celle de ses composants internes. Elle précise également les interactions entres les composants internes pour fournir les différents services.

3.3.1. Approches d'élaboration et exigences du système

Architecture guidée par des modèles

Afin d'élaborer le système d'autogestion instanciable pour toute implémentation d'un Middleware IoT, l'approche de spécification et-de conception guidée par les modèles (MDA - *Model Driven Architecture*) [sol00] a été suivie. Elle permet d'aboutir à une architecture instanciable suivant différentes techniques d'implémentation. Elle rend ainsi possible la portabilité et la réutilisabilité de l'architecture. Les modèles décrivent de manière générique la spécification du système, les opérations offertes par le système, ses composants internes et les interactions entre elles, ainsi que les techniques requises pour le déploiement du système sur différentes plateformes.

Les principaux modèles de l'approche MDA sont :

- **CIM (Computation Independent Model) :** modèle d'analyse de base indépendant de calcul. Il décrit les fonctionnalités et cas d'utilisation du système indépendamment de tout système informatique ;
- **PIM (Platform Independent Model) :** modèle de conception indépendant des plates-formes de déploiement. Il consiste en des diagrammes, essentiellement en UML (Unified Modeling Language), décrivant la structure des entités internes du système et les services indépendamment de la technologie de déploiement de l'application ;
- **PDM (Platform Independent Model) :** modèle dépendant des plates-formes. Il contient les informations de transformation des modèles permettant le passage du PIM vers le PSM ;
- **PSM (Platform Specific Model) :** modèle spécifique à la plateforme. Il décrit les détails techniques liés à l'implantation pour une plate-forme d'exécution donnée ainsi qu'à la génération du code exécutable.

Dans ce chapitre, nous nous positionnons au niveau PIM pour la description de l'architecture du système IoT-Q. Nous élaborons dans la suite les acteurs du système, les services fournis, ainsi que les composants internes et les interactions entre eux.

Besoins fonctionnels du système

Les différents services offerts par le système sont fournis aux acteurs suivants :

- **Application IoT :** définie comme un ou plusieurs programmes s'exécutant sur une ou plusieurs machines clientes. Une application IoT a pour objectif de fournir un service métier au client (domotique, télésurveillance, voiture intelligente, ville intelligente, etc.). Une application IoT est dite QoS-aware quand elle est capable d'exprimer explicitement ses besoins en QoS au système IoT-Q, et éventuellement de prendre en compte les commandes orientées QoS issues du système. Une application IoT est dite QoS-unaware quand elle est incapable de communiquer explicitement ses besoins au

système IoT-Q ; dans ce cas, l'administrateur doit définir la stratégie avec laquelle les besoins en QoS de ce genre d'applications doivent être gérées ;
- **Administrateur :** défini comme une personne physique utilisant son terminal afin de démarrer, arrêter, configurer l'ensemble des composants du système IoT-Q ;
- **Développeur de mécanismes orientés QoS :** défini comme une personne physique ayant la capacité de développer des mécanismes orientés QoS, et de les introduire dans le système IoT-Q afin de lui doter de capacités de gestion des besoins en QoS des applications IoT ;
- **Développeur de modèles :** défini comme une personne physique utilisant son terminal pour enrichir le système de modèles de gestion de la QoS.

Besoins non fonctionnels du système

Exigences de conception

Afin de mener à bien les phases d'analyse et de conception, nous avons choisi d'utiliser la méthodologie *Unified Process* basée sur le langage UML 2.0 [omg06]. Cette méthodologie permet d'analyser la spécification des exigences du système et d'identifier les principaux services qui seront fournis par le système. De cette analyse, les cas d'utilisation textuels sont construits afin de décrire la façon dont les différents sous-systèmes et composants collaborent pour mettre en œuvre les différents services. En outre, les principales interfaces permettant la communication entre ces entités internes sont également à identifier. Ce processus permet ainsi de proposer la conception de haut niveau de l'architecture du système, incluant ses services et ses fonctionnalités internes.

Choix technologiques et interopérabilité

La compatibilité multiplateforme (au sens système d'exploitation) est une condition importante pour un système tel que IoT-Q. Les langages multiplateformes telles que Java seront utilisés pour mettre en œuvre la plupart du code à exécuter sur les entités de gestion (implémentant la boucle MAPE-K).

Afin de garantir l'interopérabilité de la solution, un ensemble d'interfaces uniformes, standardisées et de haut niveau doivent être fournies. L'objectif étant de permettre aux développeurs et à l'administrateur d'intégrer, de déployer et d'étendre facilement le système IoT-Q en composant, réutilisant et rendant portable et interopérable les composants du système.

3.3.2. Spécification et Conception du Système IoT-Q

Partant de l'architecture de haut niveau décrite précédemment, cette section fournit une description de l'architecture fonctionnelle de haut niveau du système. Suivant la méthodologie de conception adoptée, nous présentons successivement : les cas d'utilisation illustrant les services fournis aux utilisateurs ; les composants fonctionnels du système ; et les interactions entre les acteurs et les composants internes du système.

Services fournis par le système

Cette section décrit les cas d'utilisation textuels illustrant les services fournis aux acteurs externes par le système IoT-Q. Ces cas d'utilisation sont synthétisés sur la Figure 3.5.

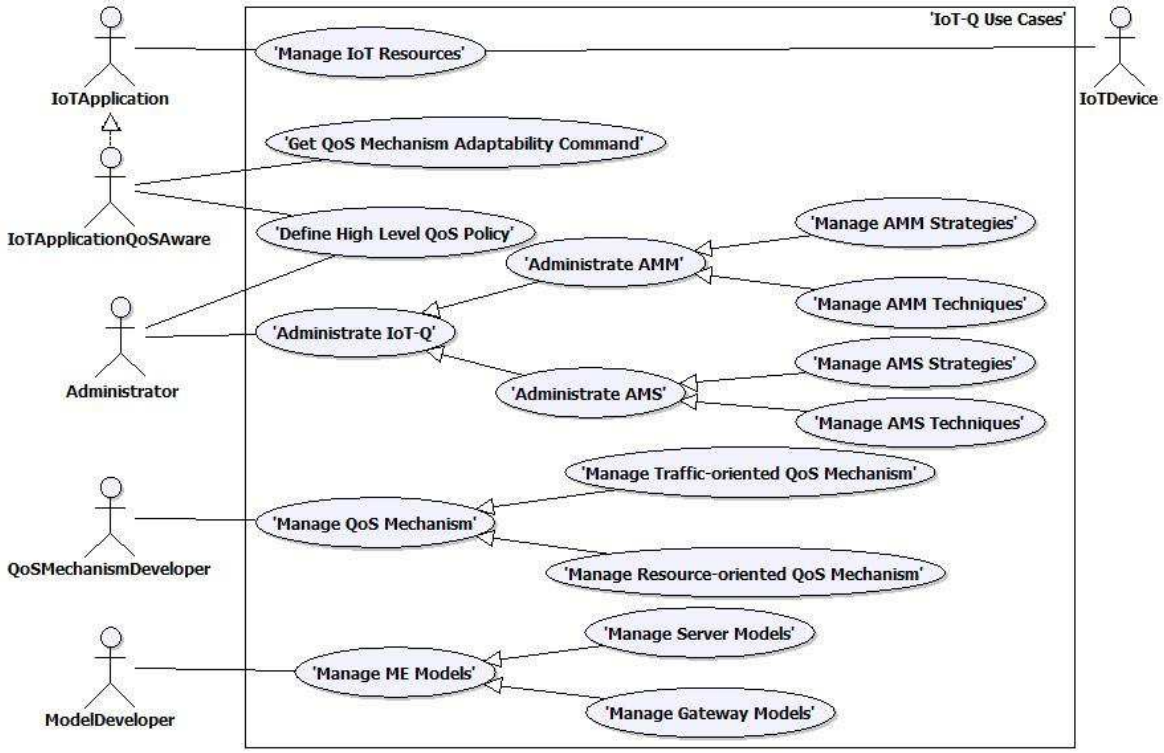

Figure 3.5 - Cas d'utilisation du système IoT-Q

Service "Manage IoT Resources"

Ce service permet aux applications de gérer les ressources des équipements via une API REST. Cette gestion passe par la possibilité de créer, récupérer, modifier ou supprimer des ressources IoT. Ces ressources peuvent être des métriques de l'environnements (par exemple, de température, d'humidité, de pression artérielle) et sont enrichies par les équipements IoT. Ils peuvent aussi être un moyen d'interaction avec des actionneurs (par exemple, un moteur ou une caméra).

Concernant les réponses possibles du système en cas de satisfaction de la requête, les interactions se font via une API REST qui peut être implémentée via HTTP ou CoAP par exemple. Ainsi, pour une requête de création (POST), le code d'état est 201. Alors que pour une requête de récupération (GET) ou une requête de modification (PUT), il est de 200. Enfin, ce code devient 204 en cas de requête de suppression (DELETE) avec succès. D'autres réponses existent aussi en cas de réponse négative et dont le code de retour dépend de la nature de l'erreur.

Service "Define High Level QoS Policy"

Les acteurs de ce service sont *Administrator* et *IoTApplicationQoSAware* et interagissent soit via une API REST soit via une interface graphique spécifique. Ce service donne la possibilité de définir les besoins en QoS de haut niveau. Ces besoins constituent des contraintes de bout en bout que le système IoT-Q doit satisfaire. Ils peuvent se décliner par exemple en délai de bout en bout, en taux de pertes moyen, etc. Cette politique de haut niveau est ensuite traduite par le système en des politiques locales à respecter par chaque entité Middleware par laquelle passera le trafic.

Puisque l'API d'interaction est RESTful, les réponses du système vont être avec un code d'état 201 en cas de bonne définition de la politique de QoS.

Service "Get QoS Mechanism Adaptability Command"

Ce service est accessible à l'acteur *IoTApplicationQoSAware*. Il permet de faire participer les applications dans la gestion de la QoS en leur demandant par exemple de réduire leur taux de requêtes envoyées en cas de perte.

Service "Manage ME Models"

L'acteur *ModelDeveloper* peut, à travers ce service, enrichir les bases de connaissance des différents gestionnaires autonomiques. Compte tenu des caractéristiques de l'entité gérée, l'acteur introduit des modèles de données et de gestion des différents composants via des patterns de caractérisation.

Ces modèles peuvent être des stratégies (ou approches) de gestion comme par exemple un monitoring actif ou passif, réactif ou proactif, une analyse qui interroge le système ou se base sur une connaissance du système, etc. Ils peuvent aussi être des techniques de gestion permettant de mettre en place les stratégies, relevant par exemple du CEP, des chroniques, de théorie des files d'attente, du Machine Learning, etc. avec la définition des règles et politiques attenantes.

Service "Administrate IoT-Q"

Le système offre à l'administrateur la possibilité de l'administrer en lui permettant de le démarrer / arrêter et de le configurer. Cette configuration passe, par exemple, par le choix de l'ensemble des modèles développés par l'acteur *ModelDeveloper*. Il choisit la stratégie, les techniques de mise en place ainsi que les règles de gestion de l'ensemble des composants intervenant dans la gestion autonomique. Par exemple, pour le composant de monitoring, l'administrateur peut choisir un monitoring proactif comme stratégie et un CEP comme technique. L'administrateur instancie alors le modèle des symptômes sur lesquels le composant de monitoring se basera pour lever des alarmes.

L'acteur établit la séquence d'actions suivante pour la configuration d'un composant :

1. l'administrateur s'authentifie auprès du système IoT-Q (utilisation d'une interface dédiée)
2. le système affiche l'ensemble des entités de gestion
3. l'administrateur choisit laquelle il veut configurer
4. une fois l'entité choisie, le système affiche les différents composants constituant cette dernière (Monitor, Analyzer, Planner, Executor)
5. l'administrateur choisit un de ces composants
6. le système affiche toutes les informations relatives au composant choisi
7. l'administrateur peut alors choisir les stratégies et techniques pour la gestion interne des événements à l'intérieur du composant
8. à la fin, l'administrateur valide ses choix et ferme l'interface de configuration

Suite à cette configuration, le système envoie la nouvelle configuration à l'entité de gestion et notifie l'administrateur de la bonne prise en considération de sa configuration via un code d'état 201 et une représentation de la configuration.

Service "Manage QoS Mechanism"

Le système permet à l'acteur *QoSMechanismDeveloper* de gérer les mécanismes orientés QoS. Ces mécanismes se déclinent en deux types : (1) les mécanismes dits *orientés trafic* qui

touchent directement aux requêtes applicatives à travers des techniques relevant des approches utilisées traditionnellement dans l'Internet (rejet, retardement, contrôle de congestion, etc.) ; (2) les mécanismes *orientés ressources* qui sont responsable de la gestion des ressources dont disposent les entités Middleware (par exemple en termes de mémoire ou de CPU). L'acteur peut alors enrichir la base des mécanismes existants en y ajoutant de nouveaux mécanismes, consulter la liste des mécanismes existants, modifier un mécanisme existant, le configurer ou le supprimer.

Description de haut niveau du système de gestion

Cette section introduit les fonctionnalités des entités principales du système IoT-Q. Ces entités sont : l'entité Middleware gérée (ME), le gestionnaire autonomique esclave (AMS), et le gestionnaire autonomique maître (AMM).

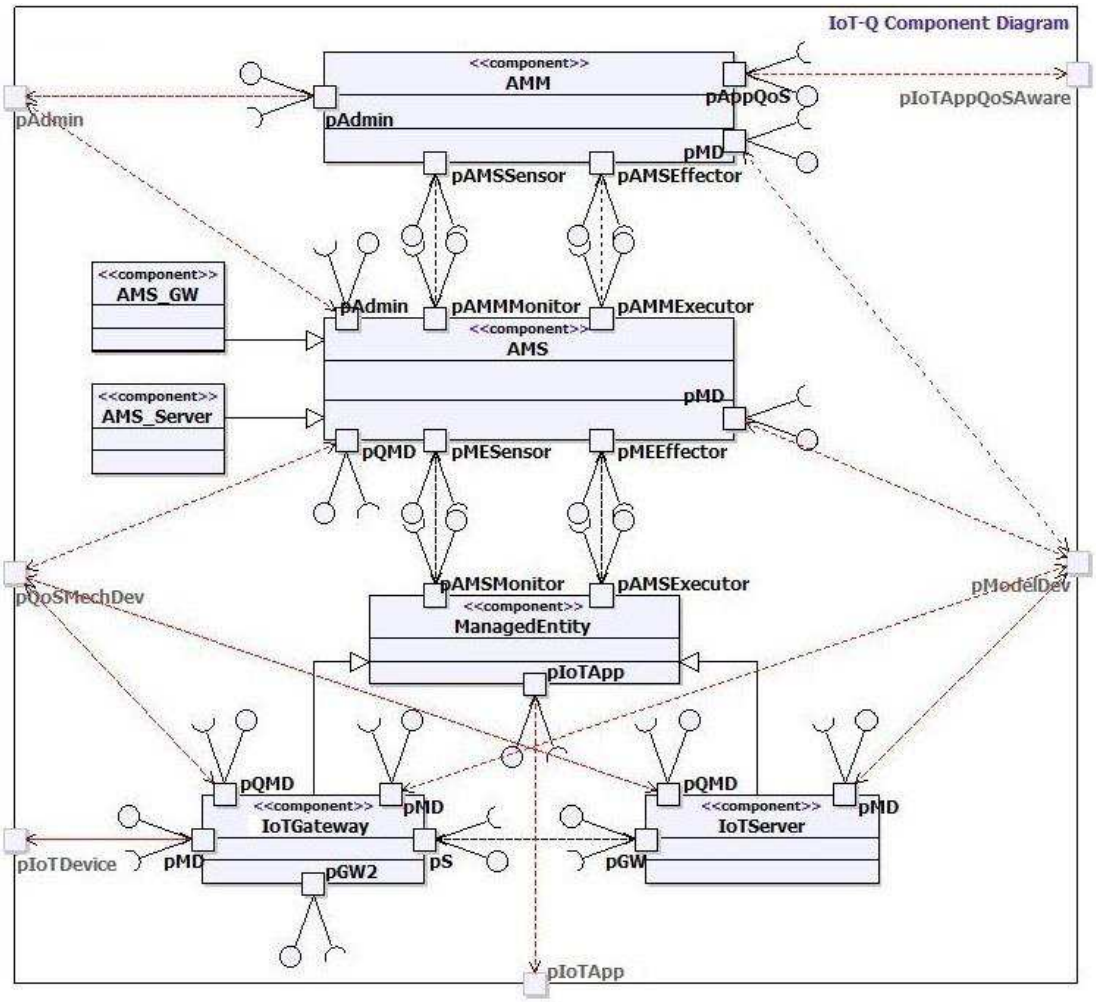

Figure 3.6 - Diagramme de structure composite de haut niveau du système IoT-Q

La Figure 3.6 fournit le diagramme de composants du système IoT-Q. Une description de l'entité gérée ainsi qu'un raffinement des composants AMM et AMS sont donnés dans les sections qui suivent. La communication entre les acteurs et le système, et entre les différents composants du système se fait via les interfaces suivantes :

- pIoTApp : permet de traiter les différentes requêtes issues des applications. Ces applications, en fonction de leur type, peuvent communiquer soit avec le serveur, soit directement avec la gateway ;

- pIoTDevice : permet la communication entre la gateway et les équipements *legacy* qui n'implémentent pas le standard et qui ne sont pas capables de communiquer directement via Internet avec le serveur ;
- pQoSMechDev : permet la communication avec le développeur de mécanismes QoS spécifiques aux gateways ou au serveur ;
- pModelDev : sert à la communication avec le développeur des modèles spécifiques aux gateways ou au serveur afin de communiquer leurs caractéristiques ;
- pAMSMonitor : sert à la communication des valeurs des métriques de performances via des capteurs logiques au composant de monitoring de l'AMS dédié à la gestion de l'entité gérée ;
- pAMSExecutor : sert à la réception des commandes de contrôle des mécanismes orientés QoS issues du composant d'exécution de l'AMS dédié à la gestion de l'entité gérée ;
- pGW2 : présent au niveau du composant *IoTGateway* pour permettre une éventuelle communication entre les gateways.

Dans ce qui suit, nous présentons l'architecture fonctionnelle de haut niveau de chaque entité ainsi que les interactions entre les composants.

Description de l'entité Middleware

Suivant notre approche de gestion autonomique hiérarchique, chaque entité Middleware est gérée d'une manière indépendante par un AMS dédié. Que ce soit dans le cadre du standards SmartM2M de l'ETSI ou celui de oneM2M, deux types[1] d'entités Middleware sont cependant à gérer : (1) les gateway IoT qui servent de passerelle vers les équipements ; selon le standard oneM2M, une gateway peut disposer d'une interface lui permettant de communiquer avec d'autres gateways ; (2) le serveur IoT qui sert au stockage centralisé de certains équipements IoT ainsi qu'à la communication entre les applications IoT et les gateways ou les équipements.

Pour les implémentations de l'entité Middleware, nous considérons celles qui sont basées sur le langage de programmation Java (notamment, la plateforme OM2M). Tel qu'illustré sur la Figure 3.7, chaque instance IoT est un programme exécuté sur une machine virtuelle Java (JVM - *Java Virtual Machine*) déployée sur une machine physique ou virtuelle en fonction de l'environnement de déploiement. L'entité Middleware est constituée d'une ou plusieurs instances IoT, de la JVM, ainsi que de la VM si c'est déployé dans un environnement virtuel.

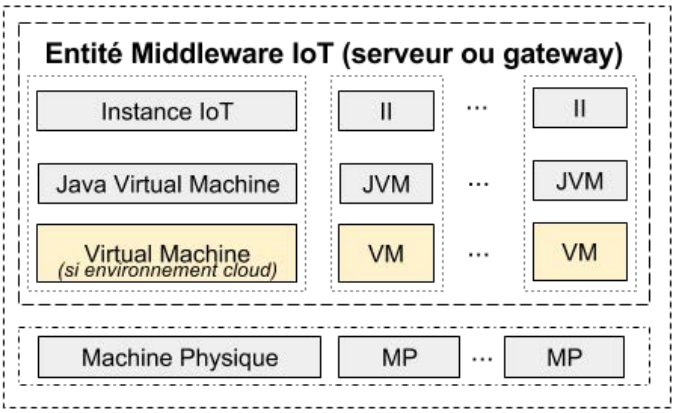

Figure 3.7 - Niveaux de gestion de l'entité Middleware IoT

[1]Dans notre cas, nous ne considérons pas l'équipement qui peut aussi implémenter ces standards

Pour assurer sa gestion par l'AMS, chaque entité Middleware doit offrir, via des points de contact, des interfaces (ou capteurs logiques) permettant de collecter les informations de supervision (temps de traitement, taux de pertes, consommation de la CPU, de la mémoire, du disque dur, etc.). Les interfaces d'actions (ou effecteurs) sont reliées aux composants implémentant les mécanismes de gestion de la QoS présentés dans le chapitre 2.

Les mécanismes orientés trafic se basent sur le composant de classification et de marquage (CCM) permettant l'attribution de la priorité en fonction du trafic et sur le composant Proxy orienté priorité (POP) pour le traitement du trafic en fonction de sa priorité. Ces composants sont placés en amont de l'entité Middleware (Figure 3.8) et disposent d'interfaces permettant leur configuration (activation, désactivation, communication de la politique de gestion, etc.).

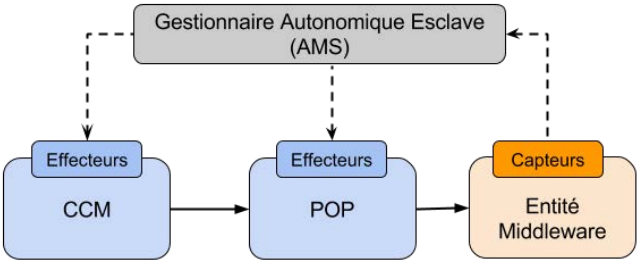

Figure 3.8 - Positionnement du CCM et POP pour une gestion orientée trafic

Alors que la gestion via des mécanismes orientés ressources, notamment via la duplication et la répartition de charge, l'AMS est responsable de la configuration du répartiteur de charge introduit au chapitre 2. Ce dernier a pour rôle de répartir la charge de trafic entre toutes les instances de l'entité Middleware (Figure 3.9).

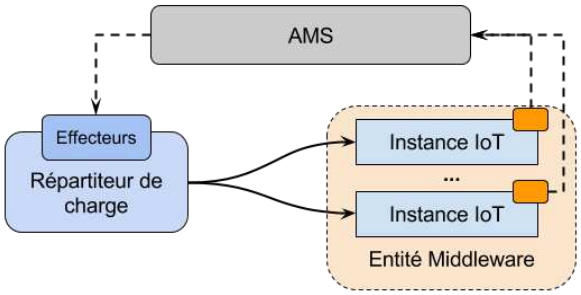

Figure 3.9 - Positionnement du répartiteur de charge pour une gestion orientée ressources

Dans ce qui suit, nous donnons la description de l'AMS. Cette description comporte les composants internes de gestion ainsi que les interfaces des communications avec l'entité Middleware gérée, ainsi qu'avec l'AMM assurant une gestion globale de la QoS.

Description de l'Autonomic Manager Esclave

Le gestionnaire autonomique esclave (AMS) est unique pour chaque entité Middleware. Son rôle est d'assurer la gestion locale d'une entité IoT dédiée (serveur ou gateway, appelé EM pour Entité Middleware) en veillant au respect de la politique locale issue de l'AMM. Afin d'assurer ses fonctionnalités, il doit implémenter la boucle MAPE-K et interagir avec les points de contact de l'entité IoT gérée (Figure 3.10).

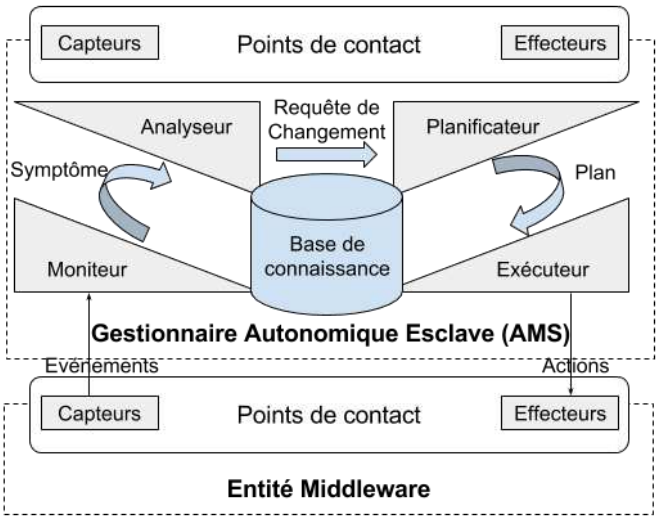

Figure 3.10 - Architecture fonctionnelle de haut niveau de l'AMS

En terme de conception et tel qu'illustré par la Figure 3.11, l'architecture fonctionnelle de haut niveau est raffinée en diagramme de composants ; ces composants sont les suivants :

- **AMSController :** il reçoit les commandes d'administration depuis l'AMM. Ce composant offre les points de contact à l'AMM pour assurer sa gestion. Il est doté de capteurs logiques dont la communication se fait via l'interface pAMMMonitor pour la collecte des métriques de performances (ici, l'objectif local atteint) par l'AMM. Ainsi que des effecteurs (via l'interface pAMMExecutor) pour la mise à jour de la politique locale à gérer par l'AMS. Après l'exécution de l'entité Middleware, l'AMS démarre et s'enregistre auprès de l'AMM via l'AMSController ;
- **AMSMonitor :** il supervise l'entité Middleware à travers ses capteurs logiques, et décide de lever ou non une alarme indiquant que la politique locale n'est pas respectée. Cette alarme se traduit par un symptôme qui sera envoyé vers l'AMMAnalyzer. Le raisonnement de ce composant peut se faire selon différents modes (passif ou actif) et approches (réactive ou proactive), via des techniques de définition de patterns telles que le CEP, les chroniques, etc. Ces techniques permettent de faire du filtrage, de l'agrégation et de la corrélation des événements collectés ;
- **AMSAnalyzer :** dès la réception du symptôme, ce composant se charge d'analyser le système afin de détecter la / les cause(s) de cette dégradation. Son raisonnement peut se baser soit sur une interrogation du système géré via l'API offerte à l'AMSMonitor, ou bien sur une connaissance du système basée sur des modèles stockés dans la base de connaissance. Dès la détection de la cause, il envoie une requête à l'AMSPlanner afin de palier à cette dégradation ;
- **AMSPlanner :** sur la base du diagnostic de l'AMSAnalyzer, il élabore un plan permettant de reconfigurer le système afin de respecter la politique locale. Cette élaboration peut se faire via des techniques allant des plus simples basées sur des moteurs de règles de type *SI… ALORS…,* jusqu'aux plus complexes basées sur des modèles d'apprentissage automatique (Machine Learning) ou autres. Le plan élaboré peut conduire à l'activation / désactivation d'un ou plusieurs mécanismes ainsi qu'à leur paramétrage. Ce plan est envoyé par la suite au composant d'exécution ;
- **AMSExecutor :** il reçoit le plan et veille à l'application des actions d'adaptation en interagissant avec les éléments gérés, par le biais de l'API spécifique que fournissent les effecteurs de l'entité gérée ;
- **AMSKnowledgeBase :** il sert au stockage des informations relatives aux modèles et instances de gestion (règles, patterns, symptômes, politique de haut niveau,

mécanismes, etc.) ainsi que des informations décrivant l'entité gérée (type, adresse, etc.). L'AMS doit aussi stocker l'objectif qu'il a pu atteindre via sa politique de gestion.

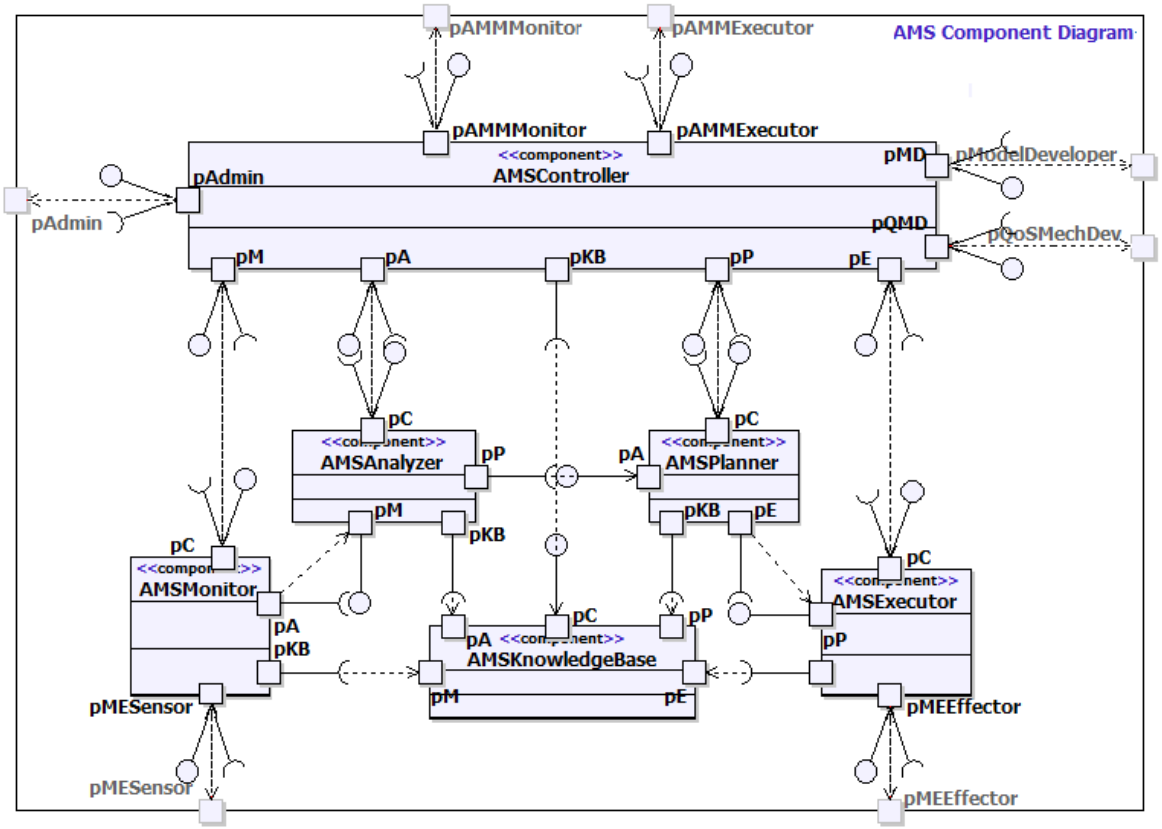

Figure 3.11 - Diagramme de composants de l'AMS

Le raisonnement interne des différents composants de l'AMS se base en entrée et produit en sortie les informations suivantes :

- en entrée, ces informations consistent en la nature et l'adresse de l'entité Middleware à gérer, la politique locale à atteindre issue du raisonnement de l'AMM, ainsi que les métriques collectées ;
- en sortie, les informations produites consistent en les actions d'adaptation à mettre en œuvre par l'entité IoT gérée.

En outre, le raisonnement interne des composants de l'AMS repose sur l'utilisation de modèles (basés sur les files d'attente, les séries temporelles et les graphes) qui seront présentés aux chapitres 4 et 5.

Description de l'Autonomic Manager Maître

Le gestionnaire autonomique maître (AMM) assure une gestion autonomique des AMS vis-à-vis de la QoS requise de bout en bout par l'application (délai de bout en bout, un taux de pertes, un débit, etc.). L'AMM implémente la boucle MAPE-K et est unique dans le système. L'objectif de QoS (politique applicative) lui est communiqué soit par l'application IoT QoS-aware soit par l'administrateur du système IoT-Q dans le cas des applications QoS-unaware. L'AMM va alors, dès réception de cette politique, faire le traitement nécessaire afin de définir les objectifs de QoS locaux (politiques locales) à respecter par chaque AMS dont la requête applicative "passe" à travers son entité IoT gérée, et les communiquer via les points de contact qu'offre chaque AMS (Figure 3.12). Par exemple, soit la politique applicative exprimée en

terme d'un temps de réponse maximal à respecter R_{MAX}. La requête parcourt n entités Middleware. La politique locale est exprimée par l'AMS à l'ensemble des AMS gérant les entités Middleware impliquées dans le traitement de la requête. Cette politique est sous la forme d'un délai maximal à respecter (d_{MAXi}) tel que : $R_{MAX} = \sum_{i=1}^{n} d_{MAXi}$.

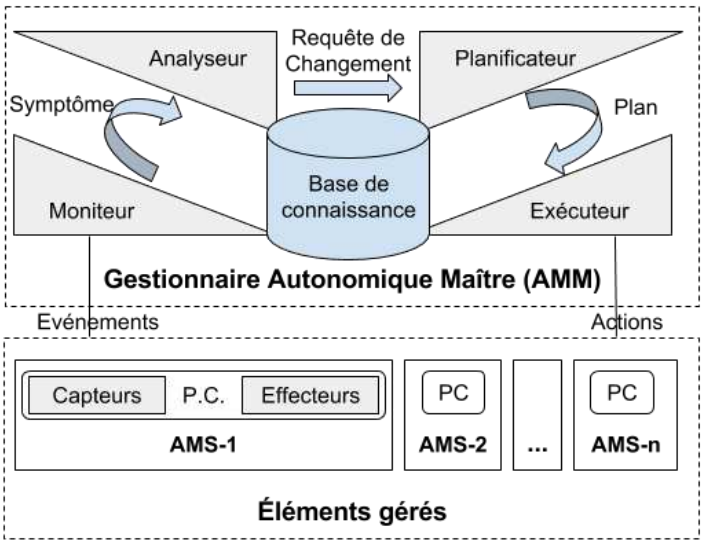

Figure 3.12 - Architecture fonctionnelle de haut niveau de l'AMM

L'architecture fonctionnelle de haut niveau de l'AMM se traduit par le diagramme de composants illustré Figure 3.13 constitué des composants suivants :

- **AMMController :** il constitue l'interface d'entrée pour les acteurs souhaitant interagir avec l'AMM. Il permet aux acteurs en ayant le droit, de configurer les différents composants de l'AMM, de définir les modèles, règles, politiques, etc., ainsi que de prendre en considération les politiques de haut niveau des applications QoS-aware. Ce composant comporte aussi une interface graphique dédiée à ces interactions ;
- **AMMMonitor :** il collecte les informations permettant de vérifier la conformité du résultat atteint par la globalité des AMS avec la politique de haut niveau ciblée (i.e. la QoS de bout en bout à satisfaire). Pour le cas d'interactions de type requête / réponse, il détermine et paramètre l'AMS d'entrée afin de collecter le résultat atteint. Par exemple, dans le cas d'une interaction requête / réponse entre une application et une gateway en passant par un serveur, l'AMMMonitor collecte le temps de réponse global atteint depuis l'AMS du serveur. En cas de non-conformité entre l'objectif atteint par les AMS et la politique de haut niveau, il génère un symptôme et l'envoie au composant AMMAnalyzer ;
- **AMMAnalyzer :** il analyse les causes potentielles de dégradation, par exemple,-un ou plusieurs AMS qui n'arrivent plus à respecter les politiques locales qui leurs ont été attribuées. Le raisonnement pour l'AMMAnalyzer consiste à interroger les différents AMS dont les entités gérées interviennent dans la satisfaction de la requête applicative et à détecter la source de la non satisfaction de la QoS. Suite à cette analyse, il envoie le résultat de son diagnostic sous forme d'une requête pour un changement de configuration (RFC) à l'AMMPlanner ;
- **AMMPlanner :** il est responsable de l'établissement du plan de (re)configuration des objectifs de QoS locaux à atteindre par les AMS. A l'analogue de l'AMSPlanner, son raisonnement peut être basé sur des règles de type "*Si … Alors …",* ou sur des modèles plus complexes. Ce plan est envoyé par la suite à l'AMMExecutor ;

- **AMMExecutor :** il transforme le plan d'action précédent en des requêtes spécifiques à l'API d'interaction avec les effecteurs des AMS puis les envoie à ces derniers ;
- **AMMKnowledgeBase :** il sert au stockage des informations relatives aux modèles et techniques de gestion ainsi qu'à l'enregistrement des AMS auprès de l'AMM.

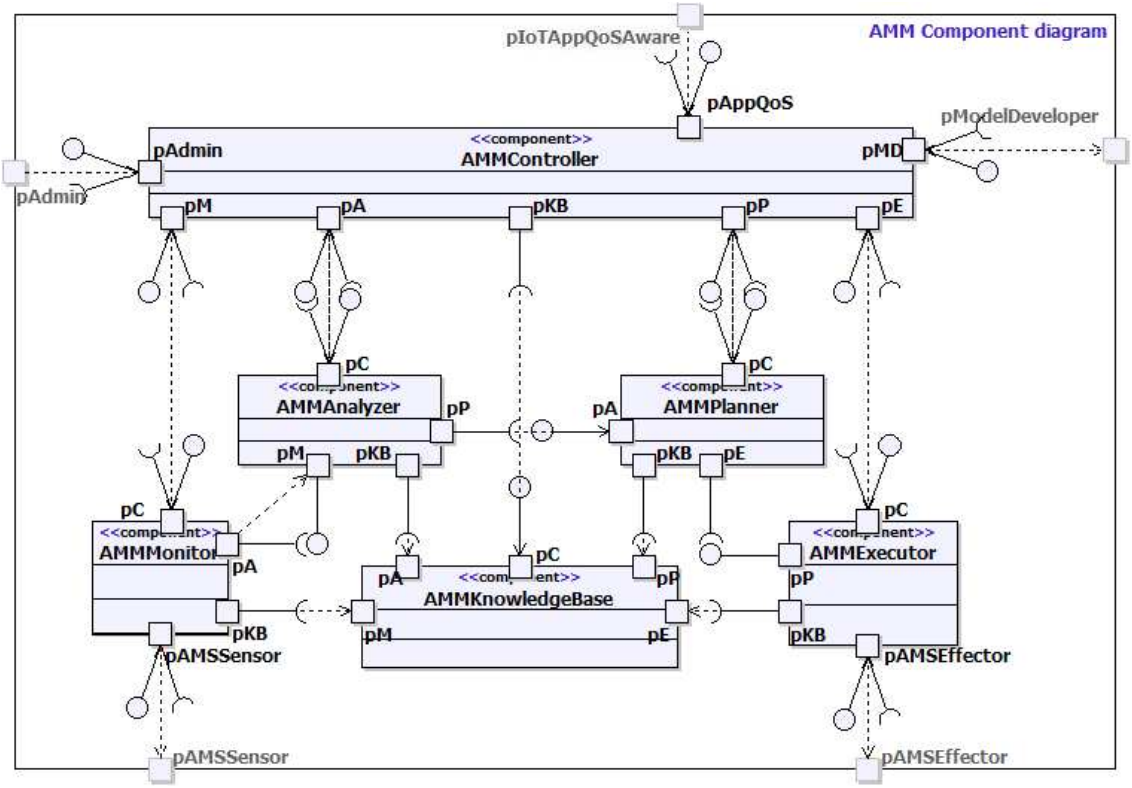

Figure 3.13 - Diagramme de composants de l'AMM

Le raisonnement interne des différents composants de l'AMM se base en entrée et produit en sortie les informations suivantes :

- en entrée, les informations consistent en la politique de haut niveau à atteindre qui correspond au besoin en QoS de bout en bout, ainsi qu'en les métriques de supervision de l'AMS ;
- en sortie, les informations produites consistent en les objectifs QoS locaux des AMS que chaque entité Middleware doit respecter.

Interactions entre les acteurs et les composants

Dans cette section, nous illustrons les interactions entre les composants afin de fournir les différents services. Nous nous contentons de présenter quelques interactions à titre illustratif.

Administration de l'AMM

Ce service vise à configurer les différents composants de l'AMM par l'administrateur et le démarrer. Supposons que les bases de connaissance de l'AMM et des AMS ont été déjà enrichies par l'acteur *ModelDeveloper*. L'administrateur va alors choisir les stratégies et les techniques de gestion qui lui conviennent.

Comme illustré par la Figure 3.14, l'administrateur procède comme suit pour configurer l'AMM :

- au premier lancement du système IoT-Q, l'administrateur doit réaliser sa configuration. Pour ce faire, celui-ci s'identifie tout d'abord auprès de l'AMMController via une interface graphique dédiée ;
- si l'authentification se passe correctement, l'ensemble des gestionnaires autonomiques AMM et AMS enregistrés auprès de ce dernier sont affichés ainsi que leurs configurations. Cette configuration contient les modèles des stratégies et techniques enrichis par le ModelDeveloper ;
- l'administrateur choisit alors le gestionnaire autonomique qu'il souhaite configurer. Il envoie alors la configuration choisie vers l'AMMController qui l'analyse et la route vers le(s) composant(s) interne(s) concerné(s) par cette configuration.

Dans notre cas, la configuration se fait pour l'AMM. L'administrateur peut aussi choisir de configurer directement un AMS ; la configuration est alors redirigée par l'AMMController vers l'AMSController de l'AMS concerné.

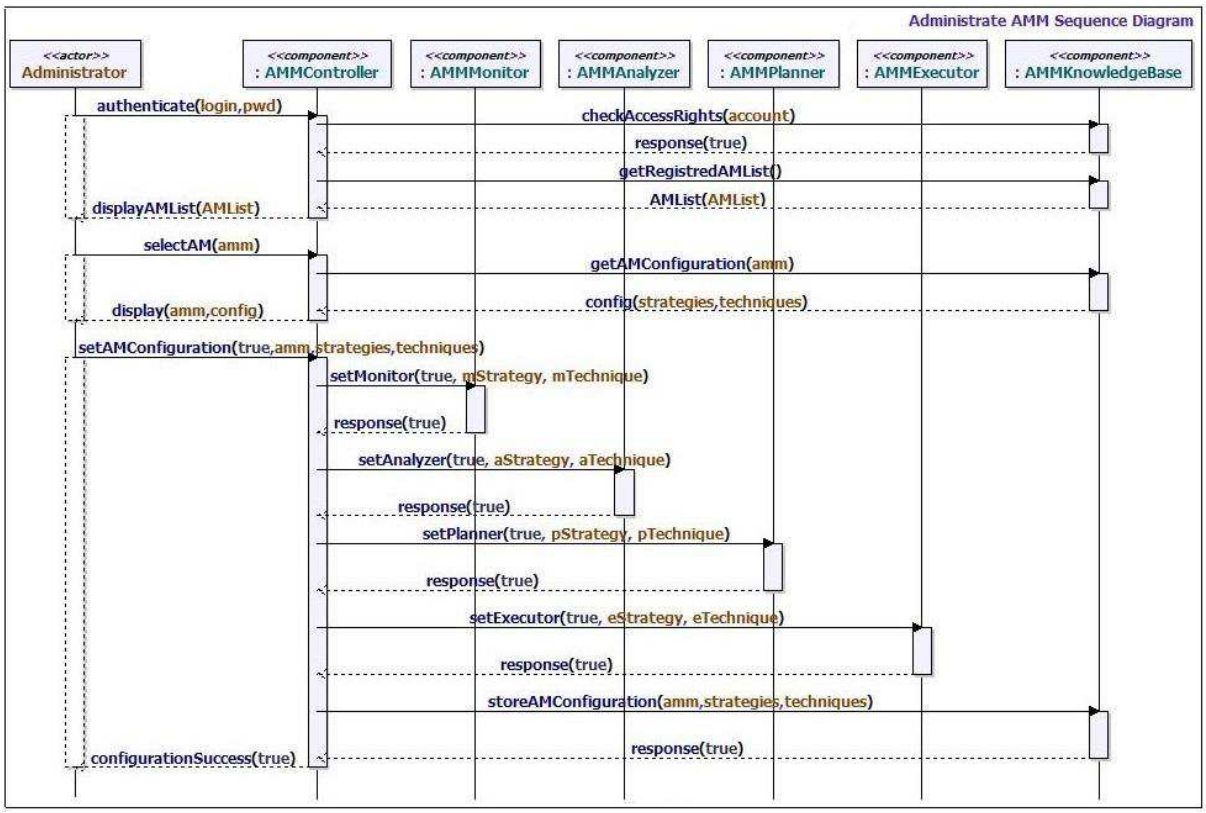

Figure 3.14 - Diagramme de séquence pour l'administration de l'AMM

Définition et prise en considération des besoins en QoS

Les besoins en QoS définis, soit par l'application QoS-aware, soit par l'administrateur, doivent être pris en considération par l'ensemble des entités du système intervenant dans le traitement du trafic applicatif.

La Figure 3.15 illustre l'ensemble des interactions nécessaires :

- l'application (de type IoTApplicationQoSAware) envoie une commande defineHighPolicy vers le composant AMMController. La requête comporte : (1) la politique correspondant aux besoins en QoS de bout en bout à satisfaire par le système IoT-Q ; et (2) l'entité destinataire (le Middleware et la ressource) ;

- l'AMMController reçoit la requête, la stocke au niveau du composant AMMKnowledgeBase et invoque la méthode setAMSPolicies afin de demander au composant AMMPlanner de calculer les politiques locales pour les différents AMS ;
- la base de connaissance AMMKnowledgeBase contient les politiques actuelles des AMS ; l'AMMPlanner lui envoie une demande via la méthode getAMSPolicies pour récupérer la liste des AMS et leurs politiques actuelles (AMSListInfo) ;
- l'AMMPlanner procède alors au calcul de l'objectif local de QoS pour chaque AMS en fonction de la politique de haut niveau issue de l'application. Il détermine aussi l'AMS de l'entité Middleware d'entrée (via AddHighToAMSPolicy) afin que la collecte du résultat final par l'AMMMonitor se fasse depuis elle ;
- l'AMMPlanner construit son plan comportant les nouvelles politiques locales calculées pour chaque AMS. Il envoie ensuite ces nouvelles politiques à l'AMMKnowledgebase afin qu'elles soient stockées, et le plan à l'AMMExecutor afin qu'il soit transmis aux différents AMS concernés ;
- l'AMMExecutor reçoit le plan et l'envoie vers le composant AMSController des AMS concernés. L'AMSController veille alors à la configuration des métriques à collecter par l'AMSMonitor ainsi que les nouveaux seuils afin de guider sa supervision ;
- en plus, l'AMMController fournit comme informations à l'AMMMonitor : les caractéristiques des requêtes applicatives, l'AMS d'entrée depuis lequel la collecte va se faire, les métriques à collecter, ainsi que le(s) seuil(s) permettant de générer des symptômes de haut niveau reflétant l'insatisfaction de la politique de haut niveau.

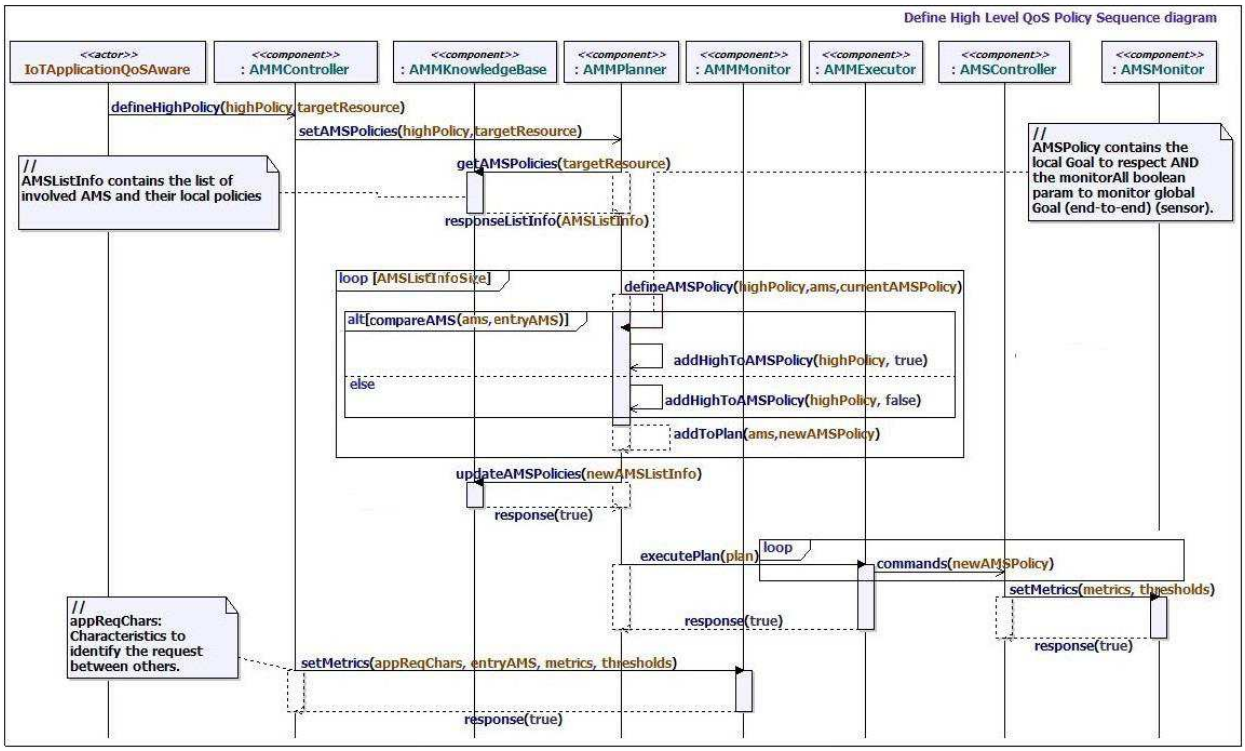

Figure 3.15 - Diagramme de séquence pour la définition de la politique de haut niveau

Gestion de la conformité des AMS à la politique locale

La politique de haut niveau (besoin en QoS de bout en bout) se décline en des politiques locales (besoin local en QoS) que chaque AMS aura à satisfaire. Pour ce faire, ce dernier doit collecter les métriques nécessaires et agir en conséquence. L'AMS dispose des mécanismes QoS qui ont été développés par l'acteur *QoSMechanismDeveloppe*r.

La Figure 3.16 définit les interactions visant à garantir la satisfaction de la politique locale de la QoS d'une manière autonomique :

- l'AMSMonitor démarre la supervision en spécifiant les métriques qu'il souhaite collecter (en fonction de l'entité gérée) ;
- sur la base de requêtes envoyées vers l'entité gérée depuis l'application IoT pour la gestion d'une ressource IoT donnée, les événements sont envoyés vers le composant de monitoring comportant les métriques et leurs valeurs collectées ;
- l'AMSMonitor stocke ces événements dans la base de connaissance et les analyse via la méthode analyseEvents(events). A l'issue de cette analyse, un symptôme peut être levé ;
- Si un symptôme est levé, il sera envoyé vers l'AMSAnalyzer qui se chargera d'analyser la cause de cette dégradation (via la méthode analyseSymptom(symptom)). Il peut interroger l'entité gérée afin de collecter plus de métriques pour aider son diagnostic. Le résultat sera exprimé sous forme d'une requête de changement envoyée vers le composant AMSPlanner ;
- l'AMSPlanner se charge alors de déterminer le plan adéquat permettant de palier à ce symptôme. Il construit le plan et l'envoie vers l'AMSExecutor qui l'exécutera.

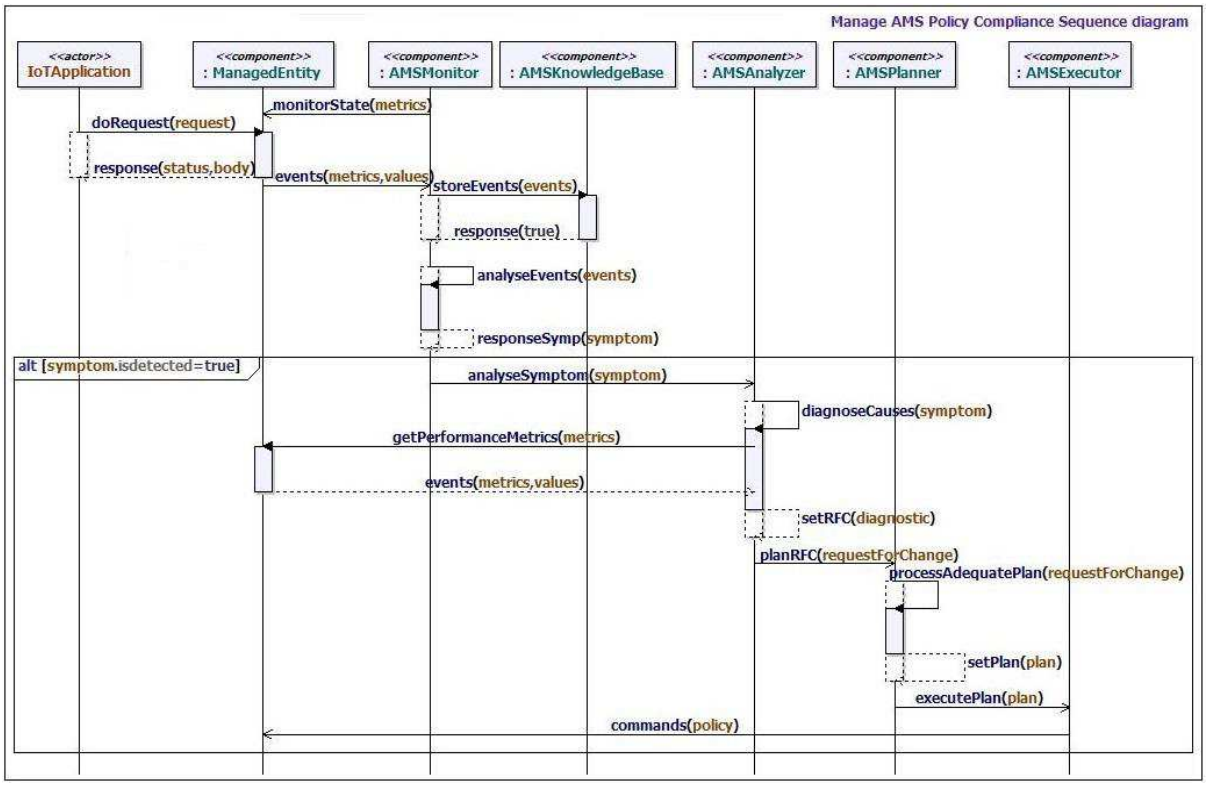

Figure 3.16 - Diagramme de séquence pour la satisfaction de la politique locale par l'AMS

Exemple d'une gestion hiérarchique de la politique de haut niveau

Considérons un exemple d'architecture (Figure 3.17) constitué d'un serveur IoT (IoT Server) et d'une seule gateway IoT (IoTGW_1) à laquelle est attaché un équipement (d). Soit une application QoS-aware (IoTAPP) souhaitant envoyer des requêtes vers cet équipement avec une certaine exigence en QoS (highPolicy) vis-à-vis du traitement de ses requêtes par le système.

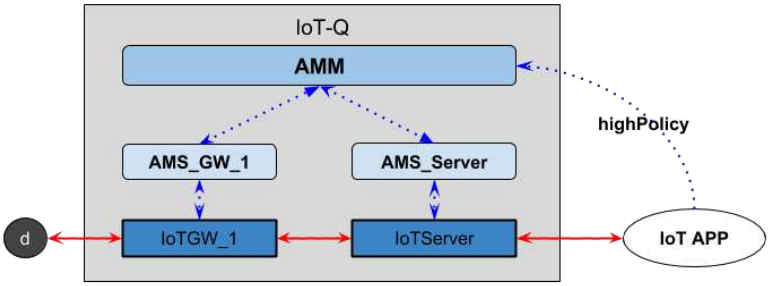

Figure 3.17 - Exemple illustratif de déploiement du système IoT-Q

Afin que le système puisse prendre en considération les besoins de l'application et gérer les entités le constituant, il procède à l'enchaînement illustré par la Figure 3.18.

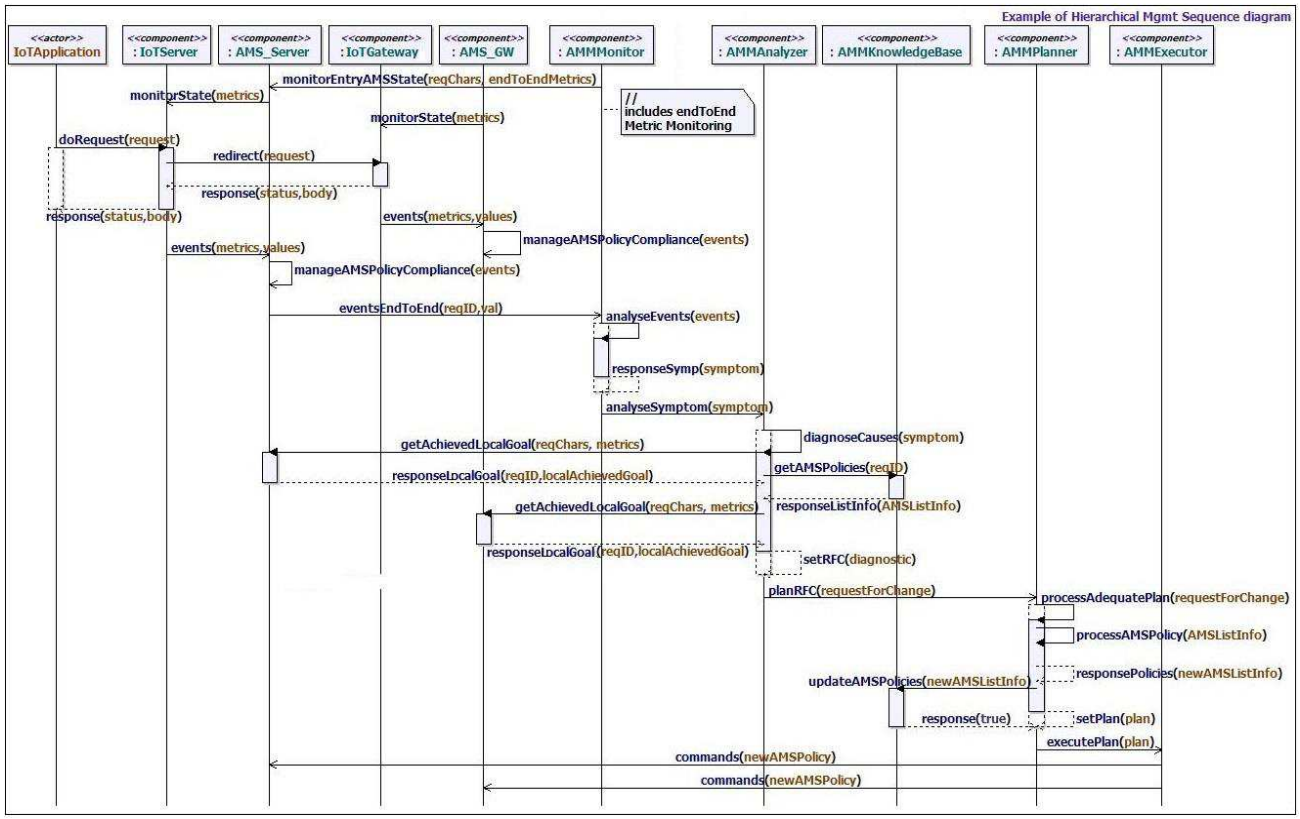

Figure 3.18 - Exemple de gestion hiérarchique de la politique de haut niveau

Tout d'abord, la supervision est déclenchée par l'entité Middleware par chaque AMS en spécifiant les métriques à collecter. En parallèle, puisque le serveur IoT constitue le point d'entrée au système IoT-Q, l'AMS_Server collecte aussi la métrique caractérisant la politique de bout en bout permettant de vérifier sa conformité avec la politique de haut niveau (highPolicy) et la stocke dans sa base de connaissance (AMSKnowledgeBase). Ensuite, le composant AMMMonitor récupère cette métrique ainsi que l'identifiant de la requête en question afin de la comparer à la politique de haut niveau (via la correspondance des événements avec le pattern de levée de symptôme). Si un symptôme est levé, il sera envoyé vers l'AMMAnalyzer afin d'être diagnostiqué. Ce diagnostic se basera sur l'interrogation (getAchievedLocalGoal) des différentes bases de connaissance où est stocké le meilleur cas que chaque AMS a pu atteindre à travers sa gestion de l'entité gérée. L'AMMAnalyzer compare dans ce cas les objectifs atteints par chaque AMS avec sa politique locale afin de déduire la / les entité(s) qui est / sont incapable(s) de respecter la contrainte. Une fois la comparaison faite, le composant envoie une requête de changement comportant le résultat de

cette analyse. L'AMMPlanner reçoit ce RFC et élabore un plan sur la base d'un calcul des nouvelles politiques locales afin que la politique de haut niveau soit respectée. Pour ce faire, il se base sur la politique de haut niveau, la politique locale attribuée à chaque AMS, l'objectif qu'il a pu atteindre, ainsi qu'un algorithme d'optimisation interne à la fonction. Une fois le plan élaboré, il est envoyé au composant AMSExecutor afin d'expédier les nouvelles politiques locales vers les différents AMS. Enfin, l'AMSPlanner enregistre le résultat de son traitement (nouvelles politiques locales) au niveau de la base de connaissance AMMKnowledgBase.

3.4. CONCLUSION

Dans ce chapitre, nous avons proposé la spécification et la conception du système IoT-Q. Les différents acteurs et fonctionnalités ont été identifiés. IoT-Q intègre les services de gestion de la QoS en plus des services basiques du Middleware IoT. Les composants internes du système et les interactions entre eux ont aussi été décrits. Une approche basée MDA a été suivie pour l'élaboration du système de manière générique applicable à toute implémentation Middleware. L'architecture du système de gestion de la QoS de bout en bout repose sur l'utilisation du paradigme de l'*Autonomic Computing* pour assurer la reconfiguration dynamique et auto-adaptative des composants.

La conception du système suit une approche hiérarchique constituée de deux niveaux de politiques de gestion de l'entité Middleware (Figure 3.19). Le premier niveau gère des actions *opérationnelles*. Il est assuré par un AMS local à chaque entité Middleware. Le rôle de cet AMS est la gestion (activation, désactivation et configuration) des mécanismes orientés trafic et orientés ressources, proposés dans le chapitre précédent, afin de respecter l'objectif local de QoS, propre à chaque entité Middleware. Cet objectif local est communiqué par l'AMM représentant le deuxième niveau de gestion d'actions dites *stratégiques*. Unique dans le système, il assure la prise en compte du besoin de bout en bout de la QoS de l'application IoT si elle est capable de l'exprimer (QoS-aware), sinon, il établit des politiques pour chaque profil applicatif (par exemple, en fonction du type de données ou d'interactions). Il établit ensuite les objectifs locaux de QoS à faire respecter par chaque entité Middleware parcourue par le trafic.

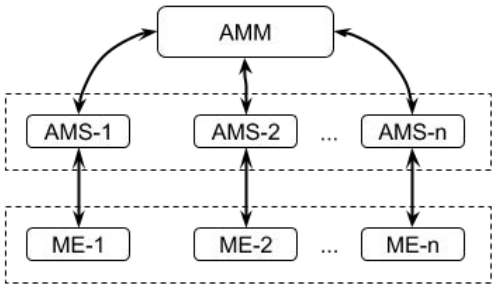

Figure 3.19 - Architecture du système IoT-Q

La contribution principale de ce chapitre est d'ordre structurel. Dans ce qui suit, nous nous focalisons sur l'aspect comportemental de l'architecture de l'AMS. Il s'agit de propositions pour alimenter le comportement interne (raisonnement) des composants de l'AMS afin de guider leur fonctionnement. Les propositions dans les deux chapitres suivants sont d'ordre comportemental vis-à-vis des composants de gestion. Elles sont divisées en deux familles. La première famille porte sur la proposition de modèles de représentation des différents éléments du système géré (entité Middleware, trafic, etc.) pour guider la gestion locale des entités Middleware. La deuxième famille concerne l'application de ces modèles aux différents composants de l'AMS pour l'élaboration des approches de gestion, spécifiquement, pour le composant de monitoring (*AMSMonitor*) et le composant de Planification (*AMSPlanner*).

Chapitre 4

Monitoring guidé par les modèles pour la gestion locale de la QoS

Contenu

4.1. INTRODUCTION

Le chapitre 3 précédent a porté sur la partie structurelle de notre proposition d'architecture IoT-Q pour une gestion autonomique de la QoS locale au niveau Middleware. Dans ce chapitre, nous traitons la partie comportementale de l'architecture par le biais de techniques, algorithmes et modèles amenés à nourrir les composants de la boucle MAPE-K des gestionnaires autonomiques locaux (AMS) identifiés au chapitre 3.

Pour aborder cette problématique, tel que présenté dans le chapitre 1 (Etat de l'art), les approches envisagées dans la littérature peuvent être classées en deux familles. La première nécessite des interactions fortes avec le système pour en capturer les états réels. Elle est conceptuellement plus simple ; cependant, elle est par définition plus lourde en terme d'implémentation et plus coûteuse en terme de performance parce qu'elle est amenée à collecter toutes les métriques du système. La deuxième famille vise à réduire ces interactions en utilisant un ou des modèles du système. L'avantage de cette approche est qu'elle est moins lourde en terme d'implémentation et moins coûteuse en terme de performance car elle ne nécessite de collecter que les métriques du système nécessaire pour alimenter les modèles considérés. En revanche, ce type d'approche est conceptuellement plus complexe et les modèles utilisés, en fonction de leur degré de précision, peuvent induire un écart par rapport à la réalité du système,

et donc une moindre précision. Notre approche se positionne dans la famille basée modèles et vise à faire progresser l'état de l'art des solutions proposées pour la gestion : (1) de la phase de monitoring, présentée en cas d'étude dans le présent chapitre, et (2) de la phase de planification, présentée dans le chapitre suivant (chapitre 5).

Dans ce chapitre, nous proposons tout d'abord un modèle, basé sur la théorie des files d'attente, d'une entité de Middleware (ici, la plateforme OM2M) amenée à enrichir la base de connaissance à des fins d'utilisation dans la phase de monitoring du gestionnaire autonomique esclave (AMS) de l'architecture du système IoT-Q. Nous proposons ensuite deux approches d'application de ce modèle dans le cadre de la phase de monitoring de l'AMS. La première vise une adaptation *en réaction* à une dégradation de la QoS (approche *réactive*), en couplage avec une technique de traitement d'événements complexes (CEP - *Complex Event Processing*). La deuxième voie promeut la volonté *d'anticiper* la dégradation de la QoS (approche *proactive*), en couplage avec un modèle de prédiction du taux d'arrivée des requêtes (le modèle ARMA : autorégressif et moyenne mobile). Notons enfin que notre modèle à base de la théorie des files d'attente sera également utilisé dans le chapitre 5, en couplage avec un modèle du système basé sur les graphes (contribution principale du chapitre 5), pour guider la phase de planification de l'AMS.

La suite de ce chapitre est divisée en deux sections majeures. La section 4.2 décrit notre proposition de modèle analytique du Middleware OM2M basé sur la théorie des files d'attente. Ce modèle a pour objectif d'enrichir la base de connaissance de la boucle autonomique en donnant une estimation des métriques de performances (temps de réponse, taille de chaque file, etc.) en fonction du taux d'arrivée des requêtes. Alors que la section 4.3 présente l'application de ce modèle dans le cadre de la supervision via le composant de monitoring de la boucle autonomique de l'AMS. Les deux approches de monitoring introduites précédemment (*réactif* et *proactif*) sont également décrites. Nous terminons ce chapitre par une conclusion sur le travail effectué et par les perspectives spécifiques relatives au modèle proposé et son application dans la phase de monitoring de l'AMS.

4.2. MODELE ANALYTIQUE DU MIDDLEWARE OM2M

Dans une approche de gestion dynamique et auto-adaptative de la QoS au niveau Middleware, l'évaluation et/ou la mesure des performances du système considéré est nécessaire pour estimer sa capacité à traiter le trafic entrant tout en satisfaisant les exigences de la QoS des applications critiques (ou les performances attendues de l'opérateur du système).

Les métriques considérées sont, par exemple, le temps de réponse, le nombre de requêtes traitées par seconde, etc. L'estimation de la valeur de ces métriques peut être envisagées de différentes façons, notamment par simulation du système et de son comportement, par expérimentation en considérant une vraie plateforme Middleware, ou bien analytiquement via un modèle du système. Notre approche s'inscrit dans l'approche *analytique*. Nous proposons un modèle basé sur la théorie des files d'attente permettant d'évaluer une entité Middleware (typiquement une gateway ou un serveur) sous l'angle des métriques de performance. Dans cette section, nous fournissons tout d'abord une étude qualitative et quantitative de la plateforme OM2M à modéliser. Nous proposons ensuite un modèle analytique de la plateforme en détaillant la démarche adoptée pour l'approximation des paramètres internes de ce modèle. Nous nous plaçons pour cela dans le contexte d'une instance de la plateforme OM2M déployée au sein d'une gateway.

4.2.1. Etude du Middleware OM2M

Afin d'aboutir à une modélisation analytique du Middleware OM2M, une analyse qualitative et quantitative de la plateforme est nécessaire. Dans cette section, nous présentons tout d'abord le concept d'*arbre de ressources* proposé par le standard SmartM2M afin de structurer les données. Ensuite, nous détaillons le fonctionnement et les performances des composants internes de la plateforme OM2M.

Concept d'arbre de ressources

Le standard SmartM2M propose une couche de services au niveau Middleware suivant le style architectural REST. Ce dernier considère que toute entité physique (capteur, actionneur, serveur ou gateway) ou logique (par exemple, combinaison d'informations depuis les entités physiques) est représentée par une *ressource*. Chaque ressource possède une représentation qui peut être créée (POST), récupérée (GET), mise à jour (PUT) ou supprimée (DELETE). L'adressage de cette ressource se fait de manière unique via l'utilisation d'un identifiant appelé URI (*Universal Resource Identifier*).

Afin de donner une structuration hiérarchique de ces ressources, le standard propose la notion d'*arbre de ressources*. Dans l'écosystème IoT, la majorité des échanges sont relatifs aux données émises depuis ou vers les équipements. Ces données sont stockées ou récupérées sur ou depuis le Middleware. La Figure 4.1 illustre la branche qui permet la représentation hiérarchique de ces données. Une ressource de type <*container*> est le parent utilisé comme conteneur des données des équipements. Chaque conteneur comporte une collection de type (*ContentInstance*) qui comporte la liste des ressources de type <*contentInstance*>. Chaque ressource de ce type contient une valeur de donnée qui peut être par exemple une donnée collectée par un capteur à un instant donné.

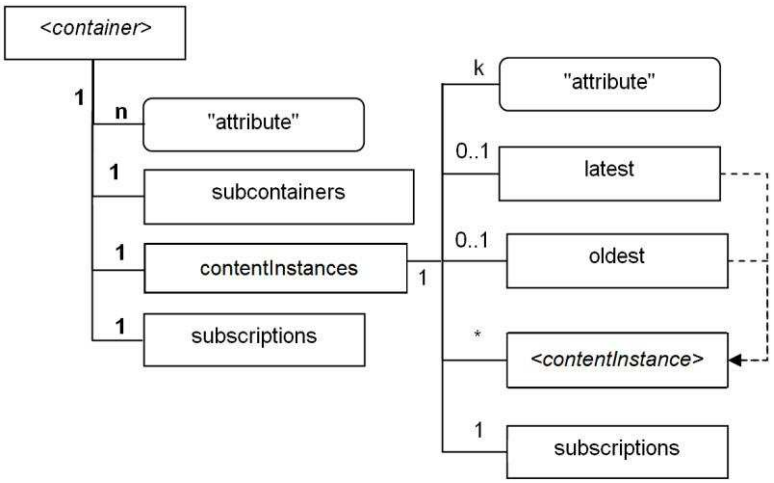

Figure 4.1 - Branche <container> de l'arbre de ressources

Dans ce qui suit, nous détaillons le comportement des composants internes de la plateforme OM2M. Nous apportons un niveau de détails pour le traitement des requêtes de gestion des ressources de type <*contentInstance*>. Dans ce qui suit, nous utilisons l'abréviation *instance* au lieu de <*contentInstance*>.

Composants de la plateforme OM2M

En termes de fonctionnalités, nous distinguons trois composants essentiels (Figure 4.2) : le composant de communication protocolaire spécifique, le composant CORE, et le composant DATABASE.

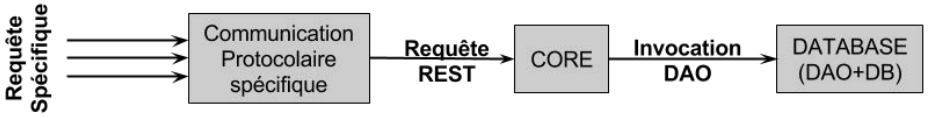

Figure 4.2 - Aperçu de haut niveau de la plateforme OM2M

Composant de communication protocolaire spécifique

Le composant de communication protocolaire spécifique comporte le serveur qui traite les requêtes entrantes en fonction du protocole de communication (HTTP ou CoAP). Basiquement, à chaque requête entrante, un thread est alloué par le serveur. Ce thread assure le passage de la requête par les différents composants internes du Middleware. Il fait partie d'un ensemble de thread (*threadPool*) qui dispose d'un nombre maximal de threads. Quand ce nombre est atteint, les nouvelles requêtes entrantes sont soit rejetées soit *bufférisées* en attente de la libération d'un thread. Chaque thread est géré par le(s) processeur(s). Il peut placer les données de traitement soit provisoirement dans la mémoire RAM, soit de manière persistante dans le disque dur via une base de donnée. A la fin du traitement de la requête, la réponse est envoyée et le thread est libéré afin d'être dédié au traitement d'une autre requête.

Le temps de traitement de chaque requête dépend en partie de la méthode utilisée. Par exemple pour les requêtes HTTP, les requêtes de création (POST) ou de modification (PUT) ont besoin de plus de temps de traitement que les requêtes de récupération (GET) ou de suppression (DELETE). Ce délai supplémentaire induit est dû au fait que la requête contient une plus grande quantité d'informations relative au corps de la requête. Ce dernier comporte la représentation de la ressource à créer sous format XML ou JSON par exemple. Notons que pour les requêtes GET ou DELETE, ce corps n'est pas présent.

Initialement, le threadPool peut être vide (aucun thread n'a été créé). A l'arrivée de chaque requête, un thread du threadPool est créé pour assurer son traitement interne. A la fin du traitement de la requête, le thread n'est pas détruit mais est remis dans le threadPool, l'objectif étant de ne pas avoir de délai supplémentaire dû à la création d'un nouveau thread. A l'arrivée d'une nouvelle requête, soit il y a un thread libre et dans ce cas la requête est traitée, soit il n'y a pas de thread libre et dans ce cas, si le nombre maximum de threads créé est atteint, la requête est rejetée. Ainsi, lorsque le threadPool n'est pas encore créé, les requêtes initiales ont un temps de réponse "grand", surtout lors d'importants taux d'arrivée.

La Figure 4.3 représente ce comportement pour des requêtes GET avec un taux d'arrivée de 12 requêtes par seconde. Après cet effet *d'avalanche*, l'évolution du temps de réponse stagne puisque les nouvelles requêtes vont utiliser les threads qui ont été déjà créés. Ce phénomène est expliqué par la latence (potentiellement grande) induite par la création du threadPool au démarrage du système [goe06] qui impacte le temps de réponse global au début (effet avalanche). Cette latence dépend à la fois de la fréquence des requêtes, des composants logiciels et du système d'exploitation. Une fois ces threads créés, ils sont utilisés par les nouvelles requêtes entrantes, l'effet d'avalanche précédent n'apparaissant plus.

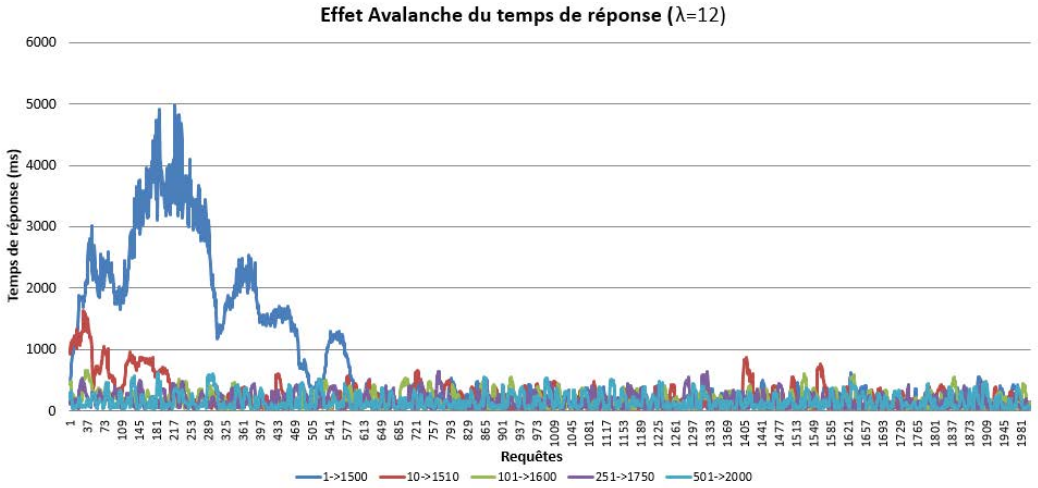

Figure 4.3 - Effet Avalanche du temps de réponse

Dans la partie modélisation, le phénomène d'avalanche étant transitoire, il n'est pas pris en considération. Nous ne considérons donc dans la suite que la partie stationnaire du comportement du composant.

Etant donné que le composant de communication protocolaire spécifique reçoit des requêtes dépendantes du protocole de communication, son rôle consiste aussi en leur traduction en des requêtes REST génériques avant de les acheminer vers le composant CORE. Une fois le traitement terminé et la réponse envoyée, le thread est libéré.

Composant CORE

Le CORE implémente une gestion générique des requêtes REST indépendamment du protocole de communication. Compte tenu des informations contenues dans la requête, il la route vers un contrôleur dédié en fonction du type de la ressource destinataire (par exemple, *applicationController, containerController, contentInstanceController*). Pour la satisfaction de la requête, le contrôleur élabore diverses opérations de vérifications (droits d'accès à la ressource, existence de la ressource mère, etc.) avant d'interagir avec la base de données à travers le DAO (*Data Access Object*).

Par exemple, pour la création d'une ressource de type *contentInstance*, le *contentInstanceController* vérifie les droits d'accès et l'existence de la ressource *container*. Une fois la vérification validée, il fait appel au *contentInstancesDAO* pour stocker la ressource dans cette collection. Pour les autres opérations, le contrôleur fait appel directement au *contentInstanceDAO* sans avoir à récupérer toute la collection.

L'impact du temps de traitement réalisé par le contrôleur est négligeable par rapport au temps de réponse global (de l'ordre de 10 ms pour des temps de réponse de l'ordre de 100 ms ou plus).

Composant DATABASE

La persistance de toutes les opérations est assurée par la base de données. L'interaction avec la base de données est assurée par le DAO qui fournit une interface abstraite sans exposer les détails sur l'implémentation de cette base de données. Chaque DAO est dédié à une ressource spécifique (*applicationDAO, containerDAO*, etc.) et permet des opérations de création, de récupération, de mise à jour et de suppression.

En terme de comportement, spécifiquement pour ce qui concerne la création d'une instance, la collection *mère* doit être entièrement récupérée. La Figure 4.4 décrit le coût de cette récupération sur le temps de traitement. L'évolution du temps de traitement est explicable par le fait que chaque instance créée est stockée dans la collection, augmentant ainsi la taille de la collection et alourdissant sa récupération pour la création d'une nouvelle instance.

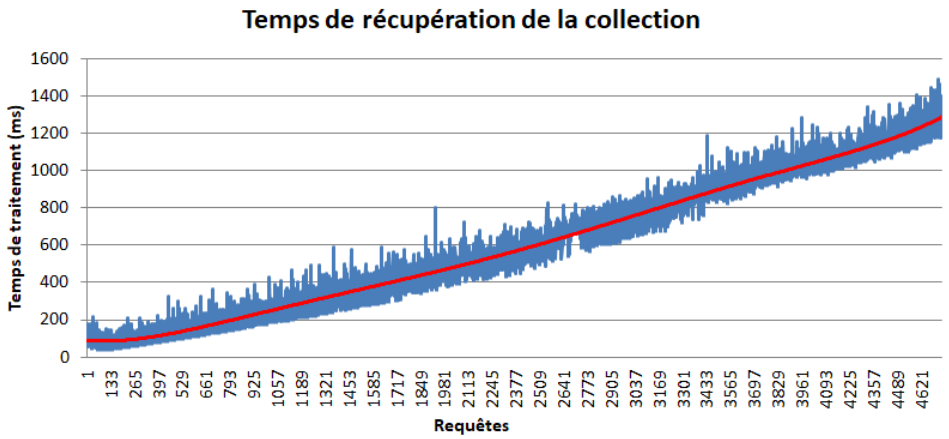

Figure 4.4 - Evolution du temps de traitement du DAO pour la récupération de la collection contentInstances

Pour l'élaboration du modèle analytique de la plateforme, ce comportement doit aussi être pris en considération concernant les requêtes de création (POST). Cela se traduit par un temps de traitement de la base de données de complexité O(n) dépendant de la taille de la collection. Pour les autres opérations, la taille de la collection n'a aucun impact sur le temps de traitement (O(1)).

4.2.2. Modélisation Analytique

Dans cette section, nous proposons un modèle analytique d'une entité OM2M basé sur la théorie des files d'attente. Ce modèle fournit une représentation mathématique des métriques de performance telles que la taille des files d'attente, le débit et le temps de réponse en fonction de plusieurs paramètres (par exemple, taux d'arrivée et de service). Nous allons tout d'abord présenter la notation de Kendall [ken53] qui va être utilisée dans la représentation du réseau de files d'attente. Ensuite, le modèle analytique du Middleware OM2M est décrit. Les formules permettant de calculer les métriques de performance (nombre de clients, taille des files, temps de réponse, etc.) sont présentées. Enfin, nous décrivons l'approche suivie pour l'approximation des paramètres du modèle. Dans ce qui suit, seules les requêtes HTTP pour la gestion des ressources sont considérées.

Modélisation en réseau de files d'attente de la plateforme OM2M

La modélisation analytique de la plateforme OM2M est basée sur la théorie des files d'attente. Dans cette théorie, la notation de Kendall permet de décrire le système par une suite de six symboles a/s/C/K/m/Z. Ils dont définis comme suit :

1. a : représente le processus d'arrivée. Le plus fréquemment utilisé est celui selon Poisson ; les temps d'inter-arrivée sont distribués exponentiellement (M). Il y a aussi d'autres distributions telles que Erlang (Ek), hyperexponentielle (Hk), Déterministe (D), Générale (G) ;

2. s : indique la distribution du temps de service. Elle représente le temps passé en utilisant la ressource et est communément supposée être une variable aléatoire ; la distribution la plus commune est celle exponentielle (Markovienne) ;
3. C : fournit le nombre de *serveurs* dans le système. Dans un système informatique, ces serveurs représentent par exemple le nombre de processeurs sur une machine, le nombre de canaux de communication en entrée et sortie, etc. ;
4. K : indique la capacité du système ou le nombre de clients qui attendent dans la file. Elle peut être finie ou bien infinie (si la capacité est très grande) ;
5. m : correspond à la taille de la population représentant le nombre total d'utilisateurs potentiels. Elle peut être soit finie ou infinie (si la taille est très grande) ;
6. Z : indique la discipline de service utilisée ; les plus communes sont : First Come, First Served (FCFS) ou FIFO, Last Come, First Served (LCFS) ou LIFO, Round Robin (RR), Shortest Processing Time (SPT), Shortest Remaining Processing Time (SRPT), Biggest In First Served (BIFS).

Sur la base de l'analyse de la plateforme OM2M faite précédemment, et en considérant principalement les composants qui impactent les performances de la plateforme, le système peut être décomposé en trois composants principaux :

- **Serveur HTTP :** responsable de la réception des requêtes HTTP entrantes et de leur traduction en des requêtes REST génériques. Il lance un thread par requête ;
- **Ressources :** représente les ressources pour le traitement des requêtes. Ce composant est constitué, compte tenu des hypothèses considérées, d'une file d'attente des threads accédant au(x) processeur(s) ;
- **DATABASE :** partie persistante responsable du stockage, de la récupération, de la modification ou de la suppression des données.

La modélisation analytique d'un système réel est très complexe (voire impossible) si nous souhaitons prendre en considération tous les paramètres le représentant. Un compromis est donc nécessaire afin d'aboutir à un modèle correct permettant de décrire les phénomènes principaux impactant les performances du système réel.

En plus de la décomposition précédente, nous considérons les hypothèses simplificatrices suivantes : (1) les processeurs internes sont représentés par un unique processeur de capacité équivalente à la somme des capacités (n) ; (2) aucune priorisation n'est faite entre les différents threads : ceux-ci sont traités de manière identique par le processeur ; et (3) la mémoire est supposée infinie et son impact est négligeable sur les performances du système. Nous sommes conscients que ces hypothèses ont un impact sur l'écart entre notre modèle et la réalité. Cependant, elles nous permettent d'aboutir à une modélisation analytique. Dans la suite de ce chapitre, nous fournissons une mesure de cet écart, le calibrage du modèle étant mené de sorte à avoir un taux de précision supérieur à 85%.

Dans la théorie des files d'attente, une station est composée d'une ou plusieurs files d'attente attachées à un ou plusieurs serveurs. La Figure 4.5 illustre notre proposition de modèle de files d'attente d'une entité OM2M.

Ce modèle est constitué de trois stations. La station *Serveur HTTP* qui reçoit le trafic HTTP et instancie un thread par requête, le thread n'est libéré qu'à l'envoi de la réponse. La station *Ressources* est constituée d'une file *Threads* pour gérer leur accès au processeur (serveur *CPU*). Enfin, la station *DATABASE* pour la partie persistance de données.

Chaque requête HTTP est destinée à une classe donnée. Une classe est une collection (r) de taille i_r pour la gestion d'une ressource de type instance. Le nombre total de collection dans le système est R. Chaque station est composée d'une file d'attente et d'un serveur. Compte tenu de ces considérations, notre réseau constituant la plateforme OM2M est équivalent à un *réseau BCMP sans changement de classe* [bas75]. En effet, le réseau possède les caractéristiques suivantes : un seul serveur à chaque station, une capacité de stockage illimitée à toutes les stations et des routages probabilistes pour chaque classe de requêtes.

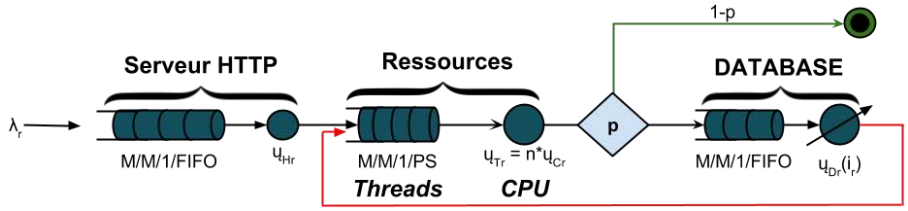

Figure 4.5 - Représentation du Modèle Analytique du Middleware OM2M

Pour une classe r, nous avons les entrées et les paramètres suivants : λ_r : taux d'arrivée du trafic ; μ_{Hr} : taux de service de la file *Serveur HTTP* ; N_T : nombre de threads dans l'ensemble du sous-système (files *Ressources* et *DATABASE*) ; μ_{Cr} : taux de service de chaque processeur ; μ_{Tr} : taux de service de la file *Ressources*, il correspond à $n*\mu_{Cr}$ où n est le nombre de processeurs dans le système ; μ_{Dr} : taux de service de la file *DATABASE* ; p : probabilité qu'une requête ne soit pas traitée complètement et revienne à la file des processeurs.

Le Tableau 4.1 donne une synthèse des configurations des stations. Le taux d'arrivée λ_r vers la plateforme OM2M peut être approché par une loi d'arrivée de type Poisson. En effet, même si dans un système IoT il y a différents types de trafic (périodique, requête/réponse et basé événement), la considération d'un trafic stochastique permet de refléter le comportement moyen de ce système au regard de l'aspect performances.

Stations	Serveur HTTP	Ressources	DATABASE
Processus d'arrivée	Markovien	Markovien	Markovien
Processus de Service	Markovien	Markovien	Markovien
Capacité de la file	Infinie	Infinie	Infinie
Nombre de Serveurs	1	1 (⇔n processeurs)	1
Discipline de Service	FIFO	PS	FIFO

Tableau 4.1 - Configurations des stations du Modèle Analytique

En ce qui concerne les taux de services (μ_{Hr}, μ_{Cr} et μ_{Dr}) des serveurs des différentes stations, à chaque discipline de service est associée une loi de service. Pour les files d'attente à discipline FIFO, elle impose que la loi soit exponentielle. Pour la file suivant une discipline PS, elle n'impose comme contrainte que la transformée associée à la loi de service doit être sous forme d'une fraction de Laplace rationnelle. Le service peut donc être distribué selon une loi générale. La capacité des files d'attente est supposée infinie. Les disciplines de gestion de chaque serveur sont : FIFO pour la station HTTP, PS (Process Sharing) pour la station *Ressources* et FIFO pour la station DATABASE. Le traitement des requêtes au niveau des stations *Serveur HTTP* et *Ressources* est indépendant de la collection cible. Les taux d'arrivée μ_{Hr} et μ_{Cr} deviennent donc μ_H et μ_C. Le paramètre p constitue la probabilité de retour à la station *Ressources* si la requête nécessite plus de traitement et de ressources pour la satisfaire.

Comme le temps de service de la station *DATABASE* est dépendant de la taille (i_r) de la collection r, nous pouvons l'écrire sous forme de $t_D(i_r) = t_{Dcte} + t_{Dvar}(i_r)$ où t_{Dcte} est le temps de

service pour la création d'une instance dans une collection vide, et $t_{Dvar}(i_r)$ est celui de la récupération d'une collection comportant i_r instances et qui peut être représenté par une fonction affine ($t_{Dvar}(i_r) = m*i_r$).

Le taux de service s'écrit donc comme $\mu_{Dr} = \mu_D(i_r) = \frac{1}{t_D(i_r)}$. Pour les requêtes de récupération, mise à jour ou suppression, la collection n'est pas récupérée et donc $t_{Dvar} = 0$ ms ($\mu_D = \mu_{Dcte}$).

Dans ce qui suit, nous déterminons les métriques de performances pour les différentes classes et sous la condition de stabilité en se basant sur [bay97].

Condition de stabilité

Soit e_{jr} le taux de visite des clients de la classe r à la station j. Cette quantité, dans le cas de notre réseau qui est considéré ouvert, s'interprète comme le nombre moyen de fois qu'un client de classe r visite la station j au cours de son séjour dans le système.

Le taux moyen d'arrivée des clients de classe r à une station j est :

$$\lambda_{jr} = \lambda_r e_{jr} \quad (4.1)$$

Le taux d'arrivée des clients, toutes classes confondues, à la station j est donc :

$$\lambda_j = \sum_{r=1}^{R} \lambda_{jr} \quad (4.2)$$

Alors, la proportion de clients de classe r qui arrivent à la station j, ou encore la probabilité pour qu'un client qui arrive à la station i soit de classe r, est :

$$q_{jr} = \frac{\lambda_{jr}}{\lambda_j} = \frac{\lambda_{jr}}{\sum_{s=1}^{R} \lambda_{js}} \quad (4.3)$$

Pour un client de classe r, il induira une charge de travail moyenne à la station j de : $\frac{1}{\mu_{jr}}$.

Un client d'une classe quelconque induira donc à chaque passage à la station j une charge moyenne de travail de : $\sum_{r=1}^{R} \frac{q_{jr}}{\mu_{jr}}$.

Le taux moyen de service de la station j est alors l'inverse de la quantité précédente, ce qui donne :

$$\mu_j = \frac{1}{\sum_{r=1}^{R} \frac{q_{jr}}{\mu_{jr}}} = \frac{\lambda_j}{\sum_{r=1}^{R} \frac{\lambda_{jr}}{\mu_{jr}}} \quad (4.4)$$

Pour la condition de stabilité, il faudra que le taux d'arrivée des clients (quelles que soient leurs classes d'appartenance) à la station j soit inférieur au taux de service ($\lambda_j < \mu_j$).

En remplaçant les équations (4.2) et (4.4) dans la condition de stabilité précédente, nous déduisons que le réseau est stable si pour toute station j : $\sum_{r=1}^{R} \frac{\lambda_{jr}}{\mu_{jr}} < 1$.

Taux de visite de chaque station

Rappelons l'expression du taux de visite des clients de classe r pour une station j :

$$e_{jr} = p_{0jr} + \sum_{k=1}^{M} e_{kr} p_{kjr} \; ; \; j = 1, \ldots, M \quad (4.5)$$

Avec :

- p_{jkr} : probabilité qu'un client de classe r sorte de la station j pour entrer à la station k avec une classe r
- p_{0jr} : probabilité pour qu'un client de classe r qui entre dans le système se rende à la station j
- p_{j0r} : probabilité qu'un client de classe r sorte de la station j pour quitter le système

En appliquant cette formule à notre réseau de file d'attente, nous obtenons pour une classe de client r les mêmes expressions que précédemment pour le réseau mono-classe. Pour chaque station, nous avons les taux suivants (respectivement pour Serveur HTTP, Ressources et DATABASE) :

- $e_{Hr} = 1$: une unique file d'attente d'entrée (*Serveur HTTP*) au système pour tous les clients ;
- $e_{Tr} = e_{Hr} + e_{Dr}$: tout le trafic de la file *Serveur HTTP* et *DATABASE* entre dans la file *Ressources ;*
- $e_{Dr} = pe_{Tr}$: le trafic de la file *Ressources* entre dans la file *DATABASE* avec la probabilité p

Alors, nous avons :

$$e_{Hr} = 1, \; e_{Tr} = \frac{1}{1-p}, \; e_{Dr} = \frac{p}{1-p}$$

Pour des requêtes de classes différentes arrivant chacune avec un taux λ_r, il est équivalent de considérer que l'ensemble des requêtes arrivent de l'extérieur selon un processus de Poisson de taux global λ. Si l'on note α_r la probabilité qu'une requête qui arrive se voit attribuer une classe r (appartenant aux classes existantes), puisque la décomposition probabiliste ou la superposition préservent la nature Poissonnienne des processus, les taux d'arrivée sont alors reliés par :

$$\lambda_r = \lambda \alpha_r \quad (4.6)$$

La probabilité qu'un client qui arrive dans le système se voit attribuer la classe r et se rende à la station j est :

$$p'_{0jr} = \alpha_r p_{0jr} \quad (4.7)$$

Notons e'_{jr} le taux de visite de la station j par des clients de classe r. Compte tenu de l'équation précédente, e'_{jr} a l'expression suivante :

$$e'_{jr} = p'_{0jr} + \sum_{k=1}^{M} e'_{kr} p_{0kr} \; ; \; j = 1, \ldots, M \quad (4.8)$$

L'expression e'_{jr} peut s'interpréter comme étant le nombre moyen de fois qu'un client (toutes classes confondues) qui arrive dans le système passe par la station i pondéré par la probabilité que ce client se soit attribuer la classe r. Ce qui permet de lier les deux quantités e_{jr} et e'_{jr} par la relation suivante :

$$e'_{jr} = \alpha_r e_{jr} \quad (4.9)$$

En remplaçant les valeurs précédemment obtenues dans l'équation (4.9), nous avons :

- $e'_{Hr} = \alpha_r e_{Hr} = \alpha_r$
- $e'_{Tr} = \alpha_r e_{Tr} = \frac{\alpha_r}{1-p}$
- $e'_{Dr} = \alpha_r e_{Dr} = \frac{\alpha_r p}{1-p}$

Régime permanent

Si nous ne nous intéressons qu'à une classe de clients donnée, une simplification du théorème BCMP est possible. Soit $p_{jr}(n_{jr})$ la probabilité marginale pour que la station j contienne n_{jr} clients d'une classe r donnée, quel que soit le nombre de clients des autres classes présents à la station (et quel que soit l'état des autres stations du réseau). Ces probabilités s'expriment à l'aide de la relation suivante (puisque nous n'avons que des files de types FIFO et PS) :

$$p_{jr}(n_{jr}) = (1 - \hat{\rho}_{jr})\hat{\rho}_{jr}^{n_{jr}} \text{ où } \hat{\rho}_{jr} = \frac{\lambda_r e_{jr}}{\hat{\mu}_{jr}} \quad (4.10)$$

Avec :

- $\hat{\mu}_{jr} = \mu_j - \sum_{s=1, s\neq r}^{R} \lambda_s e_{js}$, si la station est de type FIFO (files *Serveur HTTP* et *DATABASE*)
- $\hat{\mu}_{jr} = \mu_{jr}(1 - \sum_{s=1, s\neq r}^{R} \frac{\lambda_s e_{js}}{\mu_{js}})$, si la station est de type PS (file *Ressources*)

Métriques de performances

Pour une station j, la probabilité marginale $p_{jr}(n_{jr})$ possède l'expression des probabilités stationnaires d'une file M/M/1 mono-classe ayant un taux de service $\hat{\mu}_{jr}$et soumise à un processus d'arrivée poissonien de taux $\lambda_{jr} = e_{jr}\ \lambda_r$. Les paramètres de performances (débit moyen, nombre de clients moyen et temps de réponse moyen) s'en déduisent donc directement.

Débit moyen

L'expression du débit moyen de client de classe r pour une station j est :

$$X_{jr} = \lambda_{jr} = e_{jr}\lambda_r \quad (4.11)$$

En instanciant cette expression générale (4.12) pour les différentes stations de notre réseau, nous obtenons :

- pour la station *Serveur HTTP* : $X_{Hr} = \lambda_{Hr} = e_{Hr}\lambda_r = \lambda_r = \alpha_r\lambda$
- pour la station *Ressources* : $X_{Tr} = \lambda_{Tr} = e_{Tr}\lambda_r = \frac{\lambda_r}{1-p} = \frac{\alpha_r\lambda}{1-p}$
- pour la station *DATABASE* : $X_{Dr} = \lambda_{Dr} = e_{Dr}\lambda_r = \frac{\lambda_r p}{1-p} = \frac{\alpha_r p\lambda}{1-p}$

Nombre moyen de clients

Le nombre moyen de clients de classe r dans une station j est :

$$Q_{jr} = \frac{\hat{\rho}_{jr}}{1-\hat{\rho}_{jr}}, \text{ avec } \hat{\rho}_{jr} = \frac{\lambda_r e_{jr}}{\hat{\mu}_{jr}} \quad (4.12)$$

En remplaçant $\hat{\rho}_{jr}$par son expression, nous avons :

$$Q_{jr} = \frac{\hat{\rho}_{jr}}{1-\hat{\rho}_{jr}} = \frac{\lambda_r e_{jr}}{\hat{\mu}_{jr} - \lambda_r e_{jr}} \quad (4.13)$$

En instanciant cette nouvelle expression générale pour les différentes stations de notre réseau, nous obtenons :

- **Station *Serveur HTTP***

Cette station est de type FIFO, nous avons donc :

$$\hat{\mu}_{Hr} = \mu_H - \sum_{s=1,s\neq r}^{R} \lambda_s e_{Hr} \quad (4.14)$$

Nous avons précédemment obtenu $e_{Hr} = 1$.

$$\hat{\mu}_{Hr} = \mu_H - \sum_{s=1,s\neq r}^{R} \lambda_s \quad (4.15)$$

En considérant $\lambda = \sum_{i=1}^{R} \lambda_i$ et en remplaçant alors la valeur $\hat{\mu}_{Hr}$ dans l'expression du nombre moyen de visites (4.13), nous obtenons :

$$Q_{Hr} = \frac{\lambda_r e_{Hr}}{\hat{\mu}_{ir} - \lambda_r e_{Hr}} = \frac{\lambda_r}{\mu_H - \lambda} \quad (4.16)$$

- **Station *Ressources***

Cette station est de type PS, nous avons donc :

$$\hat{\mu}_{Tr} = \mu_{Tr}\left(1 - \sum_{s=1,s\neq r}^{R} \frac{\lambda_s e_{Ts}}{\mu_{Ts}}\right), \text{ avec } \mu_{Tr} = n*\mu_{Cr} \quad (4.17)$$

Nous avons précédemment obtenu $e_{Tr} = \frac{1}{1-p}$. En considérant $\mu_{Cr} = \mu_{Cs} = \mu_C$, nous obtenons donc :

$$\hat{\mu}_{Tr} = \mu_{Tr}\left(1 - \frac{1}{1-p}\sum_{s=1,s\neq r}^{R} \frac{\lambda_s}{\mu_{Ts}}\right) = n\mu_C - \frac{1}{1-p}\sum_{s=1,s\neq r}^{R} \lambda_s \quad (4.18)$$

En remplaçant alors la valeur $\hat{\mu}_{Tr}$ dans l'expression du nombre moyen de visites (4.13), nous obtenons :

$$Q_{Tr} = \frac{\lambda_r e_{Tr}}{\hat{\mu}_{Tr} - \lambda_r e_{Tr}} = \frac{\lambda_r}{n\mu_C(1-p) - \lambda}, avec\ \lambda = \sum_{k=1}^{R} \lambda_k \quad (4.19)$$

- **Station *DATABASE***

Cette station est de type FIFO, nous avons donc le taux de service suivant :

$$\hat{\mu}_{Dr} = \mu_D - \sum_{s=1,s\neq r}^{R} \lambda_s e_{Dr} \quad (4.20)$$

Nous avons précédemment obtenu : $e_{Dr} = \frac{p}{1-p}$.

En considérant $\lambda = \sum_{i=1}^{R} \lambda_i$, nous obtenons donc :

$$\hat{\mu}_{Dr} = \mu_{Dr} - \frac{p}{1-p}\sum_{s=1,s\neq r}^{R} \lambda_s = \mu_{Dr} - \frac{p}{1-p}(\lambda - \lambda_r) \quad (4.21)$$

Puisque $\mu_{Dr} = \mu_D(i_r)$, nous avons :

$$\hat{\mu}_{Dr} = \hat{\mu}_D(i_r) \;=\; \mu_D(i_r) - \frac{p}{1-p}(\lambda - \lambda_r) \quad (4.22)$$

En remplaçant alors la valeur $\hat{\mu}_{Dr}$ dans l'expression du nombre moyen de visites (4.13), nous obtenons :

$$Q_{Dr} = Q_D(i_r) = \frac{\lambda_r e_{Dr}}{\hat{\mu}_D(i_r) - \lambda_r e_{Dr}} = \frac{p\lambda_r}{(1-p)\mu_D(i_r) - p\lambda} \quad (4.23)$$

Nombre maximal de clients

Le nombre maximal de clients de classe r entre les files *Ressources* et *DATABASE* est limité par le nombre de threads N_T :

$$Q_{Tr} + Q_{Dr} \leq N_T \quad (4.24)$$

Pour que l'expression précédente soit valide, il faut que le taux d'arrivée vérifie la condition suivante :

$$\lambda_r \leq \lambda_l(i_r)$$

Avec :

$$\lambda_l(i_r) = \frac{1-p}{2p(2+N_T)}[(1+N_T)(\mu_D(i_r) + np\mu_C) - \sqrt{(\mu_D(i_r) - np\mu_C)^2(2+N_T)N_T + (\mu_D(i_r) + np\mu_C)^2}]$$

Il faudra donc que λ soit inférieur à λ_l pour que le trafic ne dépasse pas le nombre de threads utilisés en moyenne.

Temps de réponse moyen

L'expression générale du temps de réponse moyen de la station j est :

$$R_{jr} = \frac{Q_{jr}}{X_{jr}} \quad (4.25)$$

Pour la station *Serveur HTTP*, le temps moyen de service est :

$$R_{Hr} = \frac{Q_{Hr}}{X_{Hr}} = \frac{1}{\mu_H - \lambda} \quad (4.26)$$

Pour la station *Ressources*, le temps moyen de service est :

$$R_{Cr} = \frac{Q_{Cr}}{X_{Cr}} = \frac{1-p}{\mu_{Cr}(n(1-p) - \sum_{s=1, s\neq r}^{R} \frac{\lambda_s}{\mu_{Cs}}) - \lambda_r} = \frac{1-p}{n\mu_C(1-p) - \lambda} \quad (4.27)$$

Pour la station *DATABASE*, le temps moyen de service est :

$$R_{Dr} = R_D(i_r) = \frac{Q_D(i_r)}{X_{Dr}} = \frac{1-p}{(1-p)\mu_D(i_r) - p\lambda} \quad (4.28)$$

Le temps de service moyen de l'ensemble du système pour le traitement d'une requête de classe r est égal à $R_{OM2M}(r)$, avec :

$$R_{OM2M}(r) = R_{Hr} + R_{Tr} + R_{Dr} \quad (4.29)$$

Nous avons donc :

$$R_{OM2M}(r) = R_{OM2M}(i_r) = \frac{1}{\mu_H - \lambda} + \frac{1-p}{n\mu_C(1-p) - \lambda} + \frac{1-p}{(1-p)\mu_D(i_r) - p\lambda} \quad (4.30)$$

$$avec\ \lambda = \sum_{k=1}^{R} \lambda_k \ et\ \mu_D(i_r) = \frac{1}{t_{DCte} + m * i_r}$$

Le temps de réponse global pour la création d'une instance est donc dépendant de la taille (i_r) de la collection (r). En ce qui concerne les requêtes GET, i_r est égal à 0 et le temps de réponse global n'est plus dépendant de la taille de la collection.

Toutes ces métriques de performance dépendent de trois types de paramètres : (1) paramètres d'entrée variables (taux d'arrivée), (2) paramètres d'entrée statiques (nombre de processeurs), et (3) paramètres internes statiques (taux de service et probabilité de retour). Afin de calculer ces métriques de performances, la détermination des paramètres internes est nécessaire. Dans ce qui suit, nous présentons l'approche suivie pour l'approximation de ces paramètres (μ_H, μ_C, μ_D et p) et la calibration du modèle.

Approche d'approximation des paramètres du modèle

L'approche adoptée pour l'approximation des paramètres (μ_H, μ_C, μ_D et p) se base sur une plateforme d'émulation. Cette plateforme permet de générer un trafic suivant différents profils : Protocole (HTTP ou CoAP), Méthode (GET, POST, PUT et DELETE), une ou plusieurs sources (injecteurs de trafic). L'objectif étant, compte tenu des paramètres internes (nombre de cœurs dans notre cas), de mesurer le temps de réponse pour certains taux d'arrivée. Ensuite, de construire un système non-linéaire à optimiser et où les inconnues sont les paramètres à approcher (μ_H, μ_C, μ_D et p).

Dans notre cas, le trafic généré est de type HTTP suivant une loi de Poisson vers une entité OM2M déployée sur une plateforme réelle (serveur ou gateway IoT). Ce trafic est destiné à la ressource <contentInstance> de l'arbre de ressources. Dans notre scénario, différents taux d'arrivée sont considérés. L'émulateur de trafic (décrit dans les scénarios de validation des mécanismes du chapitre 2) est utilisé pour la génération d'un trafic stochastique depuis p injecteurs et avec un type de requêtes donné. Pour chaque taux d'arrivée, l'émulateur mesure par la suite le temps de réponse moyen (RTT_{MES}). L'objectif étant de tracer l'évolution du temps de réponse moyen (RTT_{MES1}, RTT_{MES2}, RTT_{MES3}, …, RTT_{MESn}) en fonction des taux d'arrivée (λ_1, λ_2, λ_3, …, λ_n) pour la gestion de l'instance.

En se basant sur l'évolution du temps de réponse pour les différentes valeurs de taux d'arrivée et la représentation analytique de ce temps de réponse via le modèle (4.30), nous construisons alors un système d'équations non linéaire dont les inconnues sont les paramètres à approcher (μ_H, μ_C, μ_D et p).

Sous le logiciel Matlab, la méthode de résolution de ce système d'équation se base sur la fonction $F(j) = RTT_{MOD}(\lambda_j) - RTT_{MEASj}$, et la recherche d'un minimum local $x = [\mu_H, \mu_C, \mu_D, p]$. Cette approximation est réalisée via l'algorithme de "Trust-Region-Reflective" [con00].

Une fois le calibrage est obtenu, les paramètres sont par la suite utilisés pour le calcul du temps de réponse analytique en fonction du taux d'arrivée. Dans la section suivante, nous appliquons cette technique d'approximation pour une gateway IoT déployant le Middleware OM2M.

4.2.3. Paramètres du modèle pour le cas d'une Gateway

Dans cette section, nous considérons une gateway de type BeagleBone Black disposant d'un seul processeur (donc n = 1). L'émulateur induit un délai entre les injecteurs et la gateway inférieur à 10ms, ce qui est négligeable comparé au temps de réponse mesuré. Les paramètres sont déterminés pour les requêtes HTTP GET et POST. Ces requêtes sont adressées à la même collection, il n'y a donc qu'une seule classe de clients. Les valeurs obtenues pour les paramètres sont ensuite validées en comparant l'évolution du temps de réponse mesuré avec celle du temps

de réponse analytique obtenu en remplaçant les paramètres par leurs valeurs.

Paramètres du Modèle pour les requêtes GET

En nous basant sur la plateforme d'émulation, nous mesurons les temps de réponse moyens pour la récupération d'instances dans une collection. La Figure 4.6 représente l'évolution du temps de réponse moyen en fonction du taux d'arrivée. Pour des taux d'arrivée faibles (inférieur à 10 requêtes par seconde), le temps de réponse moyen reste presque stable. Avec des taux d'arrivée plus élevés (supérieur à 20 requêtes par seconde), le temps de réponse prend une allure exponentielle. Cette évolution permet de déduire que le Middleware est non *scalable* et peut induire des temps de réponse très élevés pour des taux d'arrivée élevés (plusieurs secondes à dizaines de secondes). Il peut ainsi causer la dégradation de la QoS.

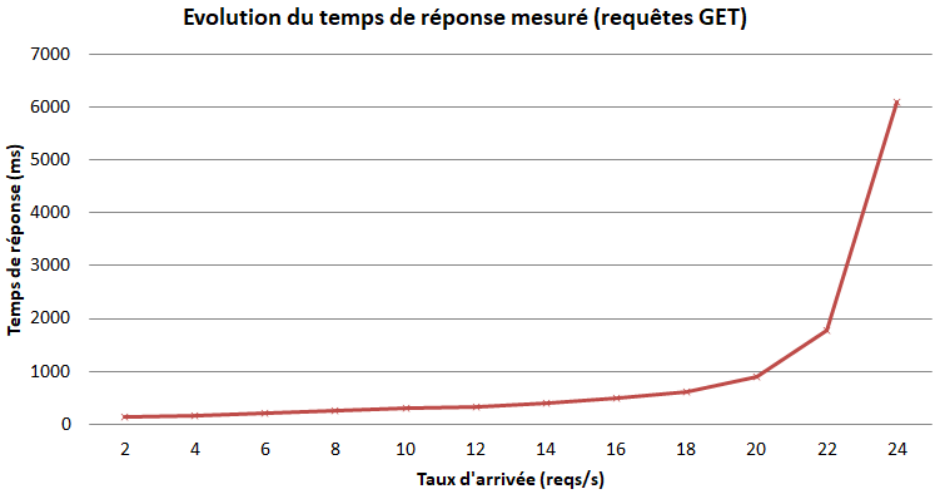

Figure 4.6 - Evolution du Temps de réponse mesuré pour les requêtes GET

Afin de construire le système non linéaire et appliquer l'approche d'approximation, plusieurs valeurs du RTT sont utilisées. Nous nous basons essentiellement sur les valeurs de RTT des taux d'arrivée λ_1=2, λ_2=10, λ_3=18, λ_4=22 et λ_5=24 reqs/s qui décrivent au mieux le comportement du système (notamment l'évolution exponentielle). Les valeurs obtenues pour les paramètres du modèle sont : μ_H = 24,6 reqs/s, μ_C = 37,2 reqs/s, μ_D = 13 reqs/s et p = 0,3446.

La Figure 4.7 représente l'évolution du temps de réponse mesuré et celui analytique obtenu via le modèle. Les deux métriques suivent à peu près la même évolution. Elles ont un taux d'erreur moyen de 13,27%, qui est inférieur à 15% et nous permet donc de valider ce calibrage.

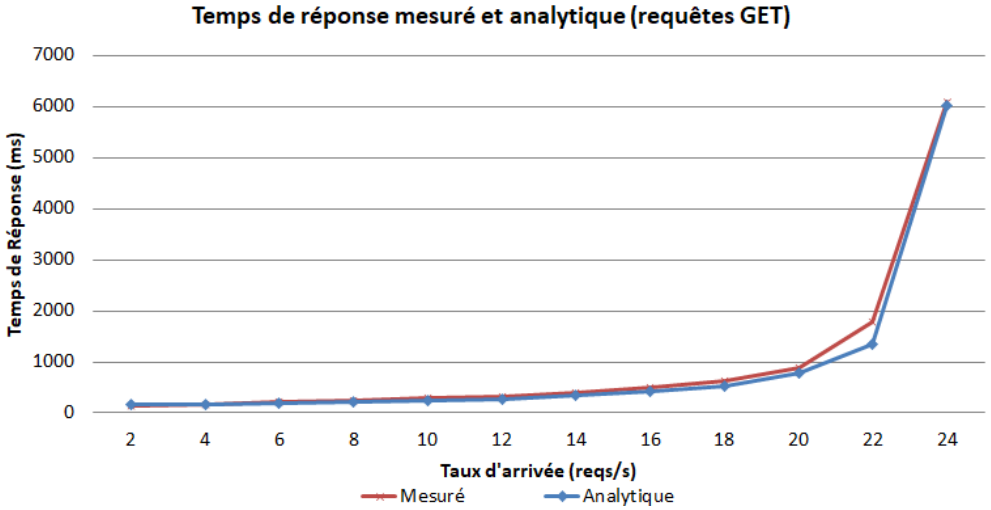

Figure 4.7 - Comparaison entre le temps de réponse mesuré et calculé pour les requêtes GET

En utilisant ces paramètres, nous pouvons donc exprimer d'autres métriques de performances en fonction du taux d'arrivée. La Figure 4.8 illustre l'évolution du nombre de requêtes moyen dans chaque file d'attente mais aussi au total dans tout le système.

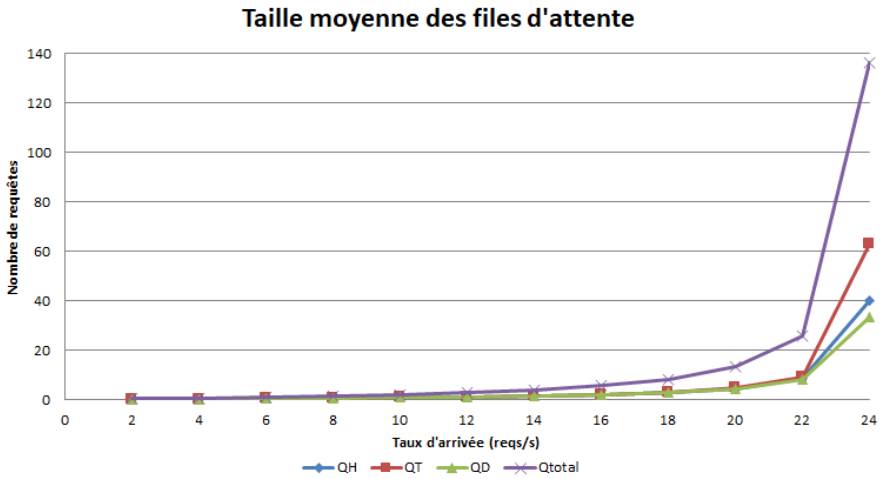

Figure 4.8 - Nombre moyen analytique de requêtes de type GET

Paramètres du Modèle pour les requêtes POST

Les requêtes POST sont envoyées par l'émulateur vers la même collection. Comme indiqué précédemment, le temps de service de la base de données dépend de la taille de la collection. Afin de déterminer les paramètres t_{Dcte} et t_{Dvar} qui constituent ce temps de service, une technique de régression linéaire est utilisée sur le temps de traitement du DAO pour la récupération de la collection (Figure 4.4). Suivant cette technique, nous obtenons les valeurs suivantes : t_{Dcte} = 19,96 ms et $t_{Dvar}(i)$ = 0,1879*i ms. Pour la détermination des autres paramètres, le temps de réponse moyen est mesuré pour 1000 échantillons (Figure 4.9). En suivant l'approche d'approximation, nous obtenons les valeurs suivantes : μ_H = 10,11 reqs/s, μ_C = 8,799 reqs/s et p = 0,55174.

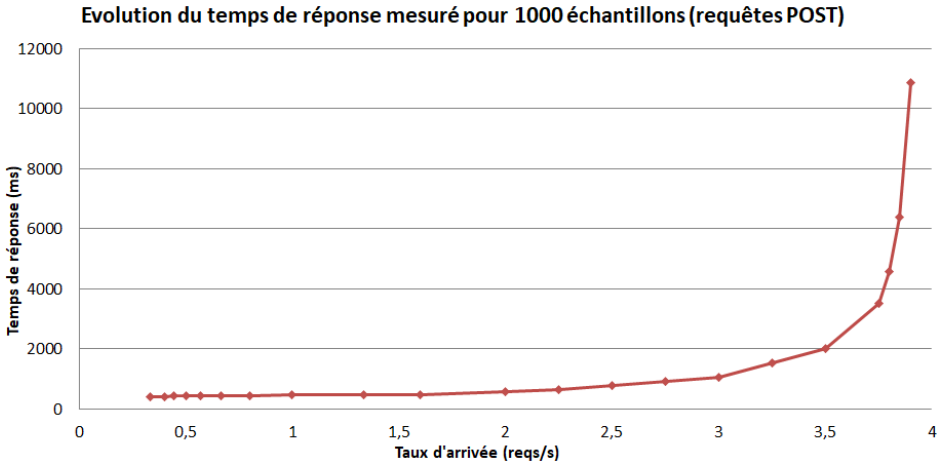

Figure 4.9 - Temps de réponse mesuré pour des requêtes POST (1000 échantillons)

Afin de valider ces valeurs de paramètres, nous calculons le taux d'erreur entre le temps de réponse mesuré et calculé pour différents d'échantillons. Pour 500 échantillons, le taux d'erreur moyen est de 8,15%. Avec 1000 échantillons, l'erreur moyenne entre les deux temps de réponse (Figure 4.10) prend la valeur de 12,718%. Pour 1500 et 2000 échantillons, les taux d'erreurs sont respectivement de 5,639% et 4,119%. A travers ces valeurs de calibration, nous pouvons constater que nous avons une bonne approximation des performances de la plateforme OM2M.

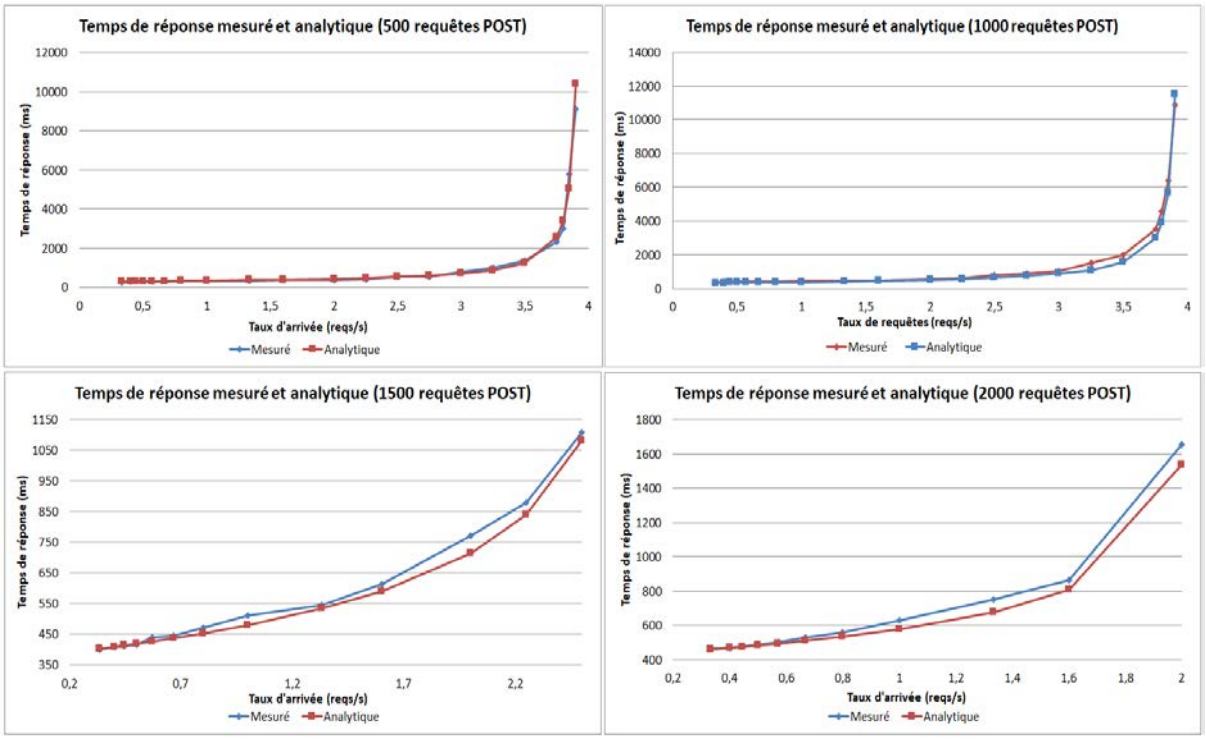

Figure 4.10 - Comparaison entre le temps de réponse mesuré et calculé pour les requêtes POST - (500, 1000, 1500 et 2000 échantillons)

Pour la validation de ces paramètres dans le cas de requêtes POST vers différentes collections, nous observons l'évolution du temps de réponse de requêtes vers une collection donnée en présence d'un trafic vers d'autres collections. Dans notre cas, nous considérons les deux trafics suivants :

- **trafic 1 :** requêtes POST avec un taux d'arrivée de 0,4 reqs/s vers la collection du container 1, qui contient déjà 500 instances ;
- **trafic 2 :** requêtes POST vers la collection du container 2 qui ne contient aucune instance. Le taux d'arrivée est de 0,8 reqs/s.

La Figure 4.11 représente la comparaison entre les temps de réponse calculé et mesuré du trafic 1 dans ce scénario (donc à partir de la 500ème requête). Pour chaque requête de création, le temps de réponse calculé reste proche de celui mesuré avec un taux d'erreur de 2,85%.

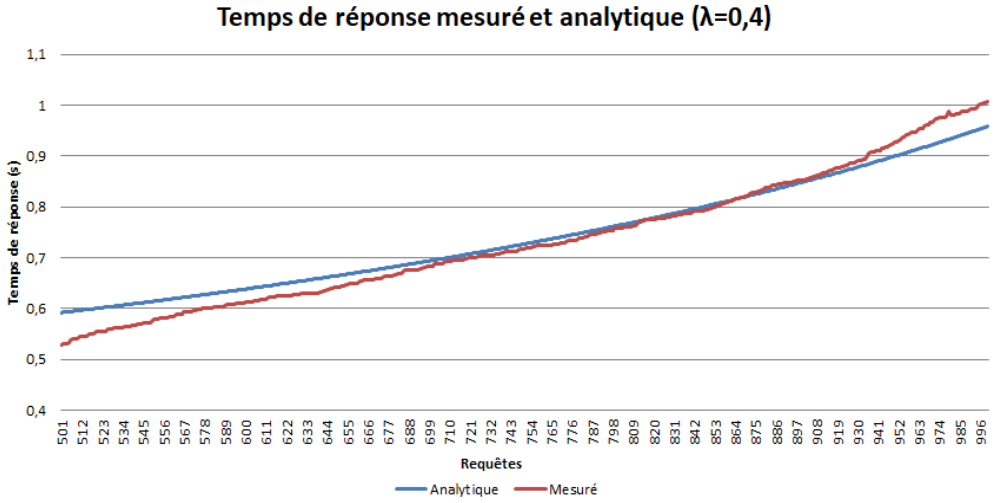

Figure 4.11 - Validation du Modèle général (plusieurs classes) pour les requêtes POST

Une analyse des paramètres pour les requêtes GET et POST conduit à une observation importante qui traduit le comportement réel de la plateforme OM2M. Pour les requêtes GET, les taux de services sont de l'ordre de (μ_H = 24,6, μ_C = 37,2 et μ_D = 13 reqs/s). Ceux des requêtes POST sont plus élevés (μ_H = 10,11, μ_C = 8,799 and $\mu_{DAve1000}$ = 6,05 reqs/s). Cette différence est expliquée par le fait que les requêtes POST sont plus coûteuses en temps de traitement, dû au fait qu'elles contiennent plus d'informations (par exemple, le corps de la requête qui comporte la représentation de l'instance à créer). Ainsi dans la plateforme OM2M, le traitement des requêtes POST nécessite plus de temps de traitement au niveau des différentes stations et conduit donc à des taux de services plus bas. Pour la probabilité de retour p, les requêtes POST (p = 0,5517) réalisent plus d'opérations avec la base de données (par exemple, les droits d'accès, la récupération du container et de la collection, le stockage de l'instance) et ont donc besoin de plus de ressources que les requêtes GET (p = 0,3446) qui réalisent moins d'opérations.

L'élaboration de ce modèle analytique permet d'enrichir la base de connaissance du gestionnaire autonomique esclave (AMS). Pour une gestion orientée QoS, le modèle donne une représentation du système géré (ici, une entité Middleware) sous forme de performances, notamment en terme de temps de réponse. Ce modèle analytique est utilisable par plusieurs composants de la boucle autonomique de l'AMS de chaque entité Middleware. Dans ce qui suit, nous montrons comment l'intégrer dans le raisonnement du composant de monitoring pour guider la génération des symptômes relatifs à la dégradation de la QoS. Nous en montrerons également l'utilisation dans le cadre du composant de planification en Chapitre 5 de ce mémoire.

4.3. COMPOSANT DE MONITORING POUR L'AMS

La supervision constitue la première étape du processus de gestion. Cette tâche est réalisée par le composant de monitoring de la boucle autonomique. Il supervise le système géré et lève des alarmes en cas de non satisfaction du besoin en QoS pour une ou des applications critiques. Ce monitoring peut suivre deux modes. Le mode *actif* consiste en l'interrogation de la plateforme à intervalle régulier afin de soulever les métriques de performance ; dans le mode *passif*, c'est le système géré qui signale sa dégradation. Ces deux modes sont considérés dans notre composant de monitoring à différents niveaux de supervision de l'entité Middleware.

Il existe aussi plusieurs approches de monitoring. Nous trouvons essentiellement l'approche *réactive* et l'approche *proactive*. L'approche réactive permet de lever des symptômes en cas de dégradation de la QoS. L'avantage de cette approche est sa facilité de conception. Cependant, la QoS sera déjà dégradée avant le début de la gestion. L'approche proactive vise à signaler une dégradation future avant son arrivée, ce qui est avantageux pour les applications critiques mais qui pose une difficulté relative à sa complexité d'élaboration. Trois types de modèles peuvent être utilisées pour cette approches. Les modèles prédictifs analysent les événements passés afin d'estimer le futur de manière probabiliste. Les modèles descriptifs permettent de quantifier les relations entre les événements et de les classer en groupes. A l'opposé des modèles prédictifs, les modèles descriptifs identifient plusieurs relations entre les événements. Enfin, les modèles de décision ou de prise de décision [hol89] élaborent les relations entre tous les éléments d'une décision (les événements connus, la décision et les résultats attendus de la décision) afin de prédire les résultats de ces décisions.

Dans cette section, nous exposons l'application du modèle analytique présenté précédemment dans la phase de Monitoring. L'objectif de cette utilisation est de minimiser le coût du monitoring sur le système géré. Le modèle est utilisé dans un premier temps dans une approche

de monitoring réactif, en couplage à des techniques de traitement d'événements complexe pour détecter les dégradations de la QoS. Dans un second temps, suivant une approche de monitoring proactif basé prédiction, nous montrons comment coupler le modèle analytique avec un modèle de prédiction ARMA permettant d'estimer le taux d'arrivée des requêtes sur une entité Middleware.

Dans ce qui suit, nous détaillons tout d'abord les capteurs logiques permettant de collecter les métriques de performances ainsi que les techniques utilisées pour cette collecte (section 4.3.1). Nous proposons ensuite en section 4.3.2 l'architecture fonctionnelle du composant de monitoring de l'AMS qui prend en considération les différents modes et approches de monitoring. Nous détaillons dans cette section les modes actifs et passifs de monitoring et leur utilisation dans notre contexte. Enfin, la section 4.3.3 présente nos approches de monitoring réactif et prédictif adoptées dans cette thèse, ainsi que les modèles et techniques attenantes.

4.3.1. Capteurs logiques de supervision

Etant donné que le Middleware OM2M s'exécute sur une machine virtuelle Java (JVM), plusieurs niveaux de gestion (incluant en premier lieu le monitoring) sont à considérer (Figure 4.12) :

- le premier niveau concerne le logiciel OM2M lui-même. L'objectif est de collecter les métriques nécessaires en vue des actions d'adaptation adéquates sur chaque plugin. A ce niveau, les métriques peuvent être par exemple le temps de réponse du système et de chaque plugin, le taux de pertes du système, ou encore l'état du plugin (activé, désactivé, etc.) ;
- le logiciel OM2M étant développé en Java, le deuxième niveau de monitoring porte sur la JVM. Cette dernière offre les ressources nécessaires à l'exécution de OM2M ; les métriques concernées sont la CPU, la mémoire, le nombre de threads et le nombre de classes chargées ;
- le troisième niveau de monitoring est spécifique à la plateforme Cloud dans laquelle est susceptible d'être déployée l'entité Middleware ciblée. Il porte sur les métriques associées aux ressources allouées à la machine virtuelle hébergeant le logiciel OM2M (état de la VM, mémoire utilisée, CPU, utilisation disque dur, etc.) ;
- le quatrième niveau traite la machine physique hébergeante, et concerne des métriques telles que l'état de la machine (en marche ou en panne), les processeurs utilisés sur le nombre total, la RAM utilisée, ou encore le pourcentage d'espace de stockage utilisé.

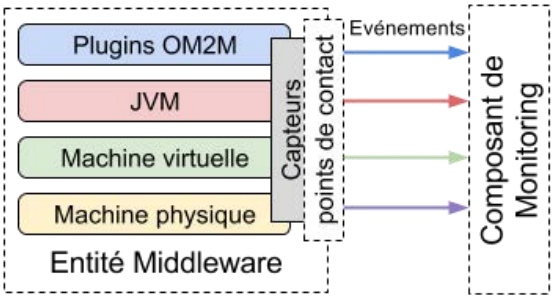

Figure 4.12 - Niveaux de gestion considérés du Middleware OM2M

Le Middleware OM2M étant basée sur le langage Java, les extensions JXM (*Java Management Extensions*) de gestion qu'offre ce langage peuvent être utilisées par le gestionnaire autonomique. Ces JMX [jmx17] permettent une interrogation non intrusive via un accès distant à la plateforme gérée grâce à un agent et des MBeans (*Management Beans*). Conceptuellement,

les MBeans sont des ressources utilisées dans les JVM pour : recueillir des statistiques sur des problèmes tels que la performance ou l'utilisation des ressources (*pull*) ; obtenir et configurer des propriétés d'application (*push/pull*) ; et enfin notifier des événements comme les défauts ou les changements d'état (*push*).

La Figure 4.13 représente une vue partielle orientée implémentation de l'architecture des capteurs de l'entité gérée au niveau OM2M. Cette architecture comporte les composants suivants :

- **MBeanAgent** : il comporte un serveur HTTP permettant d'exposer l'API d'interaction via un port spécifique (pMC). Il réalise une traduction des requêtes HTTP reçues en des invocations Java internes ;
- **MBeanServer** : c'est l'intermédiaire entre le MBeanAgent et les MBeans. Il permet d'exposer les fonctionnalités des MBeans enregistrés ;
- **OM2MMBeansBundle** : il inclut tous les MBeans permettant le relevé des métriques du logiciel OM2M. Il peut invoquer les plugins OM2M afin de relever les différentes informations (temps de réponse, plugins, etc.). Les MBeans sont enregistrés auprès du serveur.

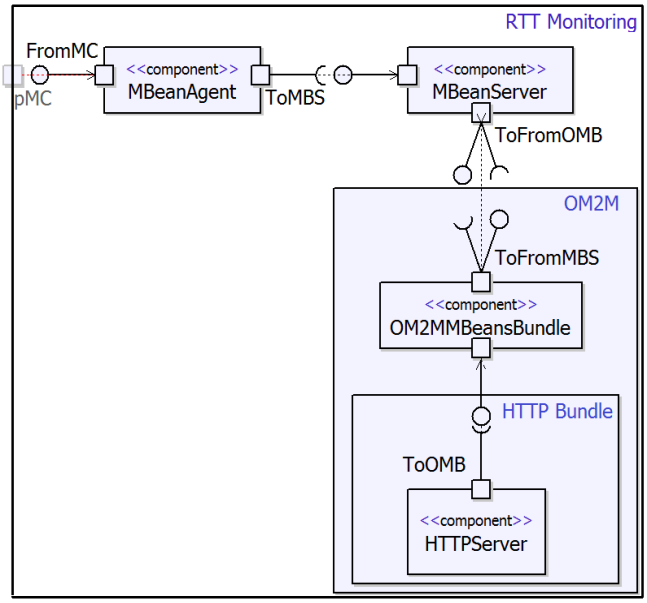

Figure 4.13 - Architecture des capteurs logiques pour le niveau Middleware

L'utilisation des MBeans (*Managed Beans*) permet de réduire considérablement la charge du monitoring sur le système géré. Pour l'observation du système et la collecte des informations, nous nous sommes basé sur l'outil Jolokia [jol17] qui est une implémentation du standard JMX.

4.3.2. Architecture fonctionnelle du composant de Monitoring

Le composant de monitoring est focalisé sur la supervision des métriques ayant une relation directe avec le besoin en QoS à satisfaire. Dans notre cas, ce composant considère les événements relevant directement du besoin en temps de réponse. Il détecte la dégradation de ce besoin applicatif au regard d'un seuil critique. Inspiré d'un effort similaire effectué de manière générique pour l'intégralité de la boucle MAPE-K [ben15], notre modèle architectural est focalisé sur le composant de monitoring au regard des besoins en QoS. La Figure 4.14 fournit une description en UML de l'architecture fonctionnelle de ce composant.

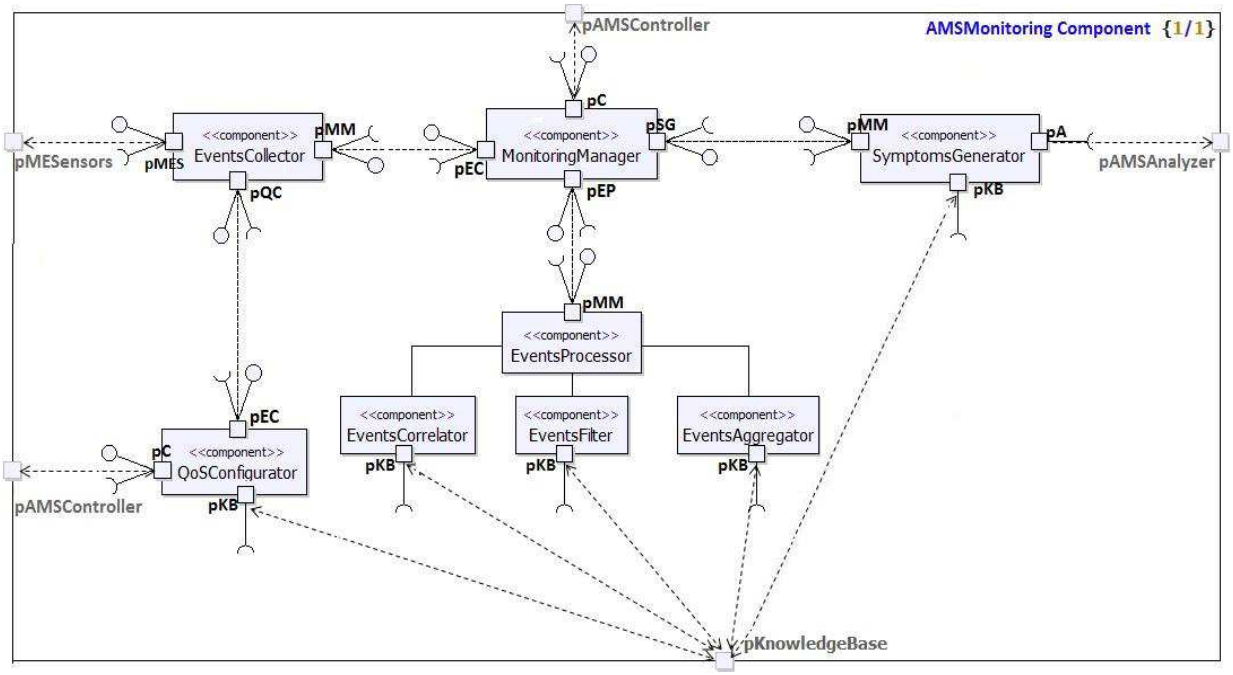

Figure 4.14 - Architecture fonctionnelle du composant de Monitoring

Cette architecture suit le modèle des architectures orientées événements [mic06, etz10]. Les composants principaux de cette architecture sont décrits ci-après.

Gestionnaire de Monitoring

Le composant *MonitoringManager* constitue le noyau du monitoring. Il est responsable de la configuration, du déploiement des différents composants ainsi que de la définition des modes de monitoring (actif ou passif). Il gère aussi l'échange des événements entre les différents composants depuis leur collecte (*EventsCollector*) jusqu'à la génération des symptômes (*SymptomsGenerator*).

Configurateur de la QoS

Le composant *QoSConfigurator* permet la définition du besoin local en. En fonction du besoin, ce composant interagit avec la base de connaissances afin d'extraire la liste des métriques permettant de superviser le degré de satisfaction de la QoS par l'entité gérée.

Une fois cette liste établie, le composant la communique au composant *EventsCollector* qui se charge de la collecte des différentes métriques. Il communique aussi le mode d'interaction en fonction des métriques à collecter par le composant.

Collecteur d'événements

Le composant *EventsCollector* est responsable de la collecte des métriques. Il exploite les capteurs logiques (via les interfaces JMX) intégrées dans l'entité OM2M gérée. Deux modes d'interaction avec l'entité gérée sont envisagés, actif et passif [low03] :

1. **Mode actif (pull) :** la supervision se fait en interrogeant la plateforme à des intervalles réguliers. Ce mode présente l'avantage de collecter les métriques indépendamment de l'état de la plateforme (satisfaction de la QoS ou pas) et de déduire ainsi le comportement de la plateforme vis-à-vis du trafic entrant. Ceci étant, la détection de toutes les infractions dépend de la périodicité choisie : une périodicité élevée peut donc

conduire à rater quelques-unes ; une périodicité courte peut amener à surcharger le système géré ;

2. **Mode passif (push) :** dans ce mode, ce sont les capteurs logiques qui signalent au composant de monitoring un changement d'état ou de valeur significative. Dans ce cas, la supervision est plus dynamique puisqu'elle capture toutes les infractions de SLA contrairement à l'approche précédente qui dépend de la périodicité. Ceci dit, avec ce mode, l'intelligence est déportée au niveau de l'entité gérée, sans pour autant être capable de gérer en même temps des besoins différents de plusieurs applications IoT critiques.

Dans notre cas, il est possible de combiner les deux modes. Le mode passif peut être utilisé par le composant de monitoring à l'arrivée de chaque requête au niveau de l'entité Middleware. Cette entité envoie donc toutes les métriques servant à la détection de la dégradation de la QoS. Alors que le mode actif peut être utilisé pour la collecte des métriques de performance informatique des niveaux sous-jacents relatives par exemple à l'état de la JVM, VM ou la machine physique, leur consommation de mémoire ou la charge de la CPU.

Processeur d'événements en CEP

Le composant *EventsProcessor* traite les événements de supervision afin de reconnaître et de détecter des *patterns* complexes corrélés à des symptômes. Ces patterns offrent une possibilité d'agrégation (*EventsAggregator*), filtrage (*EventsFilter*) ou corrélation (*EventsCorrelator*). L'ensemble des patterns sont stockés dans la base de connaissance.

Plusieurs techniques de traitement des événements et de définition de patterns existent dans la littérature. Nous retrouvons essentiellement l'approche de traitement d'événements complexes (CEP - *Complex Event Processing*) [luc02]. Le CEP est défini comme un ensemble d'outils et de techniques permettant d'analyser et de manipuler des données entrantes comme une suite complexe d'événements. Il assure un traitement en temps réel de ces événements selon des *patterns* définis. Ces *patterns* peuvent par exemple être une représentation d'une rafale d'événements ne respectant pas le seuil critique, ou une tendance lorsqu'un certain nombre d'événements est supérieur au seuil critique. Malgré les efforts de standardisation [jai08, ani11], une multitude de langage de description de patterns pour le CEP sont utilisés, notamment CQL (*Continuous Query Language*) [ara06] et EPL (*Event Processing Language*) [cug12] proposés par Oracle, qui sont similaires au langage de requêtes SQL.

L'IUT (*Union Internationale des Télécommunications*) [itu01] a donné des recommandations concernant le temps de réponse des données d'applications web ou transactionnelles (Tableau 4.2). Trois états sont définis : *préféré* lorsque le temps de réponse est inférieur à 2000 ms, *accepté* lorsqu'il est entre 2000 et 4000 ms, et enfin *non-accepté* lorsque le temps de réponse est supérieur à 4000 ms.

Intervalle (en ms)	Etat
[0, 2000)	Préférable
[2000, 4000)	Acceptable
[4000, +∞)	Non-acceptable

Tableau 4.2 - Recommandations de l'ITU-T G.1010 pour le temps de réponse d'applications web ou transactionnelles

Sur cette base, nous pouvons par exemple définir trois *patterns* (Tableau 4.3) basés sur les différents seuils de l'IUT. La description de ces *patterns* est réalisée via le langage EPL.

Pattern	Description
Pattern 1	select * from Event match_recognize (measures A as rtt1, B as rtt2, C as rtt3 pattern (A B C) define as A as A.value < 2000, B as B.value < 2000, C as C.value < 2000)
Pattern 2	select * from Event match_recognize (measures A as rtt1, B as rtt2, C as rtt3 pattern (A B C) define as A as A.value > 2000 and A.value < 4000, B as B.value > 2000 and B.value < 4000, C as C.value > 2000 and C.value < 4000)
Pattern 3	select * from Event match_recognize (measures A as rtt1, B as rtt2, C as rtt3 pattern (A B C) define as A as A.value > 4000, B as B.value > 4000, C as C.value > 4000)

Tableau 4.3 - Description des Patterns pour la génération des symptômes

Le Pattern 1 décrit le cas de trois événements successifs dont les temps de réponse sont dans l'intervalle préférable. Le Pattern 2 décrit trois événements successifs dont les temps de réponse sont dans l'intervalle acceptable. Le Pattern 3 décrit trois événements successifs pour des temps de réponse successifs dans l'intervalle non-acceptable.

Générateur de symptômes

Le composant *SymptomsGenerator* réalise la corrélation entre les patterns détectés et les symptômes à lever. Ce composant est invoqué par le composant *MonitoringManager* lorsqu'un ou plusieurs événements correspondent à un pattern. Il génère ainsi le symptôme adéquat via le catalogue des symptômes contenus dans la base de connaissance. Il envoie le symptôme généré au composant d'analyse.

Partant des recommandations de l'ITU illustrées dans le Tableau 4.2 précédant, les symptômes qui peuvent être levés sont représentés par le Tableau 4.4 suivant. Ces symptômes sont levés en fonction du pattern détecté.

Pattern	Symptôme
Pattern 1	[id=1, value=RTT_preferred, Priority=HIGH]
Pattern 2	[id=2, value=RTT_acceptable, Priority=HIGH]
Pattern 3	[id=3, value=RTT_unacceptable, Priority=HIGH]

Tableau 4.4 - Corrélation entre les patterns et les symptômes

L'implémentation des patterns et la vérification de leur correspondance peut se baser sur des outils de CEP orientés temps réel tels que ESPER [esper]. Ce framework open source et développé en JAVA est dédié à l'identification des événements significatifs dans un nuage d'événements. Au lieu de traiter des données historiques stockées dans une base de données classique, avec ESPER, le traitement peut se faire directement en temps réel sur les événements entrant.

L'architecture fonctionnelle présentée précédemment permet de considérer deux approches de monitoring : un monitoring *réactif* qui intervient lorsque l'infraction du SLA est déjà présente, et un monitoring *proactif* basée sur des techniques de prédiction pour prédire la dégradation de la QoS avant son arrivée. Dans ce qui suit, nous présentons ces deux approches de monitoring ainsi que les techniques attenantes pour leur mise en place.

4.3.3. Approches de Monitoring réactif et proactif

Monitoring réactif

Le monitoring réactif consiste à lever le symptôme lorsqu'il y a une non satisfaction du besoin en QoS de l'application par le Middleware. Le symptôme est généré en observant directement les métriques de performance (par exemple, le temps de réponse).

En mode réactif, notre approche de monitoring se base sur la technique du CEP pour la supervision de la dégradation du temps de réponse de la requête au regard du temps de réponse critique ciblé (RT_{TH}). Par exemple, un pattern décrivant trois événements successifs (ici trois temps de réponse) RT_1, RT_2 et RT_3 supérieurs au seuil critique (RT_{TH}), peut être représenté comme suit via le langage EPL :

*select * from Event match_recognize (measures A as RT_1, B as RT_2, C as RT_3 pattern (A B C) define as A as A.value > RT_{TH}, B as B.value > RT_{TH}, C as C.value > RT_{TH})*

Dans ce cas, les événements à collecter depuis le Middleware concernent les temps de réponse aux requêtes. Comme illustré sur la Figure 4.15, la collecte d'une telle métrique a un impact sur les performances du Middleware (ici, OM2M sur une Raspberry Pi avec 2 cœurs activés). Pour un scénario de 400 requêtes de récupération, l'impact atteint un coût de 11,7% de la CPU totale pour un taux d'arrivée de 16 reqs/s. La consommation de la RAM moyenne reste autour de **33 Mo** dans les deux cas, donc presque aucune surconsommation.

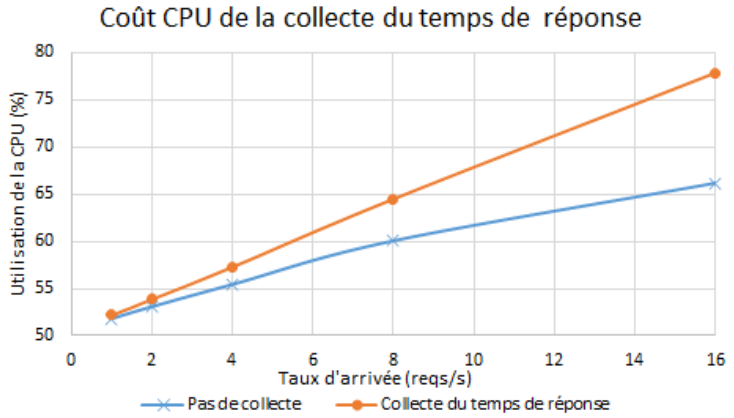

Figure 4.15 - Impact de la collecte du temps de réponse sur la CPU

Cet impact provient du fait que, malgré l'utilisation des interfaces JMX, la collecte du temps de réponse nécessite l'ajout d'estampilles à l'intérieur de plusieurs composants de la plateforme OM2M.

Afin de minimiser le coût du monitoring sur le système géré, il faut donc collecter les métriques nécessitant le moindre traitement par le système géré. Pour aboutir à cet objectif, notre approche se base sur la collecte des taux d'arrivée et l'utilisation du modèle analytique pour le calcul du temps de réponse (RT_{AN}). Pour des requêtes de récupération, le *pattern* du mode rafale va considérer cette fois-ci les taux d'arrivée comme les événements entrants. Il devient donc :

*select * from Event match_recognize (measures A as $RT_{AN}(\lambda_1)$, B as $RT_{AN}(\lambda_2)$, C as $RT_{AN}(\lambda_3)$ pattern (A B C) define as A as A.value > RT_{TH}, B as B.value > RT_{TH}, C as C.value > RT_{TH})*

$$Avec : RT_{AN}(\lambda) = \frac{1}{\mu_H - \lambda} + \frac{1-p}{n\mu_C(1-p) - \lambda} + \frac{1-p}{(1-p)\mu_D - p\lambda}$$

La Figure 4.16 représente l'impact de la collecte du temps de réponse et du taux d'arrivée sur l'utilisation de la CPU. Ces courbes représentent la consommation additionnelle liée à cette collecte. Comme illustré par cette figure, la collecte du temps de réponse engendre une consommation supérieure de la CPU liée au traitement supplémentaire que cela engendre au niveau de l'entité supervisée. En collectant directement les taux de service, la consommation de la CPU est plus basse. Elle est réduite de 3% pour le cas de 16 requêtes par seconde par exemple. Ce pourcentage devient significatif au fur et mesure que le taux d'arrivée augmente.

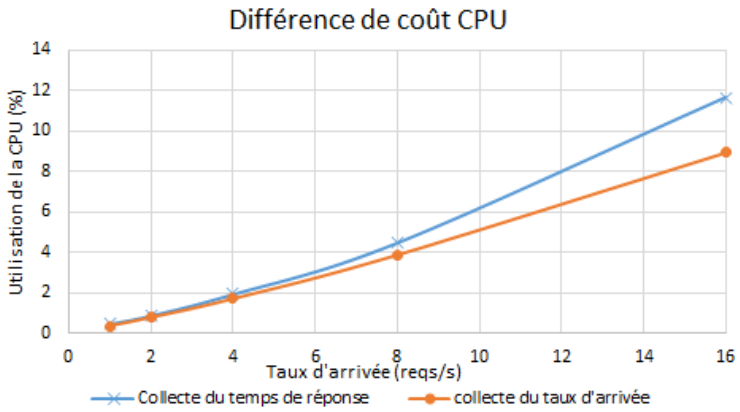

Figure 4.16 - Coût CPU de la collecte du temps de réponse et du taux d'arrivée

Afin de réduire l'impact du monitoring sur le Middleware géré, la collecte doit se faire uniquement sur le taux d'arrivée. Le temps de réponse est calculé analytiquement par le composant de monitoring en se basant sur le modèle analytique présenté précédemment.

Monitoring proactif

Par opposition au monitoring réactif, le monitoring prédictif consiste à anticiper la dégradation de la QoS en se basant sur une connaissance du contexte de déploiement. Sur ces bases, notre approche consiste à prédire du taux d'arrivée des requêtes au Middleware afin de calculer le temps de réponse du Middleware pour le prochain trafic entrant. Cette approche permet donc d'anticiper toute dégradation de la QoS et d'élaborer les plans nécessaires avant la dégradation.

Dans la littérature, il existe plusieurs techniques de prédiction. Dans notre cas, les données considérées sont issues d'un ensemble de capteurs et sont représentées sous forme d'une série temporelle [box94]. En étudiant la série, nous pouvons déterminer le modèle de prédiction le plus adapté. Pour des données ayant un caractère déterministe, les modèles déterministes tels que la régression ou le lissage exponentiel sont les plus adéquats.

Modèle Autorégressif et Moyenne Mobile (ARMA)

Dans notre cas, les données considérées ont un caractère aléatoire dans le temps. Le modèle de prédiction qui s'avère le plus adéquat est le modèle ARMA (*Auto-Regressive Moving Average*) [bro87] représenté par ARMA(p, q). Ce modèle combine deux modèles, le premier est celui autorégressif (AR) dont l'ordre est représenté par p. Le deuxième est le modèle à moyenne-mobile (MA) et dont l'ordre est représenté par q. Dans ce modèle ARMA, la valeur à l'instant *t* du processus est exprimée comme une combinaison linéaire des valeurs précédentes, des erreurs précédentes et d'un bruit blanc (4.31).

$$X_t = \varepsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \quad (4.31)$$

Où :

- φ_i : sont les paramètres du modèle AR. Le modèle ARMA(p,0) est un modèle AR(p)
- θ_i : sont les paramètres du modèle MA. Le modèle ARMA(0,q) est un modèle MA(q)
- ε_i : sont des termes d'erreur de bruit blanc

Dans le modèle ARMA, la série temporelle doit être stationnaire. Un processus temporel à valeurs réelles et en temps discret (X_1, X_2, …, X_t) est dit (faiblement) stationnaire si :

1. la *moyenne statistique* est constante :

$$E[X_j] = m, \forall\, j = 1, \ldots, t$$

2. l'*auto-covariance* ne dépend que du décalage :
$$Cov(X_j, X_{j-k}) = f(k), \forall\, j = 1, \ldots, t\ et\ \forall\ k\ = 1, \ldots, t$$

Si la série n'est pas stationnaire, une des techniques qui peut être appliquée repose sur la différence entre les valeurs ($Y_i = X_i - X_{i-1}$ pour tout i = 2, …, t) pour rendre les données stationnaires. En se basant sur la chronique (série temporelle) stationnaire, il s'agit donc de déterminer les ordres p et q ainsi que les paramètres φ_i et θ_j.

La méthodologie Box-Jenkins [box70] permet cette détermination suivant trois étapes :

1. **l'identification :** cette étape permet de reconnaître la classe d'appartenance du modèle en déterminant les ordres p et q. Ceux-ci sont obtenus à travers le calcul de la fonction d'autocorrélation (ACF) et la fonction d'autocorrélation partielle (PACF). Une chronique peut être la réalisation d'un **modèle ARMA**(p,q) si et seulement si son ACF tout comme son PACF décroissent rapidement vers zéro. L'ordre du processus est obtenu en prenant le temps *p* au-delà duquel les PACFs sont identiquement nuls et le temps *q* au-delà duquel les ACFs sont considérés comme tous nuls ;
2. **l'estimation :** cette étape consiste en le calcul des paramètres requis et leur validation à travers leur aptitude à modéliser la chronique. Plusieurs méthodes existent telles que la méthode du maximum de vraisemblance, Yule-Walker ou des moindres carrées ;
3. **le test :** une fois les paramètres vérifiés, nous devons procéder à la vérification globale de tout le modèle. Pour ce faire, nous faisons l'hypothèse que les erreurs résiduelles doivent suivre une distribution normale de tendance centrale nulle. Nous vérifions cette caractéristique en nous penchant sur l'ACF des résidus.

Suite à ces étapes, la prédiction peut être réalisée selon le modèle de prédiction obtenu dans l'environnement considéré.

Application de la méthodologie à notre contexte

Nous considérons des données issues d'un bâtiment (bâtiment ADREAM [adr12] du LAAS-CNRS) doté de capteurs et envoyées vers le Middleware OM2M. Ces données ont une tendance aléatoire dû au fait que les envois se font sur la base de la détection d'un événement. Nous suivons la démarche pour prédire le taux d'arrivée des requêtes par seconde sur un intervalle d'une minute. L'objectif est d'intégrer cette prédiction au modèle analytique à base de files d'attente afin de prédire le temps de réponse.

Les observations ont lieu à intervalle d'une minute et correspondent au nombre moyen de mesures par seconde effectuées par les capteurs. Chaque mesure étant remontée en temps réel par requête HTTP vers le Middleware OM2M, les 240 observations correspondent alors au taux d'arrivée des requêtes pendant 240 minutes, soit 4 heures de temps. Les données sont séparées en deux grandes catégories (Figure 4.17) : (1) les données d'apprentissage (210 observations) qui permettent de déterminer le modèle ARMA le mieux approprié ; et (2) les données de validation (30 observations) qui sont utilisées pour tester la précision du modèle obtenu. L'absence de tendance et la faible variabilité des données d'apprentissage autour de leur valeur moyenne mène à l'hypothèse de stationnarité des données. Cette hypothèse sera vérifiée par des tests formels suivant la fonction *auto.arima()* du logiciel R [riz12].

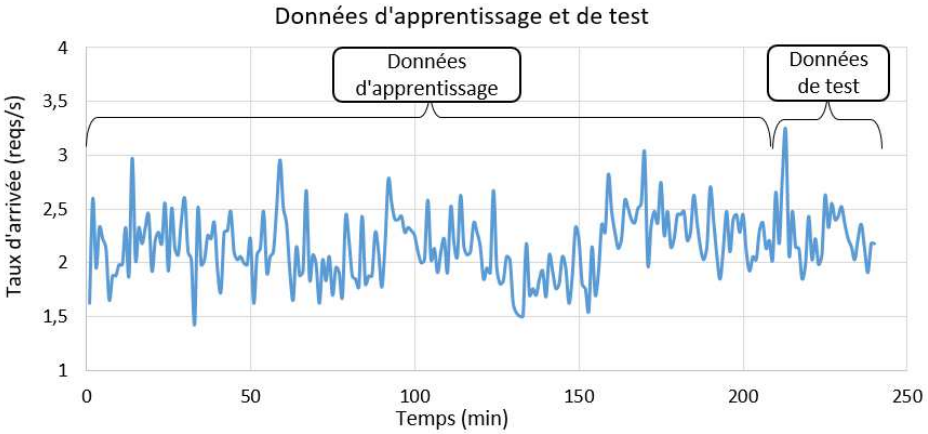

Figure 4.17 - Evolution du Taux d'arrivée pour la construction du modèle prédictif

Il est important de noter que plutôt que d'utiliser les fonctions ACF et PACF, les fonctions de R utilisent un certain nombre de critères d'information (AIC, BIC, etc.) pour identifier le modèle approprié et en estimer les paramètres. Le critère d'information le plus utilisé est le critère d'information d'Akaike [aka74] défini par $AIC = 2k - 2\ln(L)$, où k est le nombre de paramètres à estimer et L est le maximum de la fonction de vraisemblance du modèle. Les valeurs de p et q sont alors choisies en minimisant l'AIC après la *stationnarisation* des données. La métrique utilisée pour évaluer notre modèle est la moyenne des pourcentages d'erreur absolue (MAPE - *Mean Absolute Percentage Error*) calculée depuis les données d'apprentissage et celles issues du modèle. En suivant cette démarche, le modèle obtenu est un modèle ARMA(21,3).

La représentation des 21 prochaines données de prédiction comparativement aux données réelles est illustrée par la Figure 4.18. L'utilisation de ce modèle signifie que la prédiction est basée sur les 21 dernières valeurs pour estimer la prochaine valeur, ainsi que sur l'erreur de prédiction des trois dernières valeurs.

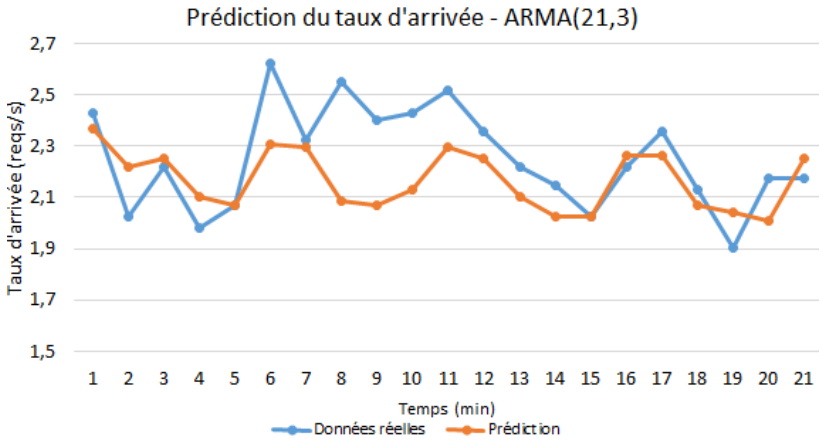

Figure 4.18 - Prédiction des 21 prochaines valeurs avec ARMA(21,3)

Comparativement à d'autres modèles de prédiction (Tableau 4.5), le modèle ARMA(21,3) donne la meilleure prédiction avec une MAPE de prédiction de 6,1%. Malgré le fait que l'écart soit petit entre les valeurs de taux d'arrivée, la prédiction permet de fournir la tendance des taux d'arrivée dans le contexte considéré. Cette information permet au système de gestion autonomique d'anticiper certaines actions de reconfiguration pour adapter l'entité Middleware face à l'arrivée du nouveau trafic.

Modèle de prédiction	MAPE (%)
AR(4)	7,38
AR(5)	7,23
ARMA(1,1)	7,28
ARMA(21,3)	6,10

Tableau 4.5 - Comparaison des modèles de prédiction

Dans une approche de monitoring proactif, la valeur du taux d'arrivée prédite (λ_{PRED}) issue du modèle ARMA est combinée avec le modèle analytique afin de prédire le temps de réponse. L'équation suivante représente un exemple de pattern considérant un seul événement supérieur au seuil critique (RT_{TH}).

*select * from Event match_recognize (measures A as* RT_{PRED} *pattern (A) define as A as A.value >* RT_{TH}*)*

$$Avec : RT_{PRED} = \frac{1}{\mu_H - \lambda_{PRED}} + \frac{1-p}{n\mu_C(1-p) - \lambda_{PRED}} + \frac{1-p}{(1-p)\mu_D - p\lambda_{PRED}}$$

Une fois que le temps de réponse prédit est supérieur au temps de réponse seuil (ce qui signifie que le pattern est identifié), un symptôme est généré à destination du composant d'Analyse.

Cette combinaison des deux modèles (analytique à base des files d'attente et ARMA) permet donc de réaliser un monitoring proactif au moindre coût sur l'entité Middleware gérée. Cette phase permet d'anticiper les dégradations de la QoS et de déclencher les phases d'analyse et de planification des composants appropriés suivant la boucle MAPE-K.

4.4. CONCLUSION

Suivant notre approche de gestion adaptative de la QoS, deux propositions essentielles ont été présentées dans ce chapitre. La première concerne la proposition d'un modèle analytique représentant le Middleware OM2M en se basant sur la théorie des files d'attente. Ce modèle a pour objectif de guider les différents composants de l'AMS pour la gestion locale de la QoS en fournissant une représentation mathématique des métriques de performance telles que le temps de réponse ou la taille des files. L'élaboration de ce modèle s'est basée sur une étude qualitative et quantitative des composants internes de la plateforme OM2M afin de détecter les potentielles stations du réseau de file d'attente. Trois stations ont été considérées. La première est celle du serveur accueillant les requêtes entrantes. La deuxième station est celle des ressources (threads et processeurs) traitant les requêtes génériques. Enfin, la station représentant la base de données pour la persistance des données. Les métriques de performance sont représentées par des paramètres d'entrée (taux d'arrivée, nombre de cœurs) et des paramètres internes (taux de service de chaque station et probabilité de retour).

Une approche basée émulation a été proposée afin d'approximer les paramètres internes du modèle. Cette approche a été appliquée au cas d'une plateforme OM2M hébergée dans une gateway, spécifiquement pour les ressources de type <*contentInstance*> et via des requêtes de type GET et POST. Les paramètres obtenus ont permis d'avoir des taux d'erreur assez faibles permettant ainsi de valider les approximations obtenues.

La deuxième contribution majeure de ce chapitre est une application de ce modèle au composant de Monitoring. Nous avons tout d'abord présenté les interfaces JMX permettant la supervision d'une entité Middleware basée sur le langage Java telle que OM2M. Nous avons ensuite donné l'architecture fonctionnelle du composant de monitoring ainsi que les

composants internes permettant le traitement des événements de supervision. La technique de CEP a été utilisée pour la description des patterns. Lorsque les événements correspondent au pattern, le symptôme est généré et est envoyé au composant d'Analyse.

Deux approches de monitoring ont été proposées dans cette partie.

La première approche, *réactive,* permet de ne lever les symptômes que si la dégradation de la QoS locale est constatée. Appliquée dans notre contexte pour la supervision du temps de réponse, cette approche part tout d'abord de la collecte du temps de réponse directement depuis la plateforme gérée. La mesure des performances de cette approche sur l'entité gérée nous laisse constater que la collecte directe du temps de réponse impacte les performances de la machine, notamment en terme de CPU. Une extension de cette approche a donc été proposée afin de réduire cet impact. Elle tire parti du modèle analytique et la collecte se fait donc sur le taux d'arrivée. Le temps de réponse est calculé analytiquement par la suite à travers le modèle analytique.

La deuxième approche de monitoring, *proactive*, vise à prédire la dégradation de la QoS en se basant sur les événements passés. Dans le contexte de l'IoT, cette approche proactive peut être dédiée aux applications très critiques. Nous avons élaboré donc un modèle de prédiction basé sur le processus ARMA permettant de prédire le taux d'arrivée. Il a par la suite été couplé au modèle analytique de l'entité Middleware. Cette combinaison permet de construire des patterns sur la base de taux d'arrivée prédits et du temps de réponse analytique. La série temporelle considérée nous a permis d'aboutir à un taux de précision d'à peu près 94%, s'approchant ainsi de celui recommandé par [bro87] et fixé à 95%.

Le Tableau 4.6 (où FA désigne le modèle analytique à base des files d'attente et ARMA le modèle de prédiction) synthétise les différentes approches d'élaboration des patterns proposées dans ce chapitre. Elles sont relatives à la dégradation du temps de réponse suivant les deux approches réactive et proactive.

Approche de Monitoring	**Approche réactive**		**Approche proactive**
Entrées	RT_{MESURE}	λ_{MESURE}	λ_{MESURE}
Techniques	CEP	CEP + FA	CEP + ARMA + FA
Calcul intermédiaire	-	RT_{FA}	λ_{ARMA} (prédit) et RT_{FA} (λ_{ARMA})

Tableau 4.6 - Approches d'élaboration de patterns pour la génération de symptômes de dégradation du temps de réponse

Le chapitre suivant poursuit la même approche. Il couple le modèle analytique avec un modèle basé sur le formalisme des graphes et des règles de réécriture de graphes. Ces règles sont appliquées au composant de planification de la gestion locale faite par l'AMS.

Chapitre 5

Planification guidée par les modèles pour la gestion locale de la QoS

Contenu

5.1. INTRODUCTION

Le chapitre 3 a porté sur la partie structurelle de l'architecture du système IoT-Q. Dans ce chapitre, nous approfondissons la partie comportementale de cette architecture au travers du composant de planification du gestionnaire autonomique local (AMS) à chaque entité.

En général, les approches envisagées pour la planification vont de l'utilisation de règles simples (de type *"SI <X> est correct, ALORS faire <Y>"*) jusqu'à des règles plus complexes basées

sur des modèles descriptifs ou de décision [hol89]. Dans une approche orientée modèle, l'architecture du système géré, de ses entités constituantes, leurs relations ainsi que les transformations possibles doivent être modélisées.

Trois types de modèles peuvent être utilisés dans cette approche. Les modèles prédictifs analysent les événements passés afin d'estimer le futur de manière probabiliste. Alors que les modèles descriptifs permettent de quantifier les relations entre les événements et de les classer en groupes. A l'opposé des modèles prédictifs, les modèles descriptifs identifient plusieurs relations entre les événements. Enfin, les modèles de décision ou de prise de décision qui élaborent les relations entre tous les éléments d'une décision (les événements connus, la décision et les résultats attendus de la décision) afin de prédire les résultats de ces décisions.

Les architectures logicielles peuvent être décrites via des modèles semi formels tel que UML 2.0. Cependant, dans un contexte tel que l'IoT susceptible d'engendrer des transformations structurelles de façon dynamique, c'est-à-dire durant le temps d'exécution du système, ce type de description gagne à être remplacé par des modèles plus adéquats [eic15]. Pour modéliser l'évolution dynamique d'une architecture, il existe des modèles, représentations et approches formelles de description basés notamment sur la théorie des graphes et sur des règles de réécriture associées [eic15]. Les graphes permettent alors de représenter les différentes configurations du système géré, les règles de réécriture formalisant leurs transformations.

La contribution générale décrite dans ce chapitre réside dans l'utilisation de techniques basées sur les graphes pour guider le comportement du système IoT-Q dans sa phase de planification. Ces techniques sont appliquées pour la planification des mécanismes orientés ressources (décrits dans le chapitre 2), dans le cadre d'un cas d'étude correspondant à une activité de supervision d'une zone à risques (en l'occurrence un métro doté de multiples capteurs et actionneurs) et d'intervention en cas de problème (incident ou danger).

Dans cette optique, ce chapitre présente deux contributions principales.

Nous proposons tout d'abord une modélisation à base de graphe du cas d'étude considéré. Nous distinguons deux niveaux d'abstraction. Le premier niveau, dit *métier*, modélise les rôles et les interactions d'entités de haut niveau impliquées dans chacune des phases du cas d'étude. Le second niveau, dit *opératoire*, fait apparaître les applications logicielles impliquées ainsi que les entités Middleware nécessaires à leurs interactions avec les objets connectés. Pour chacun des niveaux, nous donnons la description du style architectural sous forme de productions de grammaires afin de modéliser les entités et les interactions.

La deuxième contribution s'inscrit dans la phase de planification des actions de gestion de la QoS dans les deux phases du cas d'étude. Nous proposons ainsi des règles de planification suivant une approche *multi-modèles*, couplant des techniques d'appariement et de transformation de graphes avec le modèle analytique de la plateforme Middleware décrit au chapitre 3. L'approche proposée permet tout d'abord de valider l'applicabilité des mécanismes orientés ressources envisagés. Elle permet ensuite de modéliser les modifications d'attributs ou de structure du graphe correspondant aux actions d'adaptation réalisées. À des fins d'illustration, un exemple simple de protocole de planification de la mise en œuvre de ces règles est également fourni.

Dans la suite de ce chapitre, la section 5.2 détaille notre approche d'adaptation dynamique guidée par les modèles. Les différents niveaux d'abstraction ainsi que la gestion multi-modèles proposée sont présentés. La section 5.3 présente le cas d'étude de gestion d'une foule dans un réseau de lignes de métro ; un scénario en 2 phases (*supervision* et *intervention*) est exposé.

Les niveaux d'abstraction considérés (*métier* et *opératoire*) sont finalement détaillés. Les sections 5.4 et 5.5 fournissent l'ensemble des propriétés architecturales ainsi que la description des styles architecturaux pour les niveaux métier et opératoire. Enfin, la section 5.6 présente les règles de reconfiguration structurelle, pour la gestion dynamique de l'architecture, issues de la composition du modèle analytique et des règles de réécriture de graphes. Nous terminons le chapitre par une conclusion.

5.2. ADAPTATION DYNAMIQUE GUIDÉE PAR LES MODÈLES

Le contexte IoT est dynamique à la fois par les équipements, les ressources, les profils et les besoins en QoS des applications, qui sont susceptibles d'évoluer durant l'exécution du système dans son ensemble. Dans notre cas de gestion de la QoS exprimée par les applications IoT, les adaptations doivent se faire donc par le composant de planification de manière dynamique (c'est-à-dire sans interruption des services fournis aux applications). Pour ce faire, il existe des approches formelles issues de la théorie des graphes et de la transformation de graphes. Plusieurs approches pour la description des transformations de graphes ont été proposées, telles que par exemple DPO, SPO et edNCE [eng89].

Notre approche d'adaptation, guidée par des modèles multiples, couple ces techniques de description avec le modèle analytique proposé dans le chapitre 3. Le but est double : il est d'une part de déterminer le paramétrage des mécanismes (par exemple, le nombre de cœurs à activer) a priori retenus par le composant de planification, et de valider l'applicabilité de ces mécanismes ; il est d'autre part de maintenir à tout moment un modèle du système résultant des actions d'adaptation réalisé, cohérent vis-à-vis du système réel.

Dans cette section, nous définissons tout d'abord ce que sont les architectures dynamiques. Nous présentons ensuite les approches de base pour la description de ces architectures. Enfin, nous détaillons notre approche d'adaptation guidée par les modèles en la positionnant vis-à-vis d'autres approches d'adaptation basées graphes.

5.2.1. Architecture dynamique et graphes de description

Les architectures peuvent être divisées en deux types selon leur capacité ou non à évoluer lors de l'exécution du système [mor07]. Nous distinguons ainsi les architectures *statiques* et *dynamiques*. Une architecture statique est une architecture dont la topologie (entités et liens d'interaction) n'évolue pas. Au contraire, la description d'une architecture dynamique ne peut, par essence, se limiter à une unique topologie. Elle se doit de caractériser l'ensemble des topologies possibles du système, c'est-à-dire l'ensemble des configurations que le système peut adopter. Cet ensemble est spécifié par un *style architectural* caractérisant les contraintes et les caractéristiques communes des configurations acceptables. Dans le contexte des architectures dynamiques, il est également nécessaire de caractériser les règles guidant le passage d'une configuration à une autre, dans le but d'assurer la validité des *transformations*. La description d'architectures dynamiques comporte donc trois grandes notions : *configurations*, *style architectural* et *transformations*.

Par essence, l'architecture du système que nous considérons est dynamique. Deux types d'adaptation sont considérés : (1) une adaptation dite *corrective* guidée par des défaillances du système ou des contraintes de QoS non respectées durant une phase d'activité ; et (2) une

adaptation dite *évolutive* suite à l'évolution de l'activité d'une phase vers une autre, induisant des changements dans les interactions et/ou dans les exigences de QoS.

La description des architectures dynamiques diffère de celle des architectures statiques. Ces dernières peuvent adopter, par définition, une unique topologie. Une description de celle-ci précisant ses différents composants et leurs interconnexions est donc suffisante pour décrire le système. Pour des architectures dynamiques, dont les composants et connexions peuvent être ajoutés, modifiés ou supprimés durant l'exécution du système, cette approche peut être transposée en énumérant l'ensemble des topologies acceptables du système. Ceci peut néanmoins s'avérer très largement inefficace, produire une telle énumération pouvant être une tâche ardue, voire totalement inapproprié s'il existe une infinité de topologies acceptables.

Les graphes constituent un puissant modèle de description des architectures dynamiques **[**luk13**]**. De manière basique, un graphe G peut être défini par un ensemble de nœuds N et un ensemble d'arêtes A, une arête reliant 2 nœuds (arrête de cardinalité 2). Par exemple, dans le cas d'architectures orientées composants, les nœuds correspondent aux composants logiciels et les arcs à leurs relations qui peuvent être des canaux de communication [gue06]. Des représentations plus riches sont basées sur des graphes orientés, des nœuds incluant des labels et des arêtes dotées d'étiquettes.

Pour modéliser une architecture dynamique à partir des graphes, nous utilisons ici trois méta-modèles :

- des *graphes* pour représenter les configurations du système ;
- des *règles de réécriture de graphe* [eng97, ehr90] pour la spécification de transformations. Une règle de réécriture décrit à la fois les conditions d'application d'une transformation et ses effets. Elle précise un ensemble d'ajout/suppression d'arcs et de nœuds au sein d'un sous-graphe, caractérisant ainsi le passage de graphes modélisant le système dans des états donnés à de nouveaux graphes et états ;
- des *grammaires de graphes* pour la spécification générative de styles architecturaux. Une grammaire de graphes est constituée entre autres d'un ensemble de règles de réécriture appelées *productions de la grammaire*. L'application d'une séquence quelconque de ces productions permet d'aboutir à une instance de déploiement de l'architecture compatible avec le style caractérisé par la grammaire.

Plusieurs approches de réécriture peuvent être utilisées pour la spécification des productions de grammaires et des règles de transformation. Les règles de réécriture les plus simples sont basées sur une paire (L; R) où L et R désignent des graphes appelés respectivement graphes *mère* et *fille*. Ce type de règle est applicable sur un graphe hôte G s'il existe une occurrence du graphe mère L dans G. Son application a pour conséquence la suppression de cette occurrence[2] de L du graphe G et son remplacement par une copie (isomorphe) du graphe fille R. Le graphe issu de ces opérations est appelé *graphe résultant* G'. L'application de cette approche basique peut faire apparaître des arcs dits *suspendus* qui ne relient pas deux nœuds. Pour pallier à ce problème, deux approches principales existent **[**ekl90**]**, l'approche SPO (*Single PushOut*) et l'approche DPO (*Double PushOut*).

L'apport de l'approche SPO est que des parties (nœuds et arcs) du graphe L seront préservées après l'application de la règle. Ces parties sont spécifiées en les faisant apparaître dans les deux graphes L et R. L'application de la règle implique la suppression du graphe correspondant à Del = (L\(L∩R)) et le rajout du graphe correspondant à Add = (R\(L∩R)). Selon l'approche

[2] selon l'algorithme, la gestion peut être par exemple une des occurrences, la première ou toutes les occurrences.

SPO, les arcs suspendus sont supprimés. Dans l'exemple de la partie 1 de la Figure 5.1, les nœuds 2, 5 et 4 (L) sont supprimés et les nœuds 2, 5, 6 et 7 (R) sont ensuite introduits, avec préservation de 2 et 5 et des arcs les reliant (L∩R). Cette opération fait apparaître un arc suspendu attaché au nœud 3 qui est ensuite supprimé.

Alors que l'approche DPO consiste en la spécification explicite de la partie K à préserver suite à l'application de la règle. La règle est donc sous la forme (L, K, R). La différence avec l'approche SPO est l'existence d'une condition appelée condition de suspension qui stipule que la règle ne peut être appliquée que si son application ne va pas entraîner l'apparition d'arcs suspendus. Si les deux conditions d'existence de l'occurrence et d'absence d'arcs suspendus sont réunies, l'application de la règle implique la suppression du graphe correspondant à l'occurrence de Del = (L\K) et le rajout d'une copie du graphe Add = (R\K). Dans l'exemple de la partie 2 de la Figure 5.1, la suppression de L\R entraîne la suppression du nœud 4 et les arcs le reliant à 2 et 5. Alors que l'ajout de R\K entraîne l'apparition des nœuds 6 et 7 et les arcs les reliant entre eux et avec les nœuds 2 et 5.

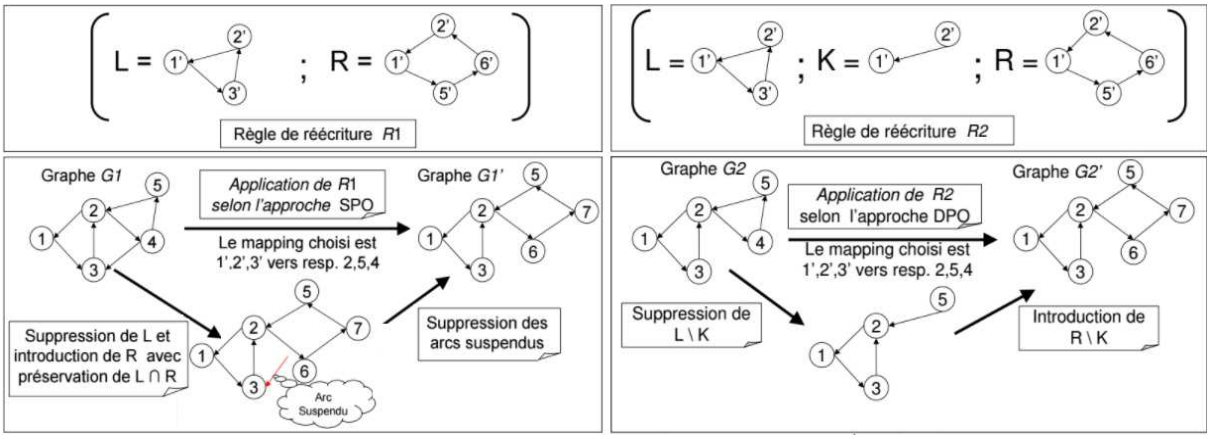

Figure 5.1 - Application de règles de réécriture suivant les Approches SPO et DPO [gue06]

Dans notre cas, nous utilisons aussi un champ E pour représenter les étiquettes des arcs. Il existe d'autres approches pour la description d'architectures plus complexes. Quelques-unes sont utilisées dans ce chapitre mais ne sont pas décrites ici, notamment l'approche edNCE [bra88, van89]. Elles permettent par exemple d'établir des connexions entre les nouveaux nœuds et les nœuds existants ainsi que leur orientation.

5.2.2. Positionnement de notre approche d'adaptation guidée par les modèles

Plusieurs approches de modélisation d'architecture dynamique basées sur les graphes et les règles de réécriture ont été proposées afin de maintenir un style architectural suite à l'application d'une/de reconfiguration(s).

[gue06] se focalise sur les applications à base de composants et orientées service. Il traite la problématique de description, de gestion, de vérification et d'implémentation de l'auto-reconfiguration dans ce type de systèmes en proposant une solution d'adaptation structurelle automatique en cours d'exécution. Il considère une adaptation *multi-niveaux* dont la description du style architectural et de son évolution (*horizontale* en changeant de phase ou *verticale* en changeant de niveau) est réalisée via des productions de grammaires élémentaires. Le travail considère aussi les besoins en QoS au niveau réseau ; cependant, les règles de transformation ne considèrent pas l'évolutivité de ces besoins et ne se focalisent que sur le cas de défaillances de composants et leur redéploiement sur d'autres machines.

[bou11] se base sur une approche de modélisation architecturale *multi-niveaux* et *multi-modèles*. Elle combine plusieurs modèles de représentation. Une partie est basée sur des règles SWRL (*Semantic Web Rule Language*) pour la description du contexte et des règles d'adaptation, et une partie basée sur les grammaires de graphes pour la transformation des configurations de déploiement. Ceci étant, ces règles sont appliquées en temps réel sur l'ensemble des nœuds du système, rendant ainsi l'implémentation de l'approche (appelée *moteur de transformation de graphes*) peu performante si le nombre de nœuds est important.

D'autres travaux similaires de reconfigurations dynamiques, se focalisant sur un contexte de gestion autonomique de systèmes M2M, étendent directement les approches précédentes [eic15] ou les combinent avec des modèles basés sur des automates temporisés pour garantir le respect des contraintes temporelles des entités M2M [gha16].

Dans nos travaux, nous nous positionnons aussi dans une approche de couplage de modèles à base de graphes et de réécriture avec d'autres modèles. Comme détaillé dans le chapitre 2, le premier type d'adaptation, dite *comportementale*, porte sur un ensemble de mécanismes portant sur le trafic sans changer la configuration du Middleware. Le deuxième type d'adaptation, dite *structurelle*, consiste en des mécanismes qui portent sur la configuration des ressources du Middleware afin de satisfaire les besoins en QoS. Notre approche de planification se focalise sur les actions d'adaptation structurelles. Elle propose un ensemble de règles pour guider le composant de planification. Ces règles sont guidées par les modèles et couplent le modèle analytique présenté dans le chapitre 3, qui donne une représentation analytique des métriques de performance du Middleware, avec les règles de production pour la reconfiguration structurelle du Middleware tout en assurant la cohérence architecturale. Dans ce cadre, le modèle analytique est utilisé pour décrire à la fois les conditions d'applicabilité de la règle mais aussi les paramètres de la nouvelle configuration.

Pour l'application de cette approche, que nous abordons par le biais d'un cas d'étude illustré en section 5.3, nous considérons également deux niveaux de modélisation que nous détaillons dans la même section :

1. le premier niveau, ci-après dit *métier*, permet de représenter les entités de haut niveau par des nœuds du graphe ainsi que les interactions entre elles par les arêtes du graphe. Ce niveau permet aussi de mettre en évidence les besoins en QoS de chaque trafic par le biais des étiquettes des arcs ;
2. le deuxième niveau, dit *opératoire*, est un raffinement du niveau métier faisant apparaître : a) via les nœuds du graphe : les entités logicielles (c'est-à-dire les applications IoT) et les entités Middleware (gateway et serveur) ; b) via les arcs reliant ces nœuds : les interactions entre ces entités ; et c) via les attributs des nœuds : les ressources informatiques allouées à ces entités.

5.3. ACTIVITÉ DE GESTION DE LA FOULE

Pour illustrer notre approche de gestion basée modèles, nous considérons un cas d'étude de gestion de la foule dans un réseau de métro. Cette section présente tout d'abord les entités physiques impliquées dans ce cas d'étude (section 5.3.1). Nous décrivons ensuite en section 5.3.2 les deux grandes phases considérées dans le cas d'étude (supervision de la zone et intervention en cas d'incident). Enfin, la section 5.3.3 fournit une modélisation de ce cas d'étude suivant deux vues que nous appelons ci-après niveaux d'abstraction *métier* et *opératoire*. Le premier niveau modélise l'activité métier par le biais des nœuds et des liens illustratifs des entités physiques et de leurs interactions. Le deuxième niveau (niveau

opératoire) modélise les entités logicielles (applications et entités Middleware) et leurs interactions qui permettent d'assurer la mise en œuvre du niveau métier.

5.3.1. Description du cas d'étude

Le cas d'étude considéré relève d'une activité de gestion de la foule dans un réseau de transport urbain, en l'occurrence un réseau de métro. Deux phases sont envisagées : une phase de supervision de la zone et une phase d'intervention pour l'évacuation de la foule en cas d'urgence. Les deux phases sont opérées à distance en exploitant les capteurs et actionneurs déployés sur site.

Plusieurs entités physiques sont impliquées (Figure 5.2) :

- **Poste général de contrôle :** assure la supervision globale du réseau de métro et l'intervention en cas d'incident ou danger. Il est doté des applications permettant la mise en place de ces opérations ;
- **Ligne de métro (L) :** constituée d'un ensemble de stations séparées par des interstations, une ligne est caractérisée par sa longueur, le nombre de stations et d'interstations qu'elle contient, ainsi que le nombre de rames qui y circulent ;
- **Station de métro (S) :** comporte un quai (Q) par direction. Elle dispose d'un ou plusieurs moyens d'accès de type escaliers, escalator (électrique) ou ascenseur (électrique). Les portes d'accès peuvent être par exemple des portillons d'accès qui sont soit coulissantes soit en tourniquet. Dans certain cas, les quais disposent de portes palières pour l'accès à la rame. Toutes ces portes et moyens d'accès sont contrôlables à distance via des actionneurs ;
- **Interstation (IS) :** sépare deux stations successives. Elle dispose de deux côtés d'accès en fonction de la direction de la rame et d'un ensemble d'issues de secours accessibles depuis les quais afin d'évacuer les passagers en cas de danger ou d'incident. L'interstation est reliée à chaque station par une porte ;
- **Rame de métro (RM) :** sert au transport des passagers et procède à des arrêts dans chaque station desservie. Elle est caractérisée par un ensemble de wagons. Chaque wagon dispose de part et d'autre d'un ensemble de portes. Lors de l'arrêt dans une station, ces portes s'ouvrent pendant une durée déterminée pour la sortie et la montée des passagers.

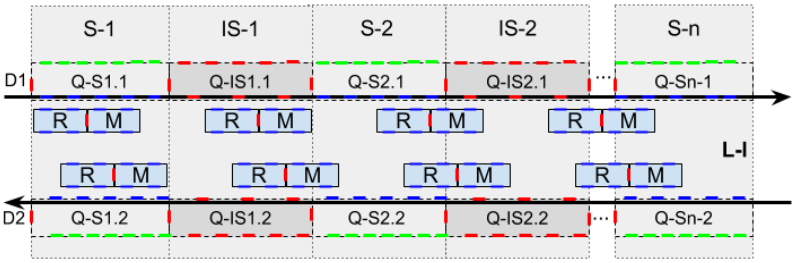

Figure 5.2 - Représentation de haut niveau d'une ligne de métro

Dans ce qui suit, nous présentons les deux phases du cas d'étude ainsi que les niveaux d'abstraction. Ces niveaux d'abstraction modélisent les éléments physiques d'un réseau de métro vis-à-vis du cas d'étude de gestion de la foule en cas de problème.

5.3.2. Phases d'exécution

L'activité de gestion de la foule dans un métro comporte deux phases principales.

La première est une phase de *supervision*. Elle exploite l'ensemble des capteurs déployés sur les différents niveaux (quais, rames, etc.) afin de collecter les événements de supervision. Ces données peuvent par exemple relever de la température d'un quai ou d'une rame, de l'état des portes et des issues, du nombre de passagers dans les wagons ou à l'intérieur de la station. L'ensemble de ces données permettent de construire une connaissance de l'état du réseau supervisé. Elles sont analysées afin de détecter d'éventuels dangers ou incidents.

La deuxième phase est une phase d'*intervention*. Elle est opérée suite à la détection d'un problème (danger ou incident). Dans cette phase, un ou des plans d'interventions sont mis en œuvre afin de répondre au problème. Leur réalisation est assurée par le biais d'actionneurs permettant par exemple de contrôler à distance les portes, les issues de secours, les rames de métro ou les moyens d'accès. Elle peut aussi faire appel à d'autres services pour solliciter leur intervention (policiers, pompiers, etc.). Cette phase se termine lorsque le problème est réglé et le système repasse en phase de supervision.

5.3.3. Niveaux d'abstraction

Précédemment, nous avons présenté les entités physiques de haut niveau dans les lignes de métro ainsi que les deux phases pour la gestion de la foule en cas de problème. Nous modélisons à présent les entités physiques et leurs interactions en entités métier et logicielles. Tel qu'introduit, nous considérons deux niveaux d'abstraction : *métier* et *opératoire* :

- **niveau métier :** donne une modélisation générique des entités physiques en des entités et des interactions métier. Les entités de ce niveau expriment les rôles joués par les entités physiques dans le scénario de gestion de la foule et pour les différentes phases ;
- **niveau opératoire :** décline les entités métier en applications logicielles et entités Middleware permettant de réaliser les interactions définies au niveau métier.

Notons que nous ne considérons pas dans cette thèse le modèle des entités réseau impliquées assurant au final la communication.

Niveau métier

Le niveau métier fournit un modèle des entités physiques (et de leur interactions) sous l'angle du rôle qu'elles jouent dans un scénario donné du cas d'étude (cf. ci-après). Nous parlerons dans la suite d'*entité métier* pour désigner une entité physique et un rôle en faisant l'hypothèse qu'une entité physique ne joue qu'un seul rôle.

A toute entité métier sont associés les attributs suivants :

- *Id :* identifiant de l'entité métier,
- *Type :* type d'entité métier,
- *Phase :* phase en cours de l'entité métier,
- L_{Id} *:* localisation, représentant la zone de couverture de l'entité métier.

Nous définissons trois types de rôles susceptibles d'être joués par les entités physiques considérées :

- **Centre de Contrôle Général (CCG) :** assure l'intervention en cas d'incident ou danger. Il possède une vision globale du réseau de métro lui permettant d'élaborer des plans d'intervention globaux sur l'ensemble de la ligne. Il doit donc gérer au moins un

CCLI. Il est représenté par un nœud *N(Id, CCG, P, L_{Id})* où *P* prend les valeurs de *sup* ou *intG* en fonction de la phase en cours. Ce rôle est rempli par le poste général de contrôle ;

- **Centre de Contrôle Local Intermédiaire (CCLI) :** Cette entité analyse les données collectées par un ou plusieurs centres de contrôle local finaux (CCLF) afin de détecter un danger ou un incident. Chaque CCLI doit donc gérer au moins un CCLF. En cas de détection d'une situation problématique, il assure une fonction d'intervention locale. Un CCLI est représenté par un nœud *N(Id, CCLI, P, L_{Id})* où *P* prend les valeurs de *sup*, *intI* ou *intG* en fonction de la phase en cours. Ce rôle est rempli par une station, une interstation ou une rame de métro ;
- **Centre de Contrôle Local Final (CCLF) :** Cette entité sert de relais entre les équipements et le CCLI. Elle collecte les données depuis les équipements et transfère également les commandes vers eux. Un CCLF est représenté par un nœud comportant 3 attributs : *N(Id, CCLF, L_{Id})* ; son rôle ne dépend pas de la phase en cours. Le rôle de CCLF peut être assuré par un quai d'une station, par un côté donné d'une interstation ou par un wagon de la rame.

Le trafic échangé entre les différentes entités représente des données issues des capteurs, des rapports ou des commandes vers les équipements. Ce trafic est représenté par un arc ayant comme étiquette <*type, taux d'arrivée, priorité, temps de réponse seuil*>. Notons que dans le cadre de notre approche de gestion, le taux d'arrivée est déduit par le composant de monitoring.

Deux types de trafic sont considérés dans cette phase :

- **Observations (O) :** représente les données d'observation émanant des équipements dont l'acheminement est assuré par le CCLF ;
- **Rapports (R) :** envoyés du CCLI au CCG concernant l'état de la zone qu'il gère. Un rapport comporte une synthèse des données d'observations et l'état du CCLI (supervision ou intervention en mentionnant le type du danger ou l'incident).

Les différentes entités et interactions métier doivent respecter un certain nombre de propriétés. Le CCG doit être unique dans le système. Les CCLI doivent posséder au moins un CCLF et être attachés au CCG. Enfin, le CCLF doit être attaché à un unique CCLI. Le système doit donc vérifier l'ensemble de ces propriétés. Dans la section 5.4.1, nous décrivons ces propriétés de manière formelle en créant des règles qui assurent leur respect. La Figure 5.3 fournit un exemple de déploiement au niveau métier dans la phase de supervision. Durant cette phase, le CCG et les CCLI des différentes zones sont en phase de supervision (sup). Les CCLF envoient des données d'observations collectées via les équipements au CCLI. Ce dernier analyse ces données et envoie des rapporteurs de synthèse au CCG.

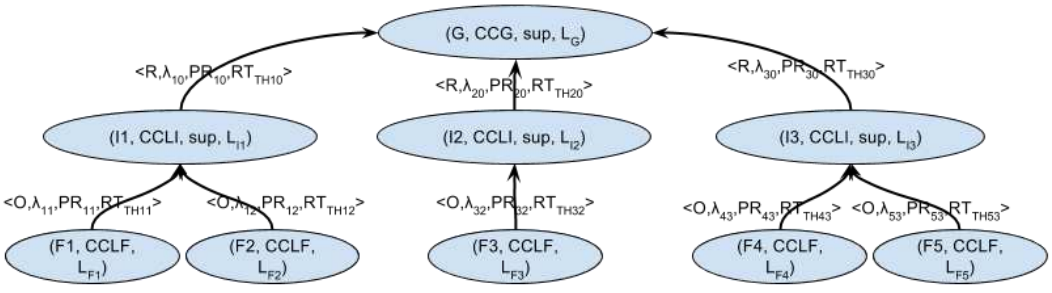

Figure 5.3 - Exemple de déploiement au niveau métier en phase de supervision

Dans la phase d'intervention, deux types de trafic peuvent être échangés en sus des données décrites précédemment :

- **Commande Locale (LC) :** commande élaborée par un CCLI en phase d'intervention intermédiaire (*intI*). Le passage à cette phase se fait lors de la présence d'un incident ou danger dans une zone donnée (L_{F1} de la Figure 5.4). Les commandes LC sont donc envoyées par le CCLI de gestion (zone L_{I1}) à destination des équipements attachés aux CCLFs des zones subordonnées (L_{F1} et L_{F2} dans la Figure 5.4) ;
- **Commande Générale (GC) :** commande émanant du CCG à l'issue de l'application d'un plan de gestion global. Ces commandes ont pour objectif la gestion des équipements des CCLF des autres zones (L_{F4} dans la Figure 5.4). Le CCG et ces CCLI passent alors en étant intG.

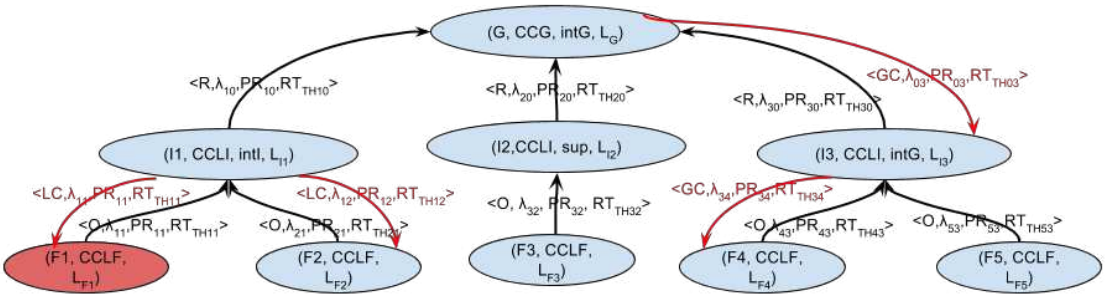

Figure 5.4 - Exemple de déploiement au niveau métier en phase d'intervention

Aussi pour cette phase d'intervention, un certain nombre de propriétés additionnelles aux précédentes de la phase de supervision doivent être respectées par le système. Tout CCLI en états intI ou intG doit être attaché au CCG et doit posséder au moins un CCLF. Le CCLI qui émettent un trafic de type LC doit être impérativement en état intI. Alors que celui qui émet un GC doit être en intG. Dans la section 5.4.2, la description formelle de ces propriétés est fournie.

La Figure 5.4 fournit un exemple de déploiement au niveau métier dans la phase d'intervention. En cas de danger / incident, sur un quai par exemple (entité F1), le CCLI associé (I1) le détecte et passe en état intI. Cette phase conduit le CCLI à élaborer un/des plans d'intervention locale à la station (par exemple, à l'échelle de la station) et l'envoi de commandes vers les équipements. Dans le même temps, il envoie un rapport vers le CCG qui peut élaborer un plan d'intervention globale en fonction de la criticité de l'incident.

Le modèle du point de vue métier permet premièrement de décrire formellement le rôle métier des différents nœuds. Il permet aussi de faire apparaître les contraintes de QoS dans chaque phase et, en particulier, de traduire leurs changements lors du passage d'une phase à une autre. Ces entités sont ci-après raffinées en des entités logicielles opérationnelles qui vont assurer ces rôles tout en considérant les contraintes et exigences de la QoS.

Niveau opératoire

Au niveau métier, l'accent est mis sur les rôles des entités considérées. Au niveau opératoire, nous donnons une représentation des entités logicielles (applications et Middlewares) et des liens implémentant ces rôles. Chacune de ces entités peut être identifiée par un *n*-uplet :

- *Id* : identifiant de l'entité opératoire ;
- *Type :* type de l'entité opératoire. Nous définissons deux types d'entités :
  - **SUPINT :** assure deux rôles essentiels. Le premier rôle concerne la supervision en analysant les données d'observations qui arrivent depuis les équipements. Le deuxième rôle est celui d'intervention. L'entité assure ces rôles au niveau des zones couvertes par les entités de type CCLI et CCG du niveau métier. Si elle

est en zone CCLI, cette entité se charge aussi de la génération et l'envoi des rapports vers l'entité du même type de la zone CCG ;

- **MW :** joue le rôle de Middleware intermédiaire. Une entité de ce type est responsable du transport et du stockage des données, ainsi que de l'acheminement des commandes entre les équipements et les entités de type SUPINT. Les besoins en QoS étant uniquement gérés par cette entité, tout trafic qu'elle remonte au SUPINT est dépourvu d'attributs de QoS.

- C_{Id} : nombre de cœurs alloués à l'entité opératoire sur l'ensemble des cœurs de la machine sur laquelle elle est déployée (C_{MId}) ;
- M_{Id} *:* identifiant de la machine hébergeant l'entité opératoire ;
- *Etat :* état de la machine ; elle prend la valeur OK (en marche) ou NOK (en panne) ;
- *Type Machine :* pour la description du type de la machine hébergeant l'entité. Cet attribut peut prendre pour valeur :
  - φ_{VIR} pour une machine déployée dans un environnement virtualisé (typiquement un cloud computing) ;
  - φ_{PRM} pour une machine physique dite *primaire* susceptible d'être remplacée ou confortée par une machine dite *secondaire* (φ_{SEC}) en cas de besoin ;
  - φ_U pour une machine physique qui n'admet pas de machine secondaire.
- L_{Id} *:* pour Localisation, représentant la zone de couverture de l'entité.

Dans une perspective de modélisation basée graphe, une entité est ainsi représentée par un nœud *N(Id, Type, C_{Id}/C_{MId}, M_{Id}, Etat, Type_Machine, L_{Id})*. Pour la machine qui héberge une entité de type CCLI, nous supposons que l'application SUPINT ne peut bénéficier que d'un seul cœur, le Middleware s'exécutant sur la même machine pouvant bénéficier de tout ou partie des cœurs restants. Si la machine héberge une entité type CCG, cette machine étant susceptible d'être déployé dans un environnement virtualisé, nous supposons que l'entité SUPINT peut alors bénéficier d'un ou plusieurs cœurs.

Pareil que pour le niveau métier, le système de ce niveau doit aussi respecter un certain nombre de propriétés. L'entité SUPINT pour la gestion d'une zone donnée ne doit se trouver qu'au niveau d'une unique machine. Alors que deux entités MW ne doivent pas être hébergées par la même machine afin de gérer deux zones différentes. La description formelle de ces propriétés est donnée dans la section 5.5.1. La Figure 5.5 représente un exemple de déploiement du niveau opératoire en phase de supervision et qui respectent les propriétés précédentes.

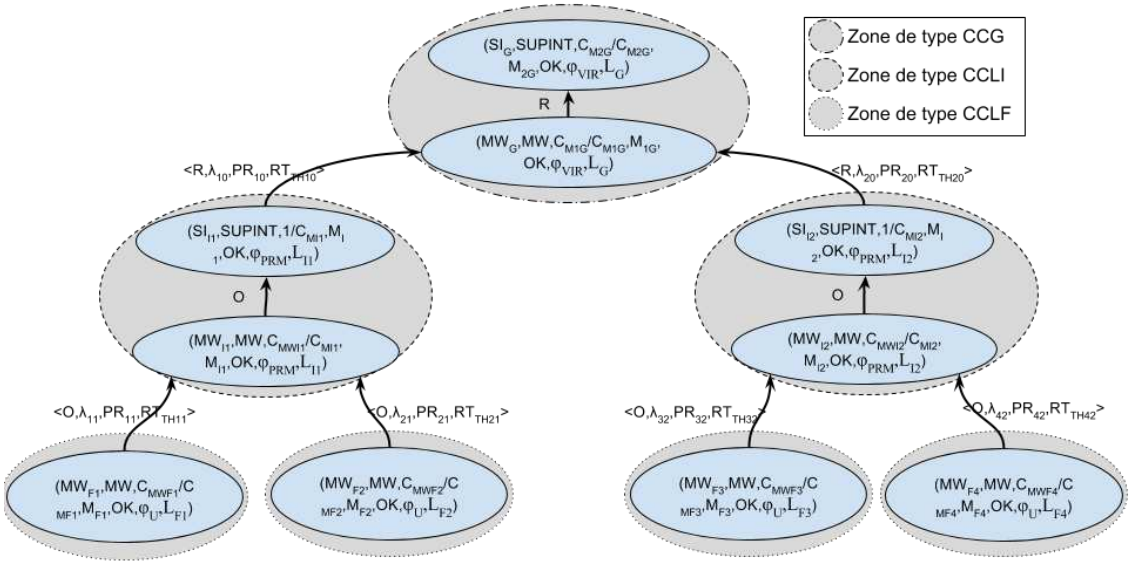

Figure 5.5 - Exemple d'architecture au niveau Middleware en phase de supervision

En phase d'intervention (Figure 5.6), les commandes (LC et GC) sont générées par les entités de type SUPINT (des zones CCLI et CCG respectivement). Ces commandes passent tout d'abord par l'entité Middleware de la zone émettrice qui assure leur redirection soit directement vers le MW distant si sa zone est de type CCLF (zone émettrice de type CCLI), soit vers l'entité SUPINT distante si sa zone est de type CCLI (zone émettrice de type CCG). Le rôle de l'entité SUPINT de la zone CCLI consiste dans ce cas à traiter la commande générale en vue de son envoi aux bons équipements avec les bons attributs. Les propriétés architecturales décrites en phase de supervision restent les mêmes pour phase d'intervention.

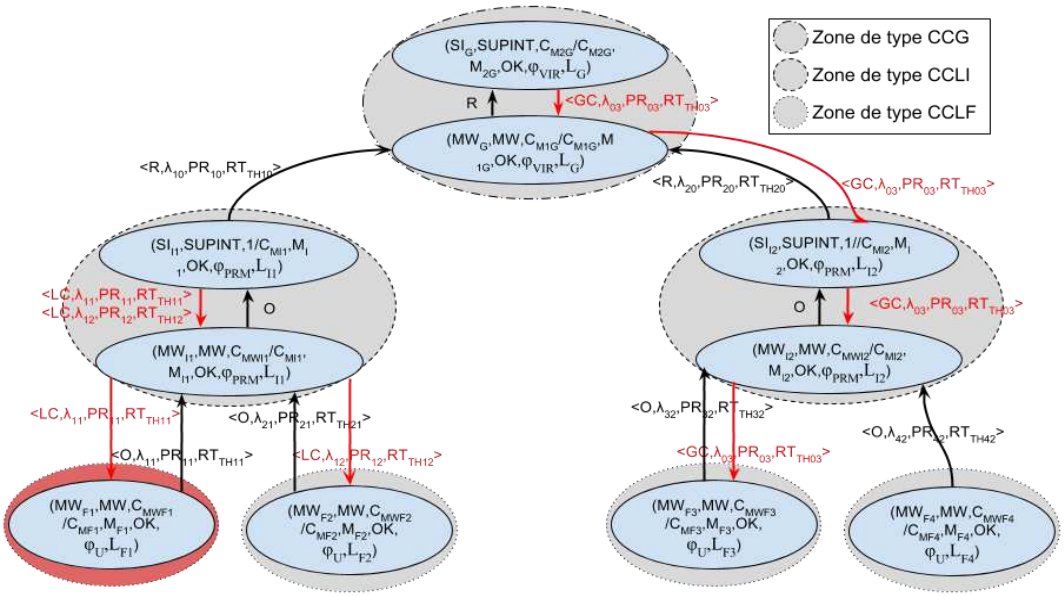

Figure 5.6 - Exemple de déploiement au niveau opératoire en phase d'intervention

Dans la section suivante, la description du style architectural permettant de modéliser chaque niveau d'abstraction est donnée pour chaque phase d'exécution du cas d'étude.

5.4. ARCHITECTURE AU NIVEAU MÉTIER

Cette section est spécifique à l'architecture du niveau métier. Elle décrit le style architectural relatif à chaque phase. Chaque description débute par une réflexion sur les propriétés architecturales que toute configuration doit respecter. Elle est ensuite formalisée à l'aide d'une grammaire de graphes dont toute instance vérifie les propriétés précédentes. L'instanciation du style architectural donne naissance à une instance respectant l'ensemble des propriétés. Ce respect des propriétés permet de certifier une forme de conformité entre la grammaire et le système à spécifier. Le style architectural du niveau métier permet aussi de faire apparaître les besoins en QoS ainsi que leurs changements dynamiques (notamment, lors du passage d'une phase à une autre) à prendre en considération par le niveau d'abstraction sous-jacent. Dans cette section, nous donnons les styles architecturaux (propriétés architecturales et grammaire) du niveau métier pour les phases de supervision et d'intervention.

Notons que dans les règles de réécriture, nous distinguons deux types d'attributs. Le premier, dit *variable*, peut prendre et donc être associé à n'importe quelle valeur. Le second, dit *fixe*, précise une valeur : l'attribut ne peut être associé qu'à un attribut possédant cette valeur. Dans la suite de ce chapitre, la déduction des attributs fixes dans les règles étant intuitive dans notre cas, ils ne seront pas représentés dans un format spécifique afin de les distinguer des attributs variables.

5.4.1. Style architectural pour le niveau métier en phase de supervision

Propriétés architecturales

Les propriétés architecturales négatives stipulent la non existence d'une sous configuration durant l'exécution du système quelle que soit la reconfiguration d'architecture réalisée. Elles sont décrites via des règles de réécriture réduites où la vérification de leur validité dépend de l'applicabilité de ces règles. Une propriété négative est donc vérifiée si elle n'est pas applicable. Par opposition, une propriété positive est violée si elle n'est pas applicable.

Pour notre spécification, nous nous basons essentiellement sur les propriétés négatives. Ces propriétés architecturales caractérisent le niveau métier en phase de supervision. Nous utilisons les modèles selon la structure des règles de transformation réduites de type (L; N; E) où N traduit une condition d'application négative (NAC).

$prop_{MS,1}$ = (L={N1(Id_{G1}, CCG, sup, L_{IdG1})}, N={N2(Id_{G2}, CCG, sup, L_{IdG2})}; E={ }), **NEG**
$prop_{MS,2}$ = (L={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; N={N2(Id_{G}, CCG, sup, L_{IdG}), N1→^{Etiq12}N2}; E={Etiq12(R, λ_{12}, PR_{12}, RT_{TH12})}), **NEG**
$prop_{MS,3}$ = (L={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; N={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq12}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})}), **NEG**
$prop_{MS,4}$ = (L={N1(Id_{LF}, CCLF, L_{IdLF})}; N={N2(Id_{LI}, CCLI, sup, L_{IdLI}), N1→^{Etiq12}N2}; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12})}), **NEG**
$prop_{MS,5}$ = (L={N1(Id_{LF}, CCLF, L_{IdLF}), N2(Id_{LI1}, CCLI, X, L_{IdLI1}), N1→^{Etiq12}N2}; N={N3(Id_{LI2}, CCLI, Y, L_{IdLI2}), N1→^{Etiq13}N3}; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12}), Etiq13(O, λ_{13}, PR_{13}, RT_{TH13})}), **NEG**

La propriété **$prop_{MS,1}$** est une propriété positive qui exprime le fait que le niveau métier possède exactement un seul CCG fonctionnel tel que spécifié par les standards SmartM2M et oneM2M (mais pouvant être déployé sur plusieurs machines distribuées).

La propriété **$prop_{MS,2}$** implique que tous les CCLI sont toujours reliés au CCG. **$prop_{MS,3}$** implique que chaque CCLI possède au moins un CCLF. **$prop_{MS,4}$** est une propriété négative qui implique qu'un CCLF est forcément lié à un CCLI. Enfin, **$prop_{MS,5}$** vérifie qu'un CCLF ne peut être relié qu'à un seul CCLI.

Productions de grammaires du niveau métier en phase de supervision

La grammaire $GG_{M,S}$ décrit le style architectural correspondant au niveau métier en phase de supervision. Elle comporte les productions de grammaire qui décrivent les propriétés de l'architecture dans cette phase. Nous adoptons la notation suivante pour la description de chaque production : p = (L; K; R; E) où : L correspond au graphe mère, K graphe à conserver, R graphe fille et E l'ensemble des étiquettes. Chaque étiquette comporte le type de donnée véhiculé, son taux d'arrivée à l'entité réceptrice, sa priorité, ainsi que le temps de réponse requis pour le traitement de cette donnée par l'entité réceptrice.

$GG_{M,S}$ = (AX, NT, T, P) avec : T={N(Id, CCG/CCLI, sup, L_{Id}), N(Id, CCLF, L_{Id})}, NT={ }, et p={$p_{MS,1}$, $p_{MS,2}$, $p_{MS,3}$}
$p_{MS,1}$ = (L={AX};

K={ }; R={N1(Id_G, CCG, sup, L_{IdG}), N2(Id_{LI}, CCLI, sup, L_{IdLI}), N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})
$p_{MS,2}$ = (L={N1(Id_G, CCG, sup, L_{IdG})}; K={N1(Id_G, CCG, sup, L_{IdG})}; R={N2(Id_{LI}, CCLI, sup, L_{IdLI}), N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})
$p_{MS,3}$ = (L={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; K={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; R={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})})

La production $\mathbf{p_{MS,1}}$ permet de générer un CCG en état de supervision (sup), un CCLI en état sup et qui envoie des rapports R vers le CCG, ainsi qu'un CCLF qui envoie des observations O vers le CCLI. A chaque application de la production $\mathbf{p_{MS,2}}$, un CCLI supplémentaire en phase de supervision est généré ainsi qu'un CCLF qui lui est attaché. Cette production exige l'existence préalable du CCG. La production $\mathbf{p_{MS,3}}$ permet de générer un CCLF supplémentaire attaché au CCLI introduit dans la partie L de la production.

L'exemple de déploiement représenté par la Figure 5.3 (niveau métier en phase de supervision) peut être obtenu en appliquant la grammaire $GG_{M,S}$ selon le chemin suivant :

(1) $p_{MS,1}$ pour générer G, I1 et F1 ;
(2) $p_{MS,2}$ pour générer F2 attaché à I1 ;
(3) $p_{MS,2}$ pour générer I2 attaché à G ;
(4) $p_{MS,3}$ pour générer F3 attaché à I2 ;
(5) $p_{MS,2}$ pour générer I3 attaché à G ;
(6) $p_{MS,3}$ pour générer F4 attaché à I3
(7) $p_{MS,3}$ pour générer F5 attaché à I3.

D'autres chemins peuvent être empruntés afin de générer la même instance de cette architecture.

Satisfaction des propriétés architecturales

Par définition et construction, les propriétés décrites précédemment sont nécessairement respectées par les instances de la grammaire. Nous allons démontrer cela pour la propriété impliquant que tout CCLI est relié au CCG ($\mathbf{prop_{MS,2}}$) à titre d'exemple. La démonstration de satisfaction des autres propriétés est similaire et triviale.

Montrons que $\mathbf{prop_{MS,2}}$ est respectée par toute instance de la grammaire. Soit une instance G de la grammaire. Par l'absurde, supposons donc que G ne respecte pas la propriété $\mathbf{prop_{MS,2}}$. Cela signifie qu'il existe un CCLI qui n'est pas relié au CCG. Si le lien n'existe pas, ça veut dire que soit (1) il n'a jamais été ajouté, soit (2) il a été ajouté puis supprimé.

Par définition, G peut être obtenue à partir de l'axiome en appliquant une séquence de règles de production. Or, aucune règle de production de la grammaire ne peut entraîner de suppression d'arc, ces productions étant génératives et limitées à l'ajout de nœud et arcs. Le cas (2) est donc impossible. Considérons donc le cas (1). Seule la règle $\mathbf{p_{MS,2}}$ permet de déployer des CCLIs, ils

ont donc nécessairement été ajoutés lors d'une application de celle-ci. La règle $\mathbf{p_{MS,2}}$ établit tous les liens nécessaires avec le CCG. Le cas (1) est donc également impossible.

Ainsi, il ne peut exister de CCLI non relié au CCG au sein de G. Par conséquent, G respecte nécessairement la propriété $\mathbf{prop_{MS,2}}$.

5.4.2. Style architectural correspondant à la phase d'intervention

Propriétés architecturales

Nous présentons la caractérisation des propriétés architecturales au niveau métier et en phase d'intervention. Elles enrichissent les propriétés précédentes qui restent valables ici.

$prop_{MI,1}$ = (L={N1(Id_{LI}, CCLI, intI, L_{IdLI})}; N={N2(Id_G, CCG, sup/intG, L_{IdG}), N1→^{Etiq12}N2, N2→^{Etiq21}N1}; E={Etiq12(R, λ_{12}, PR_{12}, RT_{TH12})}), **NEG**
$prop_{MI,2}$ = (L={N1(Id_{LI}, CCLI, intG, L_{IdLI})}; N={N2(Id_G, CCG, intG, L_{IdG}), N1→^{Etiq12}N2, N2→^{Etiq21}N1}; E={Etiq12(R, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(GC, λ_{21}, PR_{21}, RT_{TH21})}), **NEG**
$prop_{MI,3}$ = (L={N1(Id_{LI}, CCLI, X, L_{IdLI})}; N={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1, N1→^{Etiq12}N2}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21}), Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12})}), **NEG** X = sup ou intG
$prop_{MI,4}$ = (L={N1(Id_{LI}, CCLI, X, L_{IdLI})}; N={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1, N1→^{Etiq12}N2}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21}), Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12})}), **NEG** X = sup ou intI
$prop_{MI,5}$ = (L={N1(Id_{LI}, CCLI, intI, L_{IdLI})}; N={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1, N1→^{Etiq12}N2}; E={Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})}), **NEG**
$prop_{MI,6}$ = (L={N1(Id_G, CCG, intG, L_{IdG}), N2(Id_{LI}, CCLI, intG, L_{IdLI}), N2→^{Etiq21}N1, N1→^{Etiq12}N2}; N={N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2}; E={ Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq23(GC, λ_{23}, PR_{23}, RT_{TH23}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})}), **NEG**
$prop_{MI,7}$ = (L={N1(Id_{LF}, CCLF, L_{IdLF})}; N={N2(Id_{LI}, CCLI, intI, L_{IdLI}), N1→^{Etiq12}N2, N2→^{Etiq21}N1}; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(LC, λ_{21}, PR_{21}, RT_{TH21})}), **NEG**
$prop_{MI,8}$ = (L={N1(Id_{LF}, CCLF, L_{IdLF})}; N={N2(Id_{LI}, CCLI, intG, L_{IdLI}), N1→^{Etiq12}N2, N2→^{Etiq21}N1}; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(GC, λ_{21}, PR_{21}, RT_{TH21})}), **NEG**
$prop_{MI,9}$ = (L={N1(Id_{LF}, CCLF, L_{IdLF})}; N={N2(Id_{LI}, CCLI, X, L_{IdLI}), N1→^{Etiq12}N2}; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12})}), **NEG**

La propriété négative $\mathbf{prop_{MI,1}}$ et $\mathbf{prop_{MI,2}}$ impliquent que le CCLI en phase d'intervention intermédiaire (intI) ou globale (intG) est impérativement lié au CCG. $\mathbf{prop_{MI,3}}$ propriété qui implique que tout CCLI émettant un LC doit être en état intI. Alors que $\mathbf{prop_{MI,4}}$ implique que tout CCLI transférant une GC vers un CCLF doit être en état intG. $\mathbf{prop_{MI,5}}$ implique que le CCLI en intI possède au moins un CCLF auquel il envoie des commandes locales (LC). $\mathbf{prop_{MI,6}}$ précise que pour un CCLI en intG qui reçoit une commande CG depuis le CCG doit posséder au moins un CCLF vers qui envoyer cette commande GC. Enfin, les propriétés négatives $\mathbf{prop_{MI,7}}$, $\mathbf{prop_{MI,8}}$ et $\mathbf{prop_{MI,9}}$ impliquent qu'un CCLF est forcément lié à un CCLI.

Productions de grammaires du niveau métier en phase d'intervention

La grammaire $GG_{M,I}$ décrit le style architectural correspondant au niveau application et à la phase d'intervention.

$GG_{M,I}$=(AX, NT, T, P) avec : T={N(Id, CCG/CCLI, sup/intG/intI, L_{Id}), N(Id_{LF}, CCLF, L_{IdLF})}, NT={ }, et p={$p_{MI,1}$, $p_{MI,2}$, $p_{MI,3}$, $p_{MI,4}$, $p_{MI,5}$, $p_{MI,6}$, $p_{MI,7}$, $p_{MI,8}$}
$p_{MI,1}$ = (L={AX}; K={ }; R={N1(Id_G, CCG, intG, L_{IdG}), N2(Id_{LI}, CCLI, intI, L_{IdLI}), N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2, N2→^{Etiq23}N3)} E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq23(LC, λ_{23}, PR_{23}, RT_{TH23})})
$p_{MI,2}$ = (L={N1(Id_G, CCG, intG, L_{IdG})}; K={N1(Id_G, CCG, intG, L_{IdG})}; R={N2(Id_{LI}, CCLI, intI, L_{IdLI}), N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2, N2→^{Etiq23}N3}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq23(LC, λ_{23}, PR_{23}, RT_{TH23})})
$p_{MI,3}$ = (L={N1(Id_{LI}, CCLI, intI, L_{IdLI})}; K={N1(Id_{LI}, CCLI, intI, L_{IdLI})}; R={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1, N1→^{Etiq12}N2}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21}), Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12})})
$p_{MI,4}$ = (L={N1(Id_G, CCG, intG, L_{IdG})}; K={N1(Id_G, CCG, intG, L_{IdG})}; R={N2(Id_{LI}, CCLI, intG, L_{IdLI}), N1→^{Etiq12}N2, N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2, N2→^{Etiq23}N3}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq23(GC, λ_{23}, PR_{23}, RT_{TH23})})
$p_{MI,5}$ = (L={N1(Id_{LI}, CCLI, intG, L_{IdLI}), N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1}; K={N1(Id_{LI}, CCLI, intG, L_{IdLI}), N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1}; R={N1→^{Etiq12}N2}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21}), Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12})})
$p_{MI,6}$ = (L={N1(Id_{LI}, CCLI, intG, L_{IdLI})}; K={N1(Id_{LI}, CCLI, intG, L_{IdLI})}; R={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})})
$p_{MI,7}$ = (L={N1(Id_G, CCG, intG, L_{IdG})}; K={N1(Id_G, CCG, intG, L_{IdG})}; R={N2(Id_{LI}, CCLI, sup, L_{IdLI}), N2→^{Etiq21}N1, N3(Id_{LF}, CCLF, L_{IdLF}), N3→^{Etiq32}N2}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})
$p_{MI,8}$ = (L={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; K={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; R={N2(Id_{LF}, CCLF, L_{IdLF}), N2→^{Etiq21}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})})

La première production $\mathbf{p_{MI,1}}$ permet de générer un CCG en état intG, un CCLI en état intI qui envoie des rapports R vers le CCG, ainsi qu'un CCLF qui envoie des observations O au CCLI

et reçoit des commandes locales (LC) de la part de ce dernier. $\mathbf{p_{MI,2}}$ permet de générer un CCLI en état d'intervention intI et un CCLF, le CCLF envoyant des observations au CCLI qui lui envoie des LC. La production $\mathbf{p_{MI,3}}$ génère un CCLF supplémentaire attaché au CCLI en état d'intervention intI, le CCLF envoyant des observations au CCLI qui lui envoie des LC. $\mathbf{p_{MI,4}}$ crée un CCLI qui reçoit la commande général (GC) de la part du CCG pour la gestion des équipements et la transfère au CCLF concerné. Quand $\mathbf{p_{MI,5}}$ est appliquée, elle permet d'ajouter un lien GC du CCLI en état de supervision globale (intG) vers le CCLF. La production $\mathbf{p_{MI,6}}$ permet de générer un CCLF qui envoie les observations sans recevoir de commandes générales (GC). $\mathbf{p_{MI,7}}$ introduit un CCLF et un CCLI en état de supervision qui ne reçoit aucune commande du CCG. Alors que la production $\mathbf{p_{MI,8}}$ permet de générer un CCLF supplémentaire attaché au CCLI en état de supervision et qui ne reçoit aucune commande.

L'exemple de déploiement représenté par la Figure 5.4 (niveau métier en phase d'intervention) peut être obtenue en appliquant la grammaire $GG_{M,I}$ selon le chemin suivant par exemple :

(1) $\mathbf{p_{MI,1}}$ pour générer G, I1 et F1 ;
(2) $\mathbf{p_{MI,3}}$ pour générer F2 attaché à I1 ;
(3) $\mathbf{p_{MI,4}}$ pour générer F4 et I3 attaché à G ;
(4) $\mathbf{p_{MI,7}}$ pour générer F3 et I2 attaché à G ;
(5) $\mathbf{p_{MI,8}}$ pour générer F5 attaché à I3.

Passage de la phase de supervision à la phase d'intervention

Le passage d'une phase à une autre peut être guidé par les règles de production. Ce passage n'est pas détaillé dans cette thèse. Nous donnons ici à titre d'exemple deux règles (RM_{SI1} et RM_{SI2}) qui permettent de passer de la phase de supervision à la phase d'intervention pour le CCLI où se passe l'incident ou le danger.

RM_{SI1} = (L={N1(Id_{LI}, CCLI, sup, L_{IdLI})}; K = { }; R={N2(Id_{LI}, CCLI, intI, L_{IdLI})}; C={(N2, N3(Id_{LF}, CCLF, L_{IdLF}), Etiq23/Etiq23, out, out)}; E={Etiq23(O, λ_{23}, PR_{23}, RT_{TH23})})
RM_{SI2} = (L={N1(Id_{LI}, CCLI, intI, M_{IdLI}), N2(Id_{LF}, CCLF, L_{IdLF}), N2$\rightarrow^{Etiq21}$N1}; K = {N1(Id_{LI}, CCLI, intI, M_{IdLI}), N2(Id_{LF}, CCLF, L_{IdLF})}; R={N1$\rightarrow^{Etiq12}$N2}; C={ }; E={Etiq12(O, λ_{12}, PR_{12}, RT_{TH12}), Etiq21(LC, λ_{21}, PR_{21}, RT_{TH21})})

La règle RM_{SI1} décrit le changement de phase d'un CCLI de la supervision (sup) vers l'intervention intermédiaire (intI), en sauvegardant tous les arcs de type O de tout CCF vers le CCLI. La règle RM_{SI2} assure l'ajout d'un arc LC d'un CCLI en phase intI vers un CCLF, débutant l'envoie de commande au vu du traitement de l'incident ou du danger pour l'évacuation.

D'autres règles de passage peuvent être fournies mais ne font pas partie du périmètre de cette thèse, étant donné que le focus est plus mis sur les actions de reconfiguration.

5.5. ARCHITECTURE AU NIVEAU OPÉRATOIRE

Cette section donne le style architectural permettant de décrire le niveau opératoire pour les deux phases d'exécution. A ce niveau, l'instanciation du style architectural fournit un exemple de déploiement permettant de faire apparaître les exigences en QoS du trafic ainsi que les entités qui doivent les satisfaire. Les entités de type CCG et CCLI du niveau métier se traduisent au niveau opératoire par deux composants. Le premier est le Middleware (MW) des communications, du stockage et transfert des données. Le second est l'application de supervision et intervention (SUPINT). Pour le CCLI, ces deux composants sont déployés sur une même machine physique primaire (φ_{PRM}). Vu le rôle important que joue le CCLI dans la gestion locale, la machine primaire dispose d'une machine secondaire qui sert soit pour la substitution en cas de défaillance ou la duplication pour le respect des besoins en QoS. Quant au CCG, puisqu'il se trouve dans un environnement cloud dans le contexte que nous considérons, chaque composant est déployé sur une machine virtuelle à part. Pour le CCLF, il n'admet que le MW qui est responsable de la récupération des données depuis les équipements et leur transfert vers le MW du CCLI, ainsi que la réception des commandes dans l'autre sens.

5.5.1. Style architectural correspondant à la phase de supervision

Propriétés architecturales

Nous présentons ici la caractérisation des propriétés architecturales au niveau métier et en phase de supervision.

$prop_{OS,1}$ = (L={N1(Id_{IS1}, SUPINT, $C_{SUPINTM1}/C_{M1}$, M_1, Etat, φ_1, L_1), N2(Id_{IS2}, SUPINT, $C_{SUPINTM1}/C_{M1}$, M_1, Etat, φ_1, L_2)}; N={ }; E={ }), **NEG**
$prop_{OS,2}$ = (L={N1(Id_{IS1}, SUPINT, $C_{SUPINTM1}/C_{M1}$, M_1, Etat, φ_1, L_1), N2(Id_{IS2}, SUPINT, $C_{SUPINTM2}/C_{M2}$, M_2, Etat, φ_2, L_1)}; N={ }; E={ }), **NEG**
$prop_{OS,3}$ = (L={N1(Id_{IS1}, MW, C_{MWM1}/C_{M1}, M_1, Etat, φ_1, L_1), N2(Id_{IS2}, MW, C_{MWM1}/C_{M1}, M_1, Etat, φ_1, L_2)}; N={ }; E={ }), **NEG**

Les propriétés négatives **$prop_{OS,1}$** et **$prop_{OS,2}$** expriment le fait que deux SUPINT ne doivent pas se trouver dans la même machine ni gérer les mêmes zones. Alors que la propriété négative **$prop_{OS,3}$** exprime le fait que deux Middlewares ne peuvent pas être hébergés sur la même machine pour gérer deux zones différentes. Ces règles sont valables quel que soit l'état des machines (OK ou NOK). Le non-respect de certaines de ces règles font passer le système en mode "dégradé" sans pour autant arrêter son fonctionnement.

Productions de grammaires du niveau opératoire en phase de supervision

La grammaire $GG_{O,S}$ décrit le style architectural correspondant au niveau opératoire en phase de supervision. Ici, nous nous intéressons en particulier à la QoS du trafic entrant gérée par le Middleware.

$GG_{O,S}$=(AX, NT, T, P) avec : T={N(Id, MW/SUPINT, cœurs$_{/SUPINT}$/cœurs$_{MId}$, M_{Id}, $Etat_{MId}$, $Type_{MId}$, L_{Id})}, NT={ }, et p={$p_{OS,1}$, $p_{OS,2}$, $p_{OS,3}$}
$p_{OS,1}$ = (L={AX};

K={ }; R={N1(Id_{ISG}, SUPINT, C_{M1G}/C_{M1G}, M_{2G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M2G}/C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2→^{R}N1, N3(Id_{ISI1}, SUPINT, $1/C_{MI1}$, M_{I1}, OK, φ_{PRM}, L_{I1}), N3→^{Etiq32}N2, N4(Id_{MWI1}, MW, C_{MWI1}/C_{MI1}, M_{I1}, OK, φ_{PRM}, L_{I1}), N4→^{O}N3, N5(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N5→^{Etiq54}N4}; E={Etiq32(R, λ_{32}, PR_{32}, RT_{TH32}), Etiq54(O, λ_{54}, PR_{54}, RT_{TH54})}})
$p_{OS,2}$ = (L={N1(Id_{MWG}, MW, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G)}; K={N1(Id_{MWG}, MW, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G)}; R={N2(Id_{ISI1}, SUPINT, $1/C_{MI1}$, M_{I1}, OK, φ_{PRM}, L_{I1}), N2→^{Etiq21}N1, N3(Id_{MWI1}, MW, C_{MWI1}/C_{MI1}, M_{I1}, OK, φ_{PRM}, L_{I1}), N3→^{O}N2, N4(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4→^{Etiq43}N3}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq43(O, λ_{43}, PR_{43}, RT_{TH43})})
$p_{OS,3}$ = (L={N1(Id_{MWI1}, MW, C_{MWI1}/C_{M11}, M_{I1}, OK, φ_{PRM}, L_{I1})}; K={N1(Id_{MWI1}, MW, C_{MWI1}/C_{M11}, M_{I1}, OK, φ_{PRM}, L_{I1})}; R={N2(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N2→^{Etiq21}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})})

La production $\mathbf{p_{OS,1}}$ permet d'initialiser le système en déployant une configuration valide minimale. Elle génère deux couples SUPINT/MW couvrant le CCG (Localisation = L_G) et couvrant un CCLI (Localisation = L_{I1}) attaché au couple précédent, ainsi qu'un MW couvrant un CCLF envoyant des Observations et des informations de QoS au MW du CCLI. Ces composants communiquent conformément aux descriptions de la section précédente. La production $\mathbf{p_{OS,2}}$ décrit l'ajout d'un couple SUPINT/MW couvrant un CCLI en phase de supervision (déduction faite via l'arc). Le SUPINT nouvellement déployé envoie des rapports comportant des exigences de QoS au MW couvrant le CCG. Le CCLI exigeant au moins un CCLF, un MW de zone CCLF est également créé avant de lui être attaché. $\mathbf{p_{OS,3}}$ permet de générer un MW sur une zone de type CCLF en l'attachant au MW d'un CCLI introduit dans la partie L de la production.

L'exemple de déploiement représenté par la Figure 5.5 (niveau opératoire en phase de supervision) peut être obtenu en appliquant la grammaire $GG_{M,S}$ selon le chemin suivant :

(1) $\mathbf{p_{OS,1}}$ pour générer SI_G, MW_G, SI_{I1}, MW_{I1} et MW_{F1} ;
(2) $\mathbf{p_{OS,3}}$ pour générer MW_{F2} attaché à MW_{I1} ;
(3) $\mathbf{p_{OS,2}}$ pour générer SI_{I2} et MW_{I2} attaché à MW_G ;
(4) $\mathbf{p_{OS,3}}$ pour générer MW_{F3} attaché à MW_{I2} .
(5) $\mathbf{p_{OS,3}}$ pour générer MW_{F4} attaché à MW_{I2}.

D'autres chemins peuvent être empruntés afin de générer la même instance de cette architecture.

5.5.2. Style architectural correspondant à la phase d'intervention

Propriétés architecturales

Puisque les propriétés architecturales de la phase de supervision sont génériques, ne faisant pas apparaître les arcs entre les nœuds, elles restent les mêmes dans la phase d'intervention.

Productions de grammaires du niveau opératoire en phase d'intervention

La grammaire $GG_{O,I}$ décrit le style architectural correspondant au niveau opératoire en phase d'intervention.

$GG_{O,I}$ = (AX, NT, T, P) avec : T={N(Id, MW/SUPINT, cœurs/SUPINT/coeursM_{Id}, M_{Id}, $Etat_{MId}$, $Type_{MId}$, L_{Id})}, NT={ }, et p={$p_{OI,1}$, $p_{OI,2}$, $p_{OI,3}$, $p_{OI,4}$, $p_{OI,5}$, $p_{OI,6}$, $p_{OI,7}$, $p_{OI,8}$}
$p_{OI,1}$ = (L={AX}; K={ }; R={N1(Id_{ISG}, SUPINT, C_{M2G}/C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G), N2→^{R}N1, N3(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N3→^{Etiq32}N2, N4(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N4→^{O}N3, N3→^{Etiq34}N4, N5(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N5→^{Etiq54}N4, N4→^{Etiq45}N5}; E={Etiq32(R, λ_{32}, PR_{32}, RT_{TH32}), Etiq34(LC, λ_{34}, PR_{34}, RT_{TH34}), Etiq54(O, λ_{54}, PR_{54}, RT_{TH54}), Etiq45(LC, λ_{45}, PR_{45}, RT_{TH45})})
$p_{OI,2}$ = (L={N1(Id_{MWG}, MW, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G)}; K={N1(Id_{MWG}, MW, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G)}; R={N2(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N3(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N4(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N2→^{Etiq21}N1, N3→^{O}N2, N2→^{Etiq23}N3, N3→^{Etiq34}N4, N4→^{Etiq43}N3}; E={Etiq21(R, λ_{21}, PR_{21}, RT_{TH21}), Etiq23(LC, λ_{23}, PR_{23}, RT_{TH23}), Etiq34(LC, λ_{34}, PR_{34}, RT_{TH34}), Etiq43(O, λ_{43}, PR_{43}, RT_{TH43})})
$p_{OI,3}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MW1}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N2→^{O}N1, N3→^{Etiq32}N2}; K={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MW1}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N2→^{O}N1, N3→^{Etiq32}N2}; R={N1→^{Etiq12}N2, N2→^{Etiq23}N3}; E={Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12}), Etiq23(LC, λ_{23}, PR_{23}, RT_{TH23}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})
$p_{OI,4}$ = (L={N1(Id_{ISG}, SUPINT, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2→^{R}N1}; K={N1(Id_{ISG}, SUPINT, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2→^{R}N1}; R={N3(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N4(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N5(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N1→^{Etiq12}N2, N2→^{Etiq23}N3, N3→^{Etiq32}N2, N3→^{Etiq34}N4, N4→^{O}N3, N4→^{Etiq45}N5, N5→^{Etiq54}N4}; E={Etiq32(R, λ_{32}, PR_{32}, RT_{TH32}), Etiq54(O, λ_{54}, PR_{54}, RT_{TH54}), Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq23(GC, λ_{23}, PR_{23}, RT_{TH23}), Etiq34(GC, λ_{34}, PR_{34}, RT_{TH34}), Etiq45(GC, λ_{45}, PR_{45}, RT_{TH45})})
$p_{OI,5}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N1→^{Etiq12}N2}; K={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N1→^{Etiq12}N2}; R={N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N2→^{Etiq23}N3, N3→^{Etiq32}N2}; E={Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq23(GC, λ_{23}, PR_{23}, RT_{TH23}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})
$p_{OI,6}$ = (L={N1(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I)};

K={N1(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I)}; R={N2(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N2→^{Etiq21}N1}; E={Etiq21(O, λ_{21}, PR_{21}, RT_{TH21})})
$p_{OI,7}$ = (L={N1(Id_{ISG}, SUPINT, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2→^{R}N1}; K={N1(Id_{ISG}, SUPINT, C_{M1G}/C_{M1G}, M_{1G}, OK, φ_{VIR}, L_G), N2(Id_{MWG}, MW, C_{M2G}, M_{2G}, OK, φ_{VIR}, L_G), N2→^{R}N1}; R={N3(Id_{ISI1}, SUPINT, 1/C_{MI}, M_{I1}, OK, φ_{PRM}, L_{I1}), N3→^{Etiq32}N2, N4(Id_{MWI1}, MW, C_{MWI1}/C_{MI1}, M_{I1}, OK, φ_{PRM}, L_{I1}), N4→^{O}N3, N5(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N5→^{Etiq54}N4}; E={Etiq32(R, λ_{32}, PR_{32}, RT_{TH32}), Etiq54(O, λ_{54}, PR_{54}, RT_{TH54})})
$p_{OI,8}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N1→^{Etiq12}N2, N2→^{O}N1}; K={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N1→^{Etiq12}N2, N2→^{O}N1}; R={N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N3→^{Etiq32}N2}; E={Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32})})

La première production $\mathbf{p_{OI,1}}$ permet de générer deux couples MW/SUPINT couvrant les zones du CCG et d'un CCLI, ainsi qu'un MW couvrant un CCLF. Le MW du couple envoie des observations (zone CCLI) ou des rapports (zone CCG) à son SUPINT. Le SUPINT de la zone CCLI envoie des rapports R tandis que son MW envoie des LC au - et reçoit des observations du - MW du CCLF. La production $\mathbf{p_{OI,2}}$ génère un couple SUPINT/MW sur la zone d'un CCLI en intervention et un MW sur celle d'un CCLF. Ce dernier envoie des observation vers - et reçoit des LC du - SUPINT en passant par le MW du CCLI. La production $\mathbf{p_{OI,3}}$ permet d'ajouter les commandes locales (LC) a un MW d'une zone de type CCLF attaché au MW du CCLI en état d'intervention intI qui envoie des observations. $\mathbf{p_{OI,4}}$ décrit la création d'un couple SUPINT/MW couvrant une zone CCLI et d'un MW couvrant un CCLF. Ce dernier envoie des observations vers le SUPINT du CCLI en passant par le MW du CCLI. Il reçoit les commandes émanant de SUPINT du CCG et redirigées par SUPINT et MW du CCLI. $\mathbf{p_{OI,5}}$ permet de générer un MW d'un CCLF supplémentaire attaché au MW du CCLI en état d'intervention qui envoie les observations et reçoit les commandes générales (GC). La production $\mathbf{p_{OI,6}}$ est une production équivalente à $\mathbf{p_{OS,3}}$ de la phase de supervision. Elle permet de générer un MW d'un CCLF qui envoie les observations au SUPINT du CCLI en passant par le MW CCLI. $\mathbf{p_{OI,7}}$ permet de générer un CCLF et un CCLI qui ne reçoit aucune commande depuis le CCG. Alors que la propriété $\mathbf{p_{OI,8}}$ génère un MW d'un CCLF qui envoie les observations à SUPINT du CCLI en passant par le MW du même CCLI mais qui n'est pas concerné par les commandes générales (GC). Le tableau suivant (Tableau 5.1) résume le résultat de l'application de chaque production terme d'ajout de nouvelles entités et de interactions.

Entités présentes / Entités générées	**Vide**	**SUPINT + MW (CCG)**	**SUPINT + MW (CCLI)**
SUPINT + MW (CCG)	$p_{OI,1}$	-	-
SUPINT + MW (CCLI)	$p_{OI,1}$	$p_{OI,2}$(LC), $p_{OI,4}$(GC), $p_{OI,7}$(O)	-
SUPINT + MW (CCLF)	$p_{OI,1}$	$p_{OI,2}$(LC), $p_{OI,4}$(GC), $p_{OI,7}$(O)	$p_{OI,3}$(LC), $p_{OI,5}$(GC), $p_{OI,6}$, $p_{OI,8}$(GC)

Tableau 5.1 - Synthèse des productions en fonction des entités présentés en phase d'intervention

L'instance d'architecture représentée par la Figure 5.6 (niveau opératoire en phase d'intervention) peut être obtenue en appliquant la grammaire $GG_{M,I}$ selon le chemin suivant :

(1) $\mathbf{p_{OI,1}}$ pour générer SI_G, MW_G, SI_{I1}, MW_{I1} et MW_{F1} ainsi que les flux R, LC vers MW_{F1}, et O depuis MW_{F1} ;
(2) $\mathbf{p_{OI,6}}$ pour générer MW_{F2} attaché à MW_{I1} et les flux O vers MW_{I1}
(3) $\mathbf{p_{OI,3}}$ pour ajouter les commandes LC du SI_{I1} vers MW_{F2} en passant par MW_{I1} ;
(4) $\mathbf{p_{OI,4}}$ pour générer SI_{I2} et MW_{I2} attaché à MW_G, et MW_{F3} attaché à MW_{I2}. Le flux GC depuis IS_G vers MW_{F3} en passant par SI_{I2} et MW_{I2} est aussi généré ;
(5) enfin, $\mathbf{p_{OI,6}}$ pour générer MW_{F4} attaché à MW_{I2} et qui envoie des observations mais qui ne reçoit pas de flux GC.

Par la suite, nous donnons les règles décrivant les actions de reconfigurations afin de guider le composant de planification dans l'élaboration de plans de gestion de la QoS. Ce composant fait partie de l'AMS dédiée à la gestion de l'objectif local de QoS communiqué par l'AMM.

5.6. ACTIONS DE RECONFIGURATION GUIDÉES PAR LES MODÈLES

Les sections précédentes montrent qu'au niveau opératoire, les besoins en QoS (ici temps de réponse) varient en fonction des phases d'exécution. L'entité Middleware (MW) doit donc prendre en considération ces besoins afin de les satisfaire. L'adaptation dynamique intervient dans deux types de situations : (1) même activité opérationnelle avec incapacité du MW de satisfaire aux contraintes de QoS. Changement donc des configurations sous-jacentes pour satisfaire à la même activité applicative, et (2) changement d'activité applicative (passage de la phase de supervision à l'intervention ou inversement) qui induit l'apparition de nouveaux besoins en QoS. Cette adaptation est conduite par le composant de planification via un ensemble d'actions de reconfigurations. Dans le chapitre 2, nous avons proposé un ensemble de mécanismes pour l'adaptation structurelle et/ou comportementale du MW. Dans cette section, nous nous focalisons essentiellement sur une partie des actions de reconfiguration structurelles pour guider le composant de planification.

Ces actions entrent dans le cadre d'une approche guidée par les modèles. Cette approche repose sur un premier modèle analytique (basée sur la théorie des files d'attente) présenté dans le chapitre 4, ainsi qu'un deuxième modèle reposant sur la théorie des graphes et de la réécriture algébrique de graphes. Le modèle analytique permet de détecter analytiquement la non suffisance du nombre de cœurs pour la satisfaction des besoins en QoS. Il permet aussi, dans une phase de planification, d'estimer la quantité minimale de ressources (ici nombre de cœurs) à allouer à l'entité MW nécessaires pour le respect des contraintes de QoS liées aux besoins applicatifs. Le second modèle permet d'une part de décrire les actions de reconfiguration envisageables et les conditions de leur application, et d'autre part, valider l'applicabilité d'une action de reconfiguration. Ces règles viennent enrichir les productions de grammaire exprimant le style architectural au niveau opératoire pour les différentes phases, tout en respectant les propriétés architecturales.

Afin de limiter l'impact du mode *runtime* de la planification sur les performances du planificateur et ainsi que la gestion des besoins en QoS, l'approche se base sur l'application de la règle sur le premier nœud qui vérifie les conditions plutôt que de parcourir l'ensemble des nœuds de l'architecture. Dans les règles de réécriture concernant la description des actions de reconfiguration, nous les enrichissons en y ajoutant un champ *Cond* pour la description de la condition d'applicabilité de la règle. Ces conditions sont exprimées en grande partie via le modèle analytique.

5.6.1. Modèle analytique pour guider la planification

Le modèle à base de files d'attente enrichit la base de connaissance de la boucle autonomique. Il donne une représentation analytique du Middleware OM2M sous l'angle de performances au regard du traitement du trafic entrant. Ce modèle est initialement utilisé par le composant de monitoring pour le calcul du temps de réponse au moindre coût sur le système géré en se basant sur les taux d'arrivée. L'équation (5.1) présente le temps de réponse analytique émanant du modèle.

$$RT(i_r) = \frac{1}{\mu_H - \lambda} + \frac{1-p}{(1-p)n\mu_C - \lambda} + \frac{1-p}{(1-p)\mu_D(i_r) - p\lambda} \text{ avec : } \lambda = \sum_{j=1}^{R} \lambda_j \quad (5.1)$$

Le modèle analytique peut également être utilisé dans la phase de planification. En effet, dans une gestion orientée ressource, le modèle permet de fournir le nombre de cœurs alloués au Middleware et nécessaires pour le respect du temps de réponse seuil. L'équation (5.2) donne la condition sur le nombre de cœurs nécessaires (n) pour satisfaire à cette contrainte de temps de réponse (RT_{TH}).

$$\text{Soit : } RT_{TH} = min_{k=1,\dots,R}\,(RT_{THk})$$

$$n \geq \frac{1}{\mu_C\left(RT_{TH} - \frac{1}{\mu_H - \lambda} - \frac{1-p}{(1-p)\mu_D(i_r) - p\lambda}\right)} + \frac{\lambda}{(1-p)\mu_C} = FA'\left(\sum_{j=1}^{R} \lambda_j, min(RT_{THk})\right), n \in \aleph \quad (5.2)$$

Dans cette équation, un trafic de plusieurs sources peut arriver au Middleware depuis plusieurs sources avec différentes contraintes de temps de réponse. Chaque trafic entrant (k) possède une contrainte en temps de réponse RT_{THk}. Ici, nous avons pris le temps de réponse seuil minimal afin de garantir le respect des contraintes en QoS de toutes les sources. Cette approche permet de réduire la consommation énergétique puisque le nombre de cœurs déployés influence sur cette consommation au fur et à mesure que le taux d'arrivée augmente. La Figure 5.7 exprime la consommation énergétique par cœur déployé en fonction du taux d'arrivée pour des requêtes POST. Le trafic arrive vers le Middleware OM2M hébergé sur une plateforme de type Raspberry Pi 2.

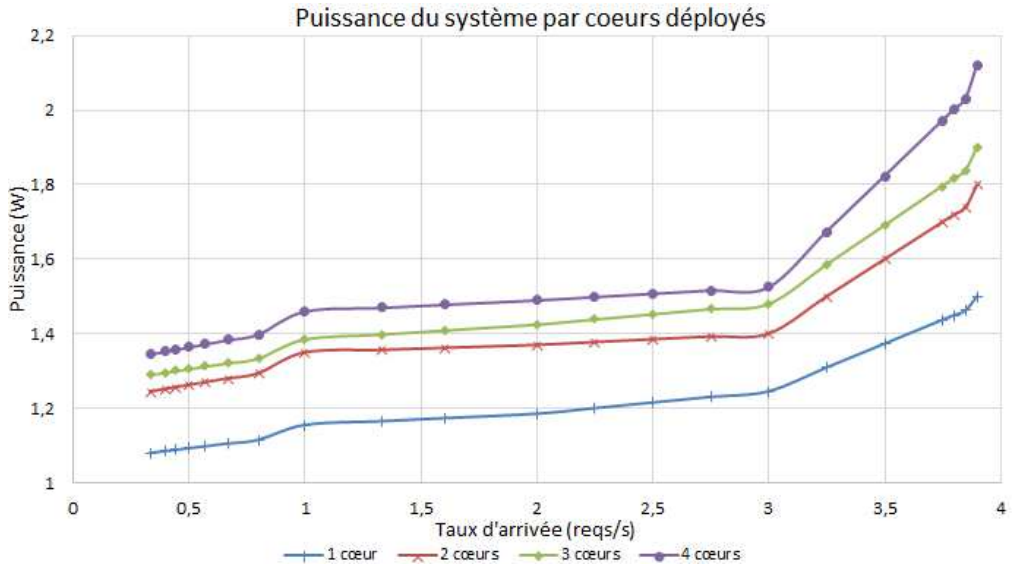

Figure 5.7 - Puissance du système par cœurs déployés

En fonction de l'environnement de déploiement du Middleware, trois actions de reconfiguration sont possibles :

1. **Redimensionnement :** action possible essentiellement dans les environnements de virtualisation où elle peut être guidée par un SLA. Dans le cas d'un environnement physique, nous allons utiliser cette action en redimensionnant le nombre de cœurs alloués au MW dans la limite des cœurs disponibles ;
2. **Migration :** elle peut être guidée par des facteurs tel que la défaillance de la machine. Cette action n'est possible que pour les entités MW de la zone CCG qui sont déployées dans des environnements de virtualisation, ou les entités des zones CCLI qui admettent des machines secondaires ;
3. **Répartition de charge :** la répartition peut se faire dans un environnement réel en zone CCLI en répartissant la charge de la machine primaire avec celle secondaire (de duplication) et/ou avec les voisins. Elle peut se faire aussi dans un environnement de virtualisation en créant / supprimant des entités de duplication en fonction de la charge entrante.

Pour le redimensionnement et en utilisant le modèle pour déterminer le nombre de cœurs, nous pouvons avoir deux actions d'adaptation complémentaires pour respecter les contraintes en QoS tout en minimisant l'utilisation énergétique.

La première action est celle de l'activation graduelle des cœurs. Elle consiste en l'activation du cœur à chaque fois que le seuil critique. A titre d'exemple, pour un seuil critique de 280 ms, la Figure 5.8 donne l'évolution du temps de réponse analytique et de la puissance en fonction des taux d'arrivée vers l'entité Middleware via l'application de cette action de reconfiguration. Elle compare aussi la puissance fournie dans le cadre de cette action avec la puissance d'activation de 4 cœurs par défaut.

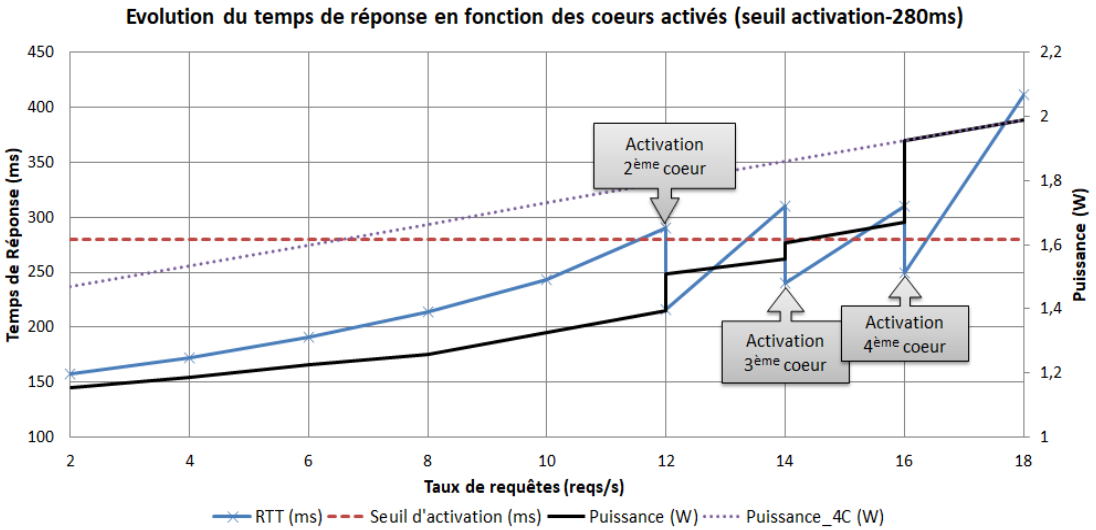

Figure 5.8 - Evolution du temps de réponse via l'action d'activation de cœurs

En complément à cette action, la désactivation graduelle des cœurs est proposée comme une seconde action. L'objectif de cette action concerne l'utilisation optimale des ressources en afin de minimiser la consommation énergétique lorsque le trafic est faible. La Figure 5.9 représente l'évolution du temps de réponse analytique ainsi que de la puissance en l'application de cette action. Le seuil de désactivation considéré à titre illustratif dans ce cas est de 250 ms.

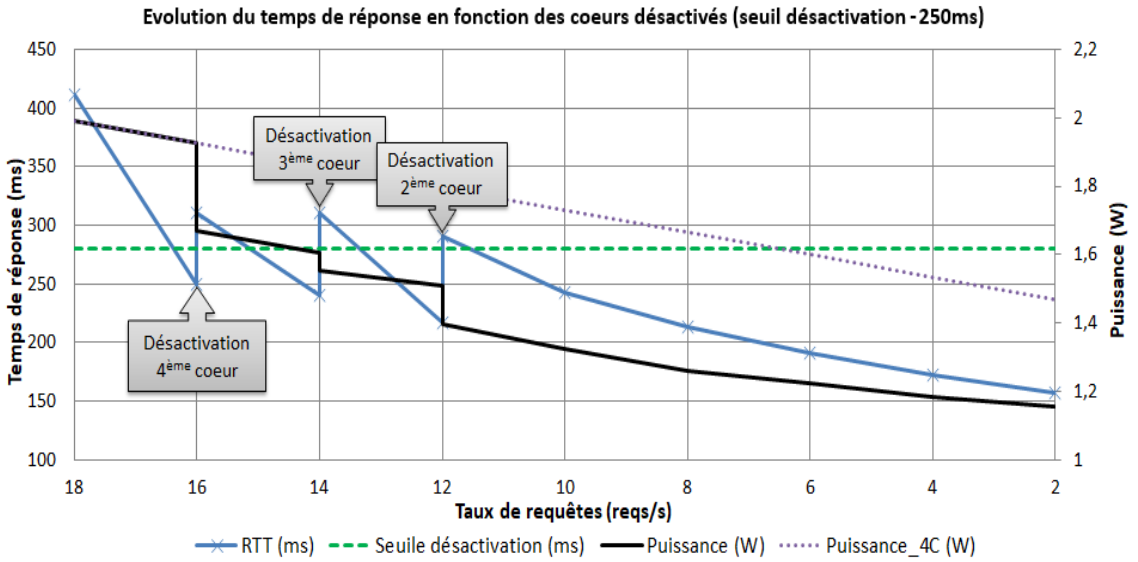

Figure 5.9 - Evolution du temps de réponse via l'action de désactivation de cœurs

Dans ce qui suit, nous nous focalisons sur le composant MW pour le CCLI déployé sur une machine physique primaire et possédant une machine secondaire. Nous proposons certaines actions de reconfiguration pour guider la planification. Ces actions sont séparées en deux types : le premier type d'actions traite la défaillance de la machine et son remplacement par un autre ; alors que le deuxième type traite le non-respect de la QoS. Dans ce deuxième cas, le modèle analytique est utilisé pour guider ces actions. Elles sont décrites via des règles de transformation qui prennent le modèle analytique comme condition de leur applicabilité. Elles suivent la structuration suivante : **Rx = (L={ }; K={ }; R={ }; C={ }; E={ }; Cond={ })**, où *C* exprime l'ensemble d'instructions de connexion, et *Cond* pour l'ensemble des conditions explicites d'applicabilité de la règle, exprimées à l'aide du modèle analytique.

5.6.2. Gestion de la défaillance de la machine primaire

Dans une zone CCLI, lorsque la machine primaire tombe en panne, elle est remplacée par la machine secondaire. Cette dernière est déployée avec le même nombre de cœurs utilisé par la machine primaire pour l'entité MW afin de supporter la même charge ainsi qu'un cœur pour l'entité SUPINT. Les règles suivantes guident cette migration lors de la défaillance de la machine primaire que ce soit en phase de supervision ou intervention.

Défaillance de la machine primaire en phase de supervision

$SUPINT_{LI}$ communique avec le MW du CCG, alors que MW_{LI} communique avec un ou plusieurs MW_F des zones CCLF couvertes. Quand la machine primaire tombe en panne (l'état passe en **NOK**), il faudra la remplacer par la machine secondaire et redéployer les deux entités MW et SUPINT. Les règles de reconfiguration sont décrites ci-après.

La première règle $R0_{SUB}$ constitue un cas spécifique où le MW_I est connecté à un seul MW_F. Elle permet de remplacer la machine dont l'état est NOK qui héberge les entités SUPINT et MW par la machine secondaire où ces mêmes entités sont redéployées avec les mêmes caractéristiques (identifiants, cœurs alloués et localisation) et les liens sont conservés. La généralisation de cette règle est exprimée par la règle $R1_{SUB}$ qui représente seulement le changement de machines hébergeant les entités SUPINT et MW. Les connexions sont exprimées par C qui établit un arc orienté du MW_F vers MW_I, hébergé par la nouvelle machine

secondaire pour tout arc orienté qui existait entre MW_F et MW_I, hébergé par la machine primaire en défaillance.

$R0_{SUB}$ = (L={N1(Id_{ISI}, SUPINT, $1/C_{MI}$, M_I, **NOK**, **φ_{PRM}**, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, **NOK**, **φ_{PRM}**, L_I), N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N4(Id_{MWG}, MW, C_{MG}, M_G, OK, φ_{VIR}, L_G), N2→^{O}N1, N3→^{Etiq32}N2, N1→^{Etiq14}N4};
K={N3(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), N4(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G)};
R={N5(Id_{ISI}, SUPINT, $1/C_{MISEC}$, M_{ISEC}, **OK**, **φ_{SEC}**, L_I), N6(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, **φ_{SEC}**, L_I), N6→^{O}N5, N3→^{Etiq36}N6, N5→^{Etiq54}N4};
C={ };
E={Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq14(R, λ_{14}, PR_{14}, RT_{TH14}), Etiq36(O, λ_{36}, PR_{36}, RT_{TH36}), Etiq54(R, λ_{54}, PR_{54}, RT_{TH54})};
Cond={ })

$R1_{SUB}$ = (L={N1(Id_{ISI}, SUPINT, $1/C_{MI}$, M_I, **NOK**, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, **NOK**, φ_{PRM}, L_I), N3(Id_{MWG}, MW, C_{MG}, M_G, OK, φ_{VIR}, L_G), N2→^{O}N1, N1→^{Etiq13}N3};
K={N3(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G)};
R={N4(Id_{ISI}, SUPINT, $1/C_{MISEC}$, M_{ISEC}, **OK**, φ_{SEC}, L_I), N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), N5→^{O}N4, N4→^{Etiq43}N3};
C={(N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, OK, φ_{SEC}, L_I), EtiqX/EtiqX, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in))};
E={Etiq13(R, λ_{13}, PR_{13}, RT_{TH13}), Etiq43(R, λ_{43}, PR_{43}, RT_{TH43}), EtiqX(O, λ_X, PR_X, RT_{THX})};
Cond={ })

Défaillance de la machine primaire en phase d'intervention

Lors de la phase d'intervention, le CCLI peut être soit en d'intervention intermédiaire (intI) soit en phase d'intervention globale (intG). Par conséquence, le trafic de types GC ou LC sera présent. La substitution de la machine primaire par celle secondaire en phase d'intervention est exprimée, en fonction des types d'interventions, par les règles de reconfiguration, en fonction des cas, sont les suivantes :

$R2_{SUB}$ = (L={N1(Id_{ISI}, SUPINT, $1/C_{MI}$, M_I, **NOK**, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, **NOK**, φ_{PRM}, L_I), N3(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G), N2→^{O}N1, N1→^{Etiq12}N2, N1→^{Etiq13}N3};
K={N3(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G)};
R={N4(Id_{ISI}, SUPINT, $1/C_{MISEC}$, M_{ISEC}, **OK**, φ_{SEC}, L_I), N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), N4→^{Etiq45}N5, N5→^{O}N4, N4→^{Etiq43}N3};
C={(N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), EtiqX/EtiqX, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out), (N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), EtiqY/EtiqY, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in)};
E={Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12}), Etiq13(R, λ_{13}, PR_{13}, RT_{TH13}), Etiq43(R, λ_{43}, PR_{43}, RT_{TH43}), EtiqX(LC, λ_X, PR_X, RT_{THX}), Etiq45(LC, λ_{45}, PR_{45}, RT_{TH45}), EtiqY(O, λ_Y, PR_Y, RT_{THY})} ;
Cond={ })

$R3_{SUB}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, **NOK**, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, **NOK**, φ_{PRM}, L_I), N3(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G), N2→^{O}N1, N1→^{Etiq12}N2, N1→^{Etiq13}N3, N3→^{Etiq31}N1};
K={N3(Id_{MWG}, MW, C_{MG}/C_{MG}, M_G, OK, φ_{VIR}, L_G)};
R={N4(Id_{ISI}, SUPINT, 1/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, **OK**, φ_{SEC}, L_I), N4→^{Etiq45}N5, N5→^{O}N4, N4→^{Etiq43}N3, N3→^{Etiq34}N4};
C={(N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, OK, φ_{SEC}, L_I), EtiqX/EtiqX, N6(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in), (N5(Id_{MWI}, MW, C_{MWI}/C_{MISEC}, M_{ISEC}, OK, φ_{SEC}, L_I), EtiqY/EtiqY, N6(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out)};
E={Etiq12(GC, λ_{12}, PR_{12}, RT_{TH12}), Etiq13(R, λ_{13}, PR_{13}, RT_{TH13}), Etiq31(GC, λ_{31}, PR_{31}, RT_{TH31}), Etiq34(GC, λ_{34}, PR_{34}, RT_{TH34}), Etiq43(R, λ_{43}, PR_{43}, RT_{TH43}), Etiq45(GC, λ_{45}, PR_{45}, RT_{TH45}), EtiqX(O, λ_X, PR_X, RT_{THX}), EtiqY(GC, λ_Y, PR_Y, RT_{THY})} ;
Cond={ })

La règle $R2_{SUB}$ permet de remplacer une machine primaire défaillante en zone L_I, où l'activité est en phase d'intervention intermédiaire (intI), par la machine secondaire. L'établissement de l'ensemble des connexions avec les MW en zone L_{Fx} qui reçoivent les commandes locales (LC) est assuré par C. La règle $R3_{SUB}$ fait la même opération mais pour le MW en zone L_I en phase d'intervention générale (intG) avec des commandes générales (GC) à destination de certains MW de zone CCLF.

5.6.3. Gestion des exigences métier en QoS au niveau opératoire

Les exigences de niveau métier en QoS doivent être respectées par le MW de la zone concernée. Dans cette section, nous allons nous focaliser sur l'adaptation du nombre de cœurs alloués au MW ainsi que la duplication et la répartition de charge, en donnant les règles correspondantes. La dynamicité des besoins peut être justifiée par exemple par l'apparition d'un nouveau trafic émanant d'une rame qui vient d'entrer en station. Des conditions basées sur le modèle analytique vont guider cette adaptation et restreindre l'applicabilité de la règle. Pour des raisons de simplification, le MW_G et les interconnexions avec lui ne seront pas présentés ici.

Gestion du nombre de cœurs du Middleware

Une première action d'adaptation orientée ressources consiste à modifier le nombre de cœurs alloués au MW afin de satisfaire aux besoins en QoS. Cette configuration (augmentation / diminution) est tirée du principe d'élasticité afin de ne pas avoir une surconsommation de ressources et d'énergie nécessaire au fonctionnement de la machine hébergeant les entités MW et SUPINT.

Les règles $R1_{QoS}$ et $R2_{QoS}$ permettent respectivement d'augmenter et diminuer le nombre de cœurs alloués au MW_I de la zone L_I. Elles considèrent et conservent toutes les connexions auxquelles pourrait être attaché le MW pour les différentes phases d'exécution (sup, intI ou intG). Les conditions d'applicabilité de la règle pour le cas d'augmentation stipulent que le nombre de cœurs actuels est insuffisant pour le traitement du trafic entrant et que la machine dispose d'un nombre suffisant de cœurs (au moins un libre). La diminution du nombre de cœurs alloués au MW peut être déclenchée si ce dernier est surdimensionné par rapport à la satisfaction des besoins du trafic entrant, et si que le nombre de cœurs qui lui sont alloués est au moins égal à 1 lors de son démarrage.

La dernière règle (R_{REDQoS}) de redimensionnement est générique. Elle permet de déterminer le nombre de cœurs minimal et nécessaire à déployer. Elle se traduit soit par l'activation de nouveaux cœurs dans la limite de ceux disponibles, ou la désactivation de cœurs.

$R1_{QoS}$ = (L={N1(Id_{MWI}, MW, C_{MWIY}/C_{MIY}, M_{IY}, OK, φ_Y, L_I)};
K={ };
R = {N2(Id_{MWI}, MW, C_{MWIY} **+ 1**/C_{MIY}, M_{IY}, OK, φ_Y, L_I)};
C={(N2, EtiqO/EtiqO, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in),
(N2, O/O, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), out, out),
(N2, EtiqLC/EtiqLC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqLC/EtiqLC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out),
(N2, EtiqGC/EtiqGC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqGC/EtiqGC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out)};
E={EtiqO(O, λ_O, PR_O, RT_{THO}), EtiqLC(LC, λ_{LC}, PR_{LC}, RT_{THLC}), EtiqGC(GC, λ_{GC}, PR_{GC}, RT_{THGC})};
Cond={$C_{MWIY} < FA'(\lambda_{SUM}, \min(RT_{THk}))$ && $C_{MWIY} + 1 \leq C_{MIY} - 1$})

$R2_{QoS}$ = (L={N1(Id_{MWI}, MW, C_{MWIY}/C_{MIY}, M_Y, OK, φ_Y, L_I)};
K={ };
R = {N2($Id_{MW,}$ MW, C_{MWIY} **- 1**/C_{MIY}, M_{IY}, OK, φ_Y, L_I)};
C={(N2, EtiqO/EtiqO, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in),
(N2, O/O, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), out, out),
(N2, EtiqLC/EtiqLC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqLC/EtiqLC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out),
(N2, EtiqGC/EtiqGC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqGC/EtiqGC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out)};
E={EtiqO(O, λ_O, PR_O, RT_{THO}), EtiqLC(LC, λ_{LC}, PR_{LC}, RT_{THLC}), EtiqGC(GC, λ_{GC}, PR_{GC}, RT_{THGC})};
Cond={$C_{MWIY} > FA'(\lambda_{SUM}, \min(RT_{THk}))$ && $C_{MWIY} - 1 \geq 0$})

Y = PRM ou SEC

R_{REDQoS} = (L={N1(Id_{MWI}, MW, C_{MWIY}/C_{MIY}, M_Y, OK, φ_Y, L_I)};
K={ };
R = {N2($Id_{MW,}$ MW, **$FA'(\lambda_{SUM}, \min(RT_{THk}))$**/ C_{MIY}, M_{IY}, OK, φ_Y, L_I)};
C={(N2, EtiqO/EtiqO, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), in, in),
(N2, O/O, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), out, out),
(N2, EtiqLC/EtiqLC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqLC/EtiqLC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out),
(N2, EtiqGC/EtiqGC, N(Id_{ISI}, SUPINT, $1/C_{MIY}$, M_{IY}, OK, φ_Y, L_I), in, in),
(N2, EtiqGC/EtiqGC, N(Id_{MWF}, MW, C_{MWF}/C_{MF}, M_F, OK, φ_U, L_F), out, out)};
E={EtiqO(O, λ_O, PR_O, RT_{THO}), EtiqLC(LC, λ_{LC}, PR_{LC}, RT_{THLC}), EtiqGC(GC, λ_{GC}, PR_{GC}, RT_{THGC})};
Cond={($C_{MWIY} < FA'(\lambda_{SUM}, \min(RT_{THk}))$ || $C_{MWIY} > FA'(\lambda_{SUM}, \min(RT_{THk}))$) && $0 \leq FA'(\lambda_{SUM}, \min(RT_{THk}) \leq C_{MIY} - 1$})

Y = PRM ou SEC

La Figure 5.10 illustre, à haut niveau, le résultat de l'application de cette règle sur l'entité Middleware (N2) en cas d'augmentation par i (tel que : $C_{MWIY} + i = FA'(\lambda_{SUM}, \min(RT_{THk}))$ du nombre de cœurs parmi ceux disponibles.

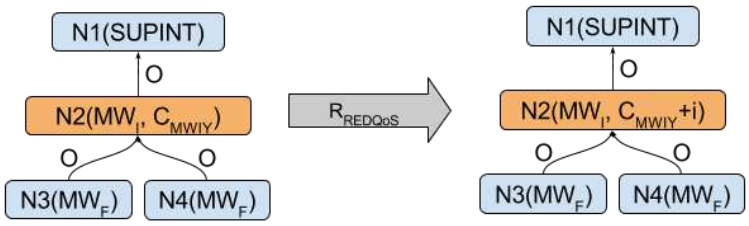

Figure 5.10 - Vue de haut niveau de l'application de la règle R_{REDQoS}

Gestion de la charge entre les entités MW

En cas de non-respect des contraintes de QoS, un protocole de gestion de charge en deux temps est enclenché. Dans un premier temps, la machine secondaire est démarrée et la charge est répartie entre les machines primaire et secondaire. Cette répartition prend en considération les contraintes en QoS de chaque trafic. Dans un deuxième temps, si cette répartition n'est plus suffisante pour respecter les contraintes en QoS du trafic, la/les machine(s) voisine(s) va/vont également traiter une partie de la charge. Dans ce cas, la répartition de charge prend en considération le paramètre de priorité ; le trafic le moins prioritaire est traité par les voisins, considérés plus éloignés sur le réseau de communications. Dans notre cas, nous allons considérer un nombre N fixe de MW_F attachés initialement au CCLI initial. Pour tout N, la transposition est triviale. Dans cette section, nous ne considérons les entités MW en zone CCLI qu'en phase de supervision (sup) et phase d'intervention intermédiaire (intI).

Duplication et répartition de charge avec la machine dupliquée

Pour les différentes phases d'exécution, les règles de duplication et répartition de charge vont prendre en considération, à titre d'illustration, le cas où le MW de la zone CCLI est interconnecté initialement avec quatre MW de zones CCLF différentes.

Phase de supervision

En phase de supervision, la règle $R3_{SQoS}$ assure le déploiement de la machine secondaire et l'instanciation d'un nouveau MW de répartition de charge. Chaque MW (MW_I et MW_{ISEC}) va gérer une partie de la charge. L'entité SUPINT va rester déployée sur la machine primaire et va recevoir les observations de la part des quatre entités MW.

$R3_{SQoS}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N3→^{Etiq32}N2, N4→^{Etiq42}N2, N5(Id_{MWF3}, MW, C_{MWF3}/C_{MF3}, M_{F3}, OK, φ_U, L_{F3}), N6(Id_{MWF4}, MW, C_{MWF4}/C_{MF4}, M_{F4}, OK, φ_U, L_{F4}), N5→^{Etiq52}N2, N6→^{Etiq62}N2};
K={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N3→^{Etiq32}N2, N4→^{Etiq42}N2};
R={N7(Id_{MWISEC}, MW, C_{MWISEC}/C_{MISEC}, M_{ISEC}, OK, φ_{SEC}, L_I), N7→^{O}N1, N5→^{Etiq57}N7, N6→^{Etiq67}N7};
C={ };
E={Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq42(O, λ_{42}, PR_{42}, RT_{TH42}), Etiq52(O, λ_{52}, PR_{52}, RT_{TH52}), Etiq62(O, λ_{62}, PR_{62}, RT_{TH62}), Etiq57(O, λ_{57}, PR_{57}, RT_{TH57}), Etiq67(O, λ_{67}, PR_{67}, RT_{TH67})};
Cond={[C_{MI} - 1 < FA'(λ_{32}+λ_{42}+λ_{52}+λ_{62}, min(RT_{TH32},RT_{TH42}, RT_{TH52},RT_{TH62}))] && [C_{MI} - 1 ≥ FA'(λ_{32}+λ_{42}, min(RT_{TH32},RT_{TH42})) && C_{MISEC} ≥ FA'(λ_{52}+λ_{62}, min(RT_{TH52},RT_{TH62}))] && [(PR_{32} ≥ PR_{52} && PR_{42} ≥ PR_{62}) || (PR_{32} ≥ PR_{62} && PR_{42} ≥ PR_{52})]})

Vu que la gestion des cœurs est assurée par les règles d'augmentation ($R1_{QoS}$) et de diminution ($R2_{QoS}$), la condition d'applicabilité raisonne seulement sur le nombre de cœurs dont dispose chaque machine. Le premier bloc de conditions vérifie que le nombre de cœurs total qui peuvent être alloués au MW_{CMI} ne peut pas respecter la QoS du trafic entrant. Le deuxième vérifie le fait qu'en faisant la répartition entre la machine primaire et la machine secondaire la QoS des MW_F est respectée. Alors que le dernier bloc concerne les priorités qui consiste à interconnecter le MW_I avec les MW_F les plus prioritaires (en terme de délai). En effet, le passage par l'entité MW_{ISEC} va induire un temps de transit supplémentaire pour arriver jusqu'à l'entité SUPINT, puisqu'elle est déployée sur une machine distante. La Figure 5.11 illustre, à haut niveau, le résultat de l'application de la règle.

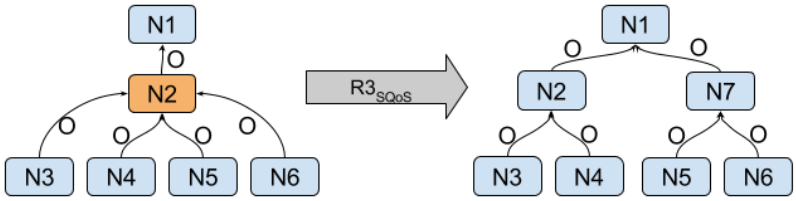

Figure 5.11 - Vue de haut niveau de l'application de la règle $R3_{SQoS}$

Phase d'intervention intermédiaire (intI)

En phase d'intervention intermédiaire (intI), la règle $R3_{IQoS}$ assure le déploiement de la machine secondaire et l'instanciation d'un nouveau MW de répartition de charge. Dans cette phase, il y aura un nouveau trafic (LC) de commande qui doit être géré par les entités MW.

$R3_{IQoS}$ = (L={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N1→^{Etiq12}N2, N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N3→^{Etiq32}N2, N4→^{Etiq42}N2, N2→^{Etiq23}N3, N2→^{Etiq24}N4, N5(Id_{MWF3}, MW, C_{MWF3}/C_{MF3}, M_{F3}, OK, φ_U, L_{F3}), N6(Id_{MWF4}, MW, C_{MWF4}/C_{MF4}, M_{F4}, OK, φ_U, L_{F4}), N5→^{Etiq52}N2, N6→^{Etiq62}N2, N2→^{Etiq25}N5, N2→^{Etiq26}6};
K={N1(Id_{ISI}, SUPINT, 1/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2(Id_{MWI}, MW, C_{MWI}/C_{MI}, M_I, OK, φ_{PRM}, L_I), N2→^{O}N1, N1→^{Etiq12}N2, N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N3→^{Etiq32}N2, N4→^{Etiq42}N2, N2→^{Etiq23}N3, N2→^{Etiq24}N4};
R={N7(Id_{MWISEC}, MW, C_{MWISEC}/C_{MISEC}, M_{ISEC}, OK, φ_{SEC}, L_I), N7→^{O}N1, N1→^{Etiq17}N7, N5→^{Etiq57}N7, N6→^{Etiq67}N7, N7→^{Etiq75}N5, N7→^{Etiq76}N6};
C={ };
E={Etiq12(LC, λ_{12}, PR_{12}, RT_{TH12}), Etiq17(LC, λ_{17}, PR_{17}, RT_{TH17}), Etiq23(LC, λ_{23}, PR_{23}, RT_{TH23}), Etiq24(LC, λ_{24}, PR_{24}, RT_{TH24}), Etiq25(LC, λ_{25}, PR_{25}, RT_{TH25}), Etiq26(LC, λ_{26}, PR_{26}, RT_{TH26}), Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq42(O, λ_{42}, PR_{42}, RT_{TH42}), Etiq52(O, λ_{52}, PR_{52}, RT_{TH52}), Etiq62(O, λ_{62}, PR_{62}, RT_{TH62}), Etiq57(O, λ_{57}, PR_{57}, RT_{TH57}), Etiq67(O, λ_{67}, PR_{67}, RT_{TH67}), Etiq75(LC, λ_{75}, PR_{75}, RT_{TH75}), Etiq76(O, λ_{76}, PR_{76}, RT_{TH76})};
Cond={[C_{MWI} = C_{MI} -1 && C_{MWI} < FA'(λ_{12}+λ_{32}+λ_{42}+λ_{52}+λ_{62}, min(RT_{TH12},RT_{TH32},RT_{TH42}, RT_{TH52},RT_{TH62}))] && [C_{MI} - 1 ≥ FA'(λ_{12}+λ_{32}+λ_{42}, min(RT_{TH12},RT_{TH32},RT_{TH42})) && C_{MISEC} ≥ FA'(λ_{17}+λ_{52}+λ_{62}, min(RT_{TH17},RT_{TH52},RT_{TH62})] && [(PR_{23} ≥ PR_{25} && PR_{24} ≥ PR_{26}) || (PR_{23} ≥ PR_{26} && PR_{24} ≥ PR_{25})]})

La règle est guidée par trois blocs de conditions d'applicabilité. Le premier bloc concerne l'incapacité du MW à traiter le trafic en respectant son besoin en QoS, même en lui allouant l'ensemble des cœurs disponibles. Le deuxième sur les performances des machines primaire et secondaire pour l'accueil du trafic entrant tout en respectant les contraintes en QoS. Alors que le dernier bloc tient compte des priorités. Ici, seules les priorités liées aux commandes sont

considérées et permettent d'interconnecter le MW_I avec les MW_F dont les priorités sont les plus élevées.

Duplication et répartition de charge avec la machine dupliquée et la machine de la zone voisine

Si les machines primaire et secondaire ne sont pas capables de supporter la charge, la répartition s'étend alors aux machines des localisations voisines. Dans les règles qui suivent, le nombre de MW_F attachés initialement au MW_I (avant l'application de la règle de reconfiguration) est de l'ordre de quatre. Ici, à titre illustratif, nous ne considérons que la phase de supervision, étant donné que la réalisation des règles pour les autres phases suit la même approche.

$R4_{VQoS}$ = (L={N1(Id_{ISIi}, SUPINT, 1/C_{MI}, M_{Ii}, OK, φ_{PRM}, L_{Ii}), N2(Id_{MWIi}, MW, C_{MWIi}/C_{MIi}, M_{Ii}, OK, φ_{PRM}, L_{Ii}), N3($Id_{MWISECi}$, MW, $C_{MWISECi}$/C_{MISECi}, M_{ISECi}, OK, φ_{SEC}, L_{Ii}), N4(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N5(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N6(Id_{MWF3}, MW, C_{MWF3}/C_{MF3}, M_{F3}, OK, φ_U, L_{F3}), N7(Id_{MWF4}, MW, C_{MWF4}/C_{MF4}, M_{F4}, OK, φ_U, L_{F4}), N2→^{O}N1, N4→^{Etiq42}N2, N5→^{Etiq52}N2, N3→^{O}N1, N6→^{Etiq63}N3, N7→^{Etiq73}N3, N8(Id_{MWIj}, MW, C_{MWIj}/C_{MIj}, M_{Ij}, OK, φ_{PRM}, L_{Ij})};
K={N1(Id_{ISIi}, SUPINT, 1/C_{MI}, M_{Ii}, OK, φ_{PRM}, L_{Ii}), N2(Id_{MWIi}, MW, C_{MWIi}/C_{MIi}, M_{Ii}, OK, φ_{PRM}, L_{Ii}), N3($Id_{MWISECi}$, MW, $C_{MWISECi}$/C_{MISECi}, M_{ISECi}, OK, φ_{SEC}, L_{Ii}), N4(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N5(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N6(Id_{MWF3}, MW, C_{MWF3}/C_{MF3}, M_{F3}, OK, φ_U, L_{F3}), N7(Id_{MWF4}, MW, C_{MWF4}/C_{MF4}, M_{F4}, OK, φ_U, L_{F4}), N2→^{O}N1, N4→^{Etiq42}N2, N3→^{O}N1, N6→^{Etiq63}N3, N7→^{Etiq73}N3, N8(Id_{MWIj}, MW, C_{MWIj}/C_{MIj}, M_{Ij}, OK, φ_{PRM}, L_{Ij})};
R={N5→^{Etiq58}N8, N8→^{O}N1};
C={ };
E={Etiq42(O, λ_{42}, PR_{42}, RT_{TH42}), Etiq52(O, λ_{52}, PR_{52}, RT_{TH52}), Etiq63(O, λ_{63}, PR_{63}, RT_{TH63}), Etiq73(O, λ_{73}, PR_{73}, RT_{TH73}), Etiq58(O, λ_{58}, PR_{58}, RT_{TH58})};
Cond={[C_{MWIi} = C_{MIi} - 1 && C_{MWIi} < FA'(λ_{42}+λ_{52}, min (RT_{TH42}, RT_{TH52})] && [C_{MWIi} ≥ FA'(λ_{42}, RT_{TH42}) && $C_{MWISECi}$ = C_{MISECi} && $C_{MWISECi}$ ≥ FA'(λ_{63}+ λ_{73}, min(RT_{TH63}, RT_{TH73})) && [C_{MIj} ≥ FA'(λ_{52}+sum(λ_{k8}), min(RT_{TH52}, RT_{THk8})) && PR_{42} ≥ PR_{52}]})

avec : j = i-1 ou i+1

La règle $R4_{VQoS}$ traite le cas où la machine primaire (avec deux MW_F attachés au MW_I) est en marche mais qu'elle ne peut plus traiter le trafic qui arrive (premier bloc de conditions) même en répartissant la charge avec la machine secondaire (deuxième bloc de conditions). Dans ce cas, la répartition de charge se fait avec le MW (MW_{Ii-1}/MW_{Ii+1}) du voisin (V_{i-1}/V_{i+1}). Dans ce cas le MW_I s'attache à un seul MW_F, le MW_{ISEC} s'attache à deux MW_F et MW_{Ii+1} / MW_{Ii+1} s'attache à un MW_F supplémentaire (en plus de ce qui lui est déjà attaché). Ensuite, chaque MW intervenant dans cette répartition se charge d'orienter le trafic d'observation vers l'entité SUPINT hébergée par la machine primaire. La condition d'applicabilité de cette règle se base sur le fait que le nombre de cœurs alloués aux entités MW_I et MW_{ISEC} atteignent le maximum disponible, que le MW_I et MW_{ISEC} et MW_{Ii-1}/MW_{Ii+1} puisse chacun accueillir le trafic du MW_F qui lui est attribué. Le choix de l'attachement de l'un des MWF attachés initialement au MW_I se fait sur la base de leurs priorités ; le plus prioritaire reste attaché au MW_I vu sa proximité avec l'entité SUPINT.

La Figure 5.12 illustre, à haut niveau, le résultat de l'application de la règle pour les données d'observation. Suite à l'application de cette règle, le voisin N8 (de type MW) prend en charge une partie du trafic en l'interconnectant avec le nœud N5, permettant ainsi de réduire la charge du nœud N2. Le trafic est ensuite redirigé vers N1 (SUPINT) responsable de sa collecte et de son analyse.

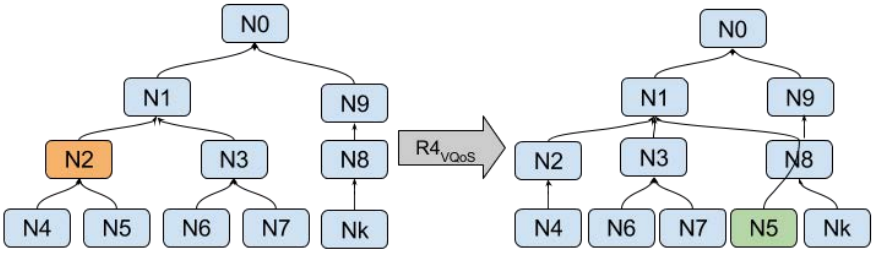

Figure 5.12 - Vue de haut niveau de l'application de la règle $R4_{VQoS}$

En phase de supervision, une autre reconfiguration est considérée. Elle traite le cas où la duplication n'est pas possible (par exemple, défaillance de la machine primaire ou secondaire de duplication) et où le Middleware de la machine utilisée n'est plus capable de satisfaire aux besoins en QoS. Dans ce cas, la répartition de charge ($R5_{VQoS}$) va se faire avec l'un des voisins (V_{i-1}/V_{i+1}) de la zone couverte. A titre illustratif, nous ne considérons ici que deux MW_F attachés au MW_I déployé sur la machine secondaire.

$R5_{VQoS}$ = (L={N1(Id_{ISIi}, SUPINT, $1/C_{MIi}$, M_{Ii}, OK, φ_X, L_{Ii}), N2(Id_{MWIi}, MW, C_{MWIi}/C_{MIi}, M_{Ii}, OK, φ_X, L_{Ii}), N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N2$\rightarrow^{O}$N1, N3$\rightarrow^{Etiq32}$N2, N4$\rightarrow^{Etiq42}$N2, N5(Id_{MWIj}, MW, C_{MWIj}/C_{MIj}, M_{Ij}, OK, φ_{PRM}, L_{Ij})};
K={N1(Id_{ISIi}, SUPINT, $1/C_{MIi}$, M_{Ii}, OK, φ_X, L_{Ii}), N2(Id_{MWIi}, MW, C_{MWIi}/C_{MIi}, M_{Ii}, OK, φ_X, L_{Ii}), N3(Id_{MWF1}, MW, C_{MWF1}/C_{MF1}, M_{F1}, OK, φ_U, L_{F1}), N4(Id_{MWF2}, MW, C_{MWF2}/C_{MF2}, M_{F2}, OK, φ_U, L_{F2}), N2$\rightarrow^{O}$N1, N3$\rightarrow^{Etiq32}$N2, N5(Id_{MWIj}, MW, C_{MWIj}/C_{MIj}, M_{Ij}, OK, φ_{PRM}, L_{Ij})};
R={N4$\rightarrow^{Etiq45}$N5, N5$\rightarrow^{O}$N1};
C={ };
E={Etiq32(O, λ_{32}, PR_{32}, RT_{TH32}), Etiq42(O, λ_{42}, PR_{42}, RT_{TH42}), Etiq45(O, λ_{45}, PR_{45}, RT_{TH45})};
Cond={[C_{MWIi} = C_{MIi} - 1 && C_{MWIi} < FA'(λ_{32}+λ_{42}, min(RT_{TH32}, RT_{TH42}))] && [C_{MWIi} ≥ FA'(λ_{32}, RT_{TH32}) && C_{MIj} ≥ FA'(λ_{42}+sum(λ_{k5}), min(RT_{TH42}, RT_{THk5}))]})

avec : j = i-1 ou i+1

5.6.4. Règles de reconfiguration dans le processus de planification

L'ensemble des règles présentées précédemment alimente la base de connaissance du gestionnaire autonomique. Elles sont utilisées par le composant de planification afin de guider sa gestion. Cette gestion peut être basée sur un protocole de gestion exécutant d'une manière séquentielle ces règles. En prenant en considération l'ensemble des actions décrites précédemment, nous pouvons considérer à titre illustratif le séquencement d'actions proposé par la Figure 5.13.

Ce séquencement peut être décrit sous forme de protocole de planification (PR_M) du MW en zone CCLI pour la gestion de la QoS.

PR_M = (R_{REDQoS}(PRM)* ; RX_{SUB}(SEC) ; R_{REDQoS}(SEC)* ; ($R4x_{QoS}$(Vx) || $R5x_{QoS}$(Vx)) ; R_{REDQoS}(Vx)*) || ($R3_{xQoS}$(SEC) ; R_{REDQoS}(SEC)* ; ($R4x_{QoS}$(Vx) || $R5x_{QoS}$(Vx)) ; R_{REDQoS}(Vx)*)

Où :

- ***R_1 ; R_2 :*** *exécution de la règle R_1 ensuite la règle R_2*
- ***R_1 || R_2 :*** *exécution de la règle R_1 ou la règle R_2*

- ***R1* :*** *exécution un ou plusieurs fois la règle R_1*

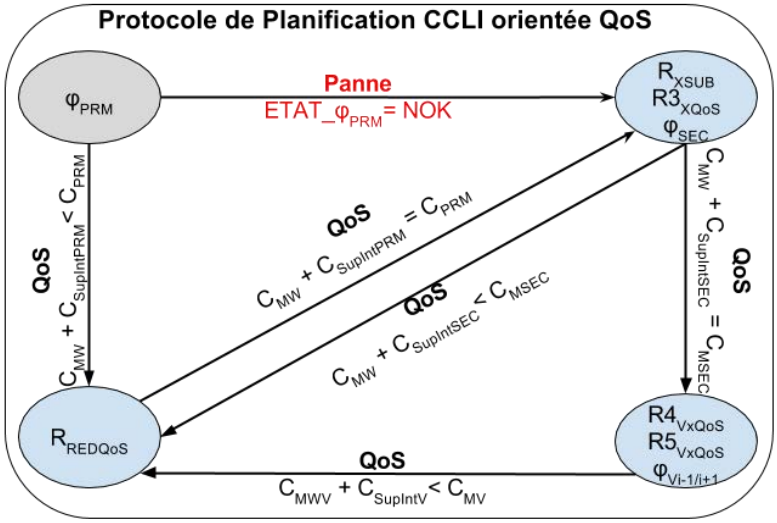

Figure 5.13 - Protocole de planification en zone CCLI

Ce protocole assume un état initial avec une machine primaire (φ_{PRM}) non dupliquée.

Si cette machine tombe en défaillance (Etat = NOK), elle va être remplacée directement par la machine secondaire de duplication (φ_{SEC}) grâce à la règle RX_{SUB}(SEC) ($R1_{SUB}$ si c'est en phase de supervision, $R2_{SUB}$ en phase d'intervention intermédiaire et $R3_{SUB}$ en phase d'intervention globale). Sinon, si la machine primaire reste opérationnelle et qu'elle n'est plus capable de respecter les contraintes en QoS, le nombre de cœurs alloués au MW sera augmenté parmi l'ensemble des cœurs disponibles (R_{REDQoS}(PRM)). Après, le MW_{ISEC} de duplication sera déployé sur la machine secondaire pour initier la répartition de charge ($R3_{xQoS}$(SEC)).

Si la machine secondaire n'est pas capable de respecter les contraintes en QoS, elle se verra initialement augmenter le nombre de cœurs parmi ceux disponibles (R_{REDQoS}(SEC)). Finalement, si la machine secondaire atteint également ses limites, le planificateur va faire appel, comme dernier recours, à la machine de zone voisine (φ_{Vi-1} et/ou φ_{Vi+1}) pour faire la répartition de charge ($R4_{VxQoS}$(Vx) ou $R5_{VxQoS}$(Vx)). Si nécessaire, il établira finalement le processus d'augmentation de cœurs alloués (R_{REDQoS}(Vx)) à son MW (MW_{Vi-1} et/ou MW_{Vi+1}).

Les règles de ce protocole peuvent être exécutées par le planificateur les unes après les autres via le moteur de graphes. Dès qu'il y a une correspondance, le planificateur considère la règle correspondante comme étant son plan qu'il envoie composant d'exécution de la boucle autonomique. En même temps, le moteur de graphes met à jour l'instance d'architecture logique stockée au niveau de la base de connaissance afin de correspondre avec celle réelle.

5.7. CONCLUSION

Dans ce chapitre, nous nous sommes intéressé à la représentation de l'architecture gérée. Nous partons d'un cas d'étude de gestion de la foule en transport urbain en cas de problème (incident ou danger) avec considération d'une phase de supervision et une phase d'intervention. Les entités et interactions du cas d'étude sont ensuite modélisées en deux niveaux d'abstraction. Le niveau métier fait apparaître les entités et les interactions de haut niveau sous l'angle des rôles métier. Alors que le niveau opératoire est un raffinement du niveau métier qui fait apparaître les applications logicielles impliquées et les entités Middleware nécessaires à leurs interactions. La spécification de chaque niveau est caractérisée par un style architectural décrit via le

formalisme des graphes et des règles de réécriture. Cette modélisation représente les entités sous forme de nœuds multi-libellés et les interactions sous forme d'arcs multi-étiquetés.

Ensuite, nous proposons une approche de planification multi-modèles au niveau opératoire pour la gestion locale de la QoS. Elle couple le modèle à base de règles de réécriture de graphes avec le modèle analytique de l'entité Middleware (chapitre 4). Les règles sont proposées pour l'entité Middleware au niveau de la zone intermédiaire (CCLI) et permettent de guider la planification. Elles traitent la défaillance de la machine ou la dégradation de la QoS en augmentant le nombre de cœurs dédiés à l'entité Middleware ou en faisant une machine secondaire dédiée ou l'entité Middleware voisine. Le modèle analytique est utilisé dans ce cas comme condition d'applicabilité de la règle afin de déterminer si la reconfiguration respectera les contraintes en QoS du trafic entrant. Il est utilisé aussi dans certains cas pour déterminer la configuration des ressources de l'entité Middleware (en termes de nombre de cœurs). Enfin, nous avons donné un protocole de planification, basé sur les règles de reconfiguration, qui peut guider le comportement du composant de planification de l'AMS (AMSPlanner).

Conclusion générale et perspectives

L'Internet des objets est amené à voir se développer de plus en plus d'applications dites *applications IoT*, dont les interactions avec les objets communicants (plusieurs milliards à l'horizon 2020) nécessiteront l'usage de *Middleware* de communication. Selon le domaine métier dont elles relèveront, ces applications présenteront des besoins en QoS pouvant s'exprimer en termes de temps de réponse, de disponibilité, de fiabilité, etc. Il est ainsi impératif d'anticiper la prise en compte de ces besoins par le système de communication sous-jacent aux applications IoT, notamment au niveau Middleware, amené à devenir un réel goulot d'étranglement vis-à-vis de la satisfaction des besoins en QoS.

Les travaux de thèse présentés dans ce mémoire s'inscrivent dans ce contexte général. Nous fournissons ci-après un résumé de nos contributions, chapitre par chapitre, ainsi que les perspectives associées.

SYNTHÈSE DES CONTRIBUTIONS

L'état de l'art présenté au ***chapitre 1*** *a permis de soulever les limites des solutions existantes au niveau Middleware et de positionner notre approche de solution pour un Middleware orienté QoS dans l'IoT.*

La majorité des solutions dédiées à l'IoT se cantonnent à relayer ou au mieux à traduire les besoins applicatifs auprès des niveaux sous-jacents (réseau et équipements), eux-mêmes chargés d'en assurer la satisfaction. Ces solutions ne prennent ainsi pas en considération l'impact du Middleware qui, en fonction de ses capacités et de la charge de requêtes qu'il a à traiter, peut cependant provoquer une dégradation importante de la QoS de bout en bout. Plus largement, les solutions QoS proposées au niveau Middleware dans d'autres contextes souffrent en majorité d'une incapacité à prendre en compte dynamiquement les besoins en QoS. Cette propriété est particulièrement nécessaire dans un environnement IoT où le déploiement, la configuration, les besoins ou encore la charge des entités (équipements, réseaux, applications) peuvent évoluer durant l'exécution du système.

Notre approche de solution pour un Middleware orienté QoS dans l'IoT s'inscrit en continuité des travaux de standardisation ayant conduit aux spécifications des Middleware SmartM2M et oneM2M. L'optique est d'étendre ces solutions en les dotant de fonctionnalités supplémentaires pour la gestion de la QoS, ceci en assurant deux propriétés : la *dynamicité* et l'*autonomie*. La première propriété conduit à ce que la réalisation des actions orientées QoS se fasse sans rupture du service fourni aux utilisateurs finaux. La deuxième propriété vise à doter le système d'une capacité d'autogestion dans ses choix d'actions afin de satisfaire les besoins en QoS des applications ; le paradigme de l'*Autonomic Computing* défini par IBM a été retenu à cette fin. Pour aboutir à la mise en œuvre d'une QoS de bout en bout, nous avons également opté pour une approche de gestion basée sur le principe des *politiques hiérarchiques*, rejoignant en cela certaines approches de gestion de réseaux, notamment dans un contexte de SDN. Deux niveaux d'actions ont ainsi été considérés : un niveau *stratégique* conduisant à une répartition adaptative

de l'objectif de QoS de bout en bout en objectifs de QoS locale à satisfaire par chacune des entités Middleware impliquées dans le chemin emprunté par les données applicatives ; un niveau *opérationnel* conduisant à l'adaptation du comportement ou des ressources des entités Middleware pour atteindre l'objectif de QoS locale fixé par le niveau stratégique.

Dans le cadre de l'approche définie au chapitre 1, le ***chapitre 2*** *a été dédié à la proposition de mécanismes opérationnels pour la gestion de la QoS, divisés en deux types : orientés trafic et orientés ressources.*

Les mécanismes orientés trafic sont inspirés des approches traditionnelles proposées dans l'Internet pour la gestion de la QoS au niveau des couches Réseau (IP) et Transport (TCP, UDP, etc.). Ils se basent sur un principe de priorités accordées au trafic pour guider le traitement des requêtes applicatives de façon différenciée. Deux principaux composants ont été proposés en amont des entités Middleware pour intégrer ce type de gestion. Un composant de classification et de marquage (CCM) permet de classer et de marquer le trafic avec une priorité donnée en fonction d'un certain nombre de critères (source du trafic, destination, type de ressource, etc.). Un composant proxy orienté priorité (POP) gère le trafic en fonction de la priorité attribuée ; il permet de réaliser trois fonctionnalités : le rejet, le retardement et/ou l'ordonnancement avant la redirection du trafic vers l'entité Middleware.

Les mécanismes orientés ressources sont inspirés des mécanismes utilisés pour la gestion de la résistance au facteur d'échelle dans des environnements de virtualisation tels que le cloud. Ces mécanismes reposent sur deux approches. La première, dite *verticale,* cible une gestion des ressources (CPU, RAM, etc.) associées à chaque entité Middleware (aux niveaux machine physique et/ou virtuelle, JVM, etc.) en fonction de la charge courante et du niveau de performance à atteindre. La deuxième approche, dite *horizontale*, est basée sur un modèle de distribution et/ou de duplication de certains composants Middleware conduisant à en répartir la charge sur plusieurs instances d'entités Middleware. La validation de chaque type de mécanismes a été réalisée via une plateforme d'émulation permettant de générer le trafic vers l'entité Middleware ciblée en passant par les composants de gestion de la QoS.

Dans le ***chapitre 3****, nous avons spécifié et conçu l'architecture d'un système de niveau Middleware pour l'IoT (IoT-Q) qui étend les standards SmartM2M et oneM2M en y intégrant des fonctionnalités de gestion de la QoS.*

En plus des services traditionnels fournis par SmartM2M ou oneM2M, IoT-Q permet aux acteurs de bénéficier de services supplémentaires permettant la prise en compte des besoins en QoS des applications. L'architecture proposée inclut la boucle de gestion (MAPE-K) du paradigme de l'Autonomic Computing. L'approche d'élaboration suivie instancie également le concept de politiques hiérarchiques. Elle considère deux niveaux de gestion dédiés chacun à un niveau d'actions (stratégiques et opérationnelles). Au niveau opérationnel, la gestion de chaque entité Middleware en vue de satisfaire un objectif de QoS locale repose sur l'utilisation d'un AMS (*Autonomic Manager Slave*). Chaque AMS prend en considération les caractéristiques de l'entité Middleware (environnement de déploiement, performances, etc.) afin de configurer les mécanismes orientés trafic et/ou orientés ressources les plus appropriés. La gestion de la cohérence des objectifs locaux vis-à-vis d'un objectif de QoS de bout en bout est assuré par un composant de plus haut niveau, l'AMM (*Autonomic Manager Master*). Ce composant est responsable de la prise en considération du besoin en QoS de bout en bout, de sa déclinaison en un ensemble de besoins locaux exprimés auprès des entités Middleware impliquées dans le chemin de données, et enfin de la supervision du respect de cet objectif global et de la planification de nouveaux objectifs de QoS local en cas de non-respect de l'objectif global.

*Dans les **chapitre 4 et 5,** nos contributions ont porté sur l'aspect comportemental de l'architecture IoT-Q, plus spécifiquement au niveau des composants de l'AMS. Les contributions principales qui en résultent portent sur la proposition de modèles sur lesquels appuyer l'algorithmique des composants de monitoring et de planification de l'AMS.*

Au **chapitre 4**, nous avons ainsi proposé et validé un modèle analytique, basé sur la théorie des files d'attente, de la plateforme Middleware OM2M (suivant le standard SmartM2M). Ce modèle permet de déduire un certain nombre de métriques de performances tels que le temps de réponse d'une entité Middleware dans le traitement d'une requête. Ce modèle a ensuite été appliqué à la phase de supervision du système via le composant de monitoring de l'AMS suivant deux approches : une première approche, *réactive*, visant à lever des symptômes lorsque la QoS locale se dégrade, et une deuxième approche, *proactive*, visant à prédire la dégradation de la QoS. Cette deuxième approche se base sur la proposition d'un second modèle de prédiction basé sur le modèle ARMA et son utilisation conjointe avec le modèle analytique. Nous avons également démontré que l'application de tels modèles permettait de réduire significativement le coût du monitoring sur l'entité Middleware supervisée en termes d'impact sur la QoS et de consommation additionnelle des ressources informatiques.

Finalement, au **chapitre 5**, nous avons proposé un ensemble de règles de reconfiguration instanciées à un cas d'étude de gestion de la foule dans un réseau de lignes de métro. Ces règles sont proposées dans le cadre d'un modèle à base de graphe pour guider la planification de l'AMS face au besoin local en QoS. Elles reposent sur une approche couplant le modèle analytique avec des techniques de description et de transformation de graphes. Cette approche permet d'une part de déterminer le paramétrage des mécanismes a priori retenus par le composant de planification, et de valider l'applicabilité de ces mécanismes ; elle permet d'autre part de maintenir à tout moment un modèle du système résultant des actions d'adaptation réalisées, en cohérence avec le système réel. Dans ce chapitre, nous avons tout d'abord présenté le cas d'étude constitué des entités physiques. Ces entités ont été modélisées sur deux niveaux. Le premier niveau, *métier*, permet de modéliser les entités physiques sous l'angle du rôle qu'elles jouent dans la gestion. Il met aussi en évidence le besoin en QoS. Le deuxième niveau, *opératoire*, permet de raffiner le niveau métier en des nœuds représentant les applications et les entités Middleware. Le style architectural relatif à chaque niveau ainsi que les règles de reconfiguration guidant la planification ont été élaborés par la suite. Via ces règles de transformation, nous avons montré que le couplage de modèles issus de différents formalismes était possible et permettait de donner une description détaillée de certaines actions de reconfiguration (ici orientées ressources) pour guider la planification.

PERSPECTIVES

Plusieurs perspectives peuvent être considérées en prolongation des contributions décrites dans ce mémoire. Certaines constituent des extensions immédiates des contributions actuelles. Elles concernent notamment les modèles proposés pour le comportement de l'AMS et de l'AMM, notamment concernant ce dernier pour sa fonction de planification des objectifs de QoS locale confiés aux AMS. Elles concernent également la partie déploiement des mécanismes proposés, tant du point de vue de l'AMS que des entités amenées à intégrer ces mécanismes. D'autres perspectives, plus complexes, sont à considérer à moyen et long terme. Elles concernent tout particulièrement l'architecture du système IoT-Q, tant du point de vue structurel que comportemental, dans une double optique. La première optique consiste en la prise en compte des opportunités et des contraintes liées à l'environnement global de déploiement des mécanismes orientés QoS, dans une logique de *virtualisation de fonctions de niveau Middleware* inspirées de l'approche NFV. La deuxième optique consiste en la prise en compte

du goulot d'étranglement que représentent les réseaux traversés dans une logique de *gestion adaptative multi niveaux* de mécanismes orientés QoS.

Nous détaillons ci-après certaines de ces différentes perspectives.

Comportement de l'AMS

Concernant la phase de monitoring présenté au chapitre 4, le focus a été mis sur deux approches de supervision, réactive et proactive. Suivant la première approche, nous avons proposé une méthode basée sur le couplage du modèle analytique avec des techniques de CEP. Dans la deuxième approche, nous avons intégré à la méthode précédente un modèle ARMA pour prédire les taux d'arrivée des requêtes. Dans les deux cas, les *patterns* correspondant à une dégradation (observée ou prédite) de la QoS sont fournis pour illustrer les deux approches. Comme perspective à court terme, il s'agit ainsi d'étudier comment construire des patterns adaptés, sur la base d'une connaissance approfondie du comportement des entités Middleware dans différents scénarios. Cette capacité a vocation à être intégrée au composant de Monitoring. Des techniques issues de l'intelligence artificielle et plus spécifiquement de l'apprentissage automatique pourraient être explorées à cette fin.

Concernant la phase de planification, nous avons proposé au chapitre 5 un ensemble de règles opérant sur un modèle du système à base de graphe pour guider la reconfiguration des entités Middleware de l'architecture. Une heuristique consistant en un protocole de planification a été proposé à titre illustratif de l'approche. Pour autant, la planification proprement dite reste un problème ouvert. Cette planification pourrait être envisagée via des techniques basées sur l'intelligence artificielle.

Comportement de l'AMM

Notre proposition d'architecture IoT-Q est basée sur un principe de gestion hiérarchique de la QoS au niveau Middleware via : 1) un AMS par entité Middleware pour la gestion des mécanismes opérationnels à mettre en œuvre pour satisfaire un objectif de QoS locale, et 2) un AMM central pour la coordination des objectifs de QoS locale confiés aux AMS, pour aboutir à la satisfaction de l'objectif de QoS de bout en bout. Dans cette thèse, les approches et les modèles comportementaux proposés ont porté sur le seul AMS. A moyen terme, une perspective qui se dégage est relative à l'AMM. Il s'agit en effet d'aborder la partie comportementale de ses composants internes, en définissant les approches et les modèles pour traiter la coordination et l'adaptation dynamique des objectifs de QoS locale confiés aux AMS.

Déploiement dynamique des mécanismes de gestion de la QoS

Le chapitre 2 a présenté les mécanismes pour la mise en œuvre de la QoS au niveau Middleware. Ces mécanismes ont vocation à opérer soit sur le trafic à destination des entités Middleware, soit sur les ressources dont disposent ces dernières. Une perspective à court terme consiste à mettre en œuvre la phase de déploiement effectif de ces mécanismes. Pour cela, il s'agit d'une part de concevoir et d'implémenter les éléments d'architecture du composant d'exécution de l'AMS permettant de déployer dynamiquement ces mécanismes. De façon complémentaire, il s'agit de concevoir et d'implémenter l'architecture d'accueil de ces mécanismes au niveaux des entités gérées, notamment au sein des entités Middleware, tout en respectant les éléments de standard (SmartM2M et oneM2M).

Evolution de l'architecture IoT-Q

Nous identifions trois axes majeurs d'évolution de l'architecture IoT-Q.

Le principe de conception basée politiques hiérarchiques de l'architecture IoT-Q vise à assurer une résistance au facteur d'échelle du système vis-à-vis du nombre d'entités Middleware à gérer, du nombre de métriques à considérer, etc. La vérification de cette hypothèse constitue une perspective de notre travail. Considérer la mise en œuvre de notre solution en tant que service cloud (*IoT-Q as a Service*) est un premier axe à explorer pour en assurer la véracité.

En liaison directe avec la perspective de déploiement des mécanismes de gestion de la QoS dans des environnements physiques ou virtuels, le deuxième axe d'évolution de l'architecture concerne la prise en compte des problématiques allant, pour certaines, au-delà de celles relevant de la conception et du développement logiciel. Ces problématiques résultent notamment de l'hétérogénéité des solutions de déploiement envisageables, qui ouvrent des opportunités mais aussi des contraintes, ou encore de l'objectif d'aboutir à une forme de transparence (*service sans couture*) dans le déploiement dynamique de ces mécanismes, non seulement du point de vue des utilisateurs finaux mais également des applications et des entités Middleware. L'exploration des techniques de virtualisation de fonctions de réseau (au sens NFV - *Network Function Virtualisation*) ici appliquées au Middleware, couplées aux techniques de gestion de réseaux guidée par le logiciel (au sens SDN - *Software Defined Network*) est une voie qui nous semble prometteuse. Le traitement de ces problématiques dans cet esprit induira le besoin de reconsidérer la partie structurelle de l'architecture IoT-Q, et également d'étendre et/ou de reformuler les modèles envisagés pour le comportement des composants de monitoring et de planification.

En prolongement de ces derniers travaux, une perspective à plus long terme est d'étendre l'architecture IoT-Q de façon à doter le système d'une capacité de gestion dynamique et autonomique de la QoS au niveau des réseaux traversés, dans une vision à présent *multi-niveaux* couplant gestion au niveau Middleware et gestion au niveau réseau au sens large (couches Transport et (inter)-Réseau de l'architecture TCP/IP). Cette extension peut être envisagée suivant la même approche hiérarchique que celle adoptée pour le niveau Middleware (basée sur un AMM-MW et des AMS-MW), étendue via des gestionnaires autonomiques pour la gestion du réseau (AMN-NW et AMS-NW de la Figure i.1). Sur la figure, un niveau de gestion supplémentaire est ajouté pour assurer une gestion globale de la QoS aux niveaux Middleware et Réseau. La gestion globale est assurée par un AMM central ayant autorité sur les gestionnaires AMM-MW et AMM-NW. Cet AMM central est responsable de la prise en compte des besoins en QoS de bout en bout de l'application, de leur traduction en besoins en QoS globale aux niveaux Middleware et Réseau, ainsi que de l'adaptation de ces objectifs.

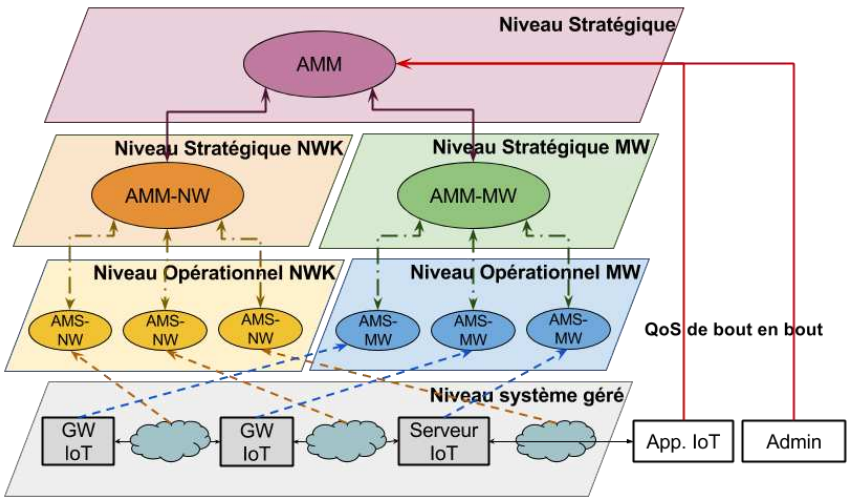

Figure i.1 - Extension de l'architecture pour la gestion hiérarchique multi-niveaux

Publications de l'auteur

Conférences internationales avec actes et comité de lecture

- Y. Banouar, T. Monteil, C. Chassot. "Analytical Model for Adaptive QoS Management at the Middleware level in IoT". The 22nd IEEE Symposium on Computers and Communications (ISCC'2017). 03 – 06 Juillet 2017. Héraklion, Grèce.
- Y. Banouar, C. Ouedraogo, C. Chassot, A. Zyane. "QoS Management Mechanisms for Enhanced Living Environments in IoT". IFIP/IEEE International Symposium on Integrated Network Management. 08 – 12 Mai 2017. Lisbonne, Portugal.
- Y. Banouar, S. Reddad, C. Diop, C. Chassot, A. Zyane. "Monitoring Solution for Autonomic Middleware-level QoS within IoT". 12th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'2017). 17 – 20 Novembre 2015. Marrakech, Maroc.
- Y. Banouar, S. Reddad, C. Diop, C. Chassot. "Towards Autonomic Middleware-level Management of QoS for IoT applications". The 2nd IOT360 Summit. Book chapter "Internet of Things – IoT Infrastructures". Springer 2016. 26 – 29 Octobre 2015. Rome, Italie.
- M. Ben Alaya, Y. Banouar, T. Monteil, C. Chassot, K. Drira. "OM2M: Extensible ETSI-compliant M2M Service Platform with Self-configuration Capability". Procedia Computer Science, vol. 32, no. 0, pp. 1079 – 1086, 2014. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014). 02 – 05 Juin 2014. Hasselt, Belgique.

Manifestation d'audience nationale

- Y. Banouar, T. Monteil, C. Chassot. "Modèle analytique pour la gestion adaptative de la QoS au niveau Middleware dans l'IoT". 1er Séminaire TOulousain en RÉseau (STORE). 8 Novembre 2016. Toulouse, France.
- Y. Banouar, T. Monteil, M. Ben Alaya, C. Chassot, K. Drira. "OM2M: standardized service platform for M2M interoperability". EclipseCON France 2014. 18 - 19 Juin 2014. Toulouse, France.

Journal

- Y. Banouar, C. Chassot, S. Medjiah, C. Eichler, K. Drira. "A model-based planning of QoS-oriented adaptation actions at the Middleware level in IoT". IEEE Transaction on Network and Service Management (IEEE TNSM) journal. En cours de soumission.

Liste des acronymes

AC	Autonomic Computing
AE	Application Entity
AM	Autonomic Manager
AMM	Autonomic Manager Master
AMS	Autonomic Manager Slave
ARMA	Auto-Regressive Moving Average
CEP	Complex Event Processing
CoAP	Constrained Application Protocol
CSE	Common Service Entity
DDS	Data Distribution Service for Real-Time Systems
DiffServ	Differentiated Services
DPO	Double PushOut
edNCE	edge directed Neighbourhood Controlled Embedding
EPL	Event Processing Language
ETSI	European Telecommunication Standard Institute
HTTP	HyperText Transfer Protocol
IntServ	Integrated Services
IoT	Internet of Things
JMX	Java Management Extensions
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
M2M	Machine-to-Machine
MAPE	Mean Absolute Percentage Error
MAPE-K	Monitor-Analyze-Plan-Execute-Knowledge base
MDA	Model Driven Architecture

MQTT	MQ Telemetry Transport
N/G/D-SCL	Network/Gateway/Device-Service Capability Layer
NAC	Negative Application Condition
OSI	Open Systems Interconnection
QoS	Quality of Service
REST	Representational State Transfer
RFC	Request For Change
RTT	Round Trip Time
SDN	Software Defined Network
SPO	Single PushOut
UDP	User Datagram Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WFQ	Weighted Fair Queuing
WTP	Web Transfer Protocol
XML	Extensible Markup Language

Références

[act04] IBM, “A Practical Guide to the IBM Autonomic Computing Toolkit,” April 2004.

[adr12] Bâtiment ADREAM, site web: https://www.laas.fr/public/fr/le-projet-adream.

[aka74] Hirotugu Akaike, “A new look at the statistical model identification”, IEEE Transactions on Automatic Control, vol. 19, no 6, 1974, p. 716-723.

[ala16] I. Al-Anbagi, M. Erol-Kantarci, H. T. Mouftah, “A survey on cross-layer quality-of-service approaches in WSNs for delay and reliability-aware applications,” IEEE Communication Survey. Tuts., vol. 18, no. 1, pp. 525-552, 1st Quart. 2016.

[alf15] A. Al-Fuqaha, A. Khreishah, M. Guizani, A. Rayes, “Toward better horizontal integration among IoT services,” IEEE Communication Magazine, vol. 53, no. 9, pp. 72-79, Sep. 2015.

[alf152] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash. ‘‘Internet of Things: A survey on enabling technologies, protocols, and applications,’’ IEEE Communication Surveys Tutorials. vol 17, no. 4, pp. 2347–2376, 4th Quarter, 2015.

[all17] AllJoyn, AllSeen Alliance, accès en ligne à https://allseenalliance.org.

[ame94] Amer P, Chassot C, Connolly C, Conrad P, Diaz M., “Partial Order Transport Service for Multimedia and other Applications,” IEEE/ACM Transactions on Networking, vol. 2, n° 5, October 1994.

[amq12] OASIS standard, “OASIS Advanced Message Queuing Protocol,” Version 1.0, October 2012.

[ani11] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N., “Ep-sparql: a unified language for event processing and stream reasoning,” In: WWW 2011, pp. 635–644 (2011).

[ara06] A. Arasu, Arvind; S. Babu J. Widom, “The CQL Continuous Query Language: Semantic Foundations and Query Execution,” VLDB Journal, Vol. 15, No. 2, June, 2006.

[ash09] Ashton K., “That 'internet of things' thing, in the real world things matter more than ideas,” RFID Journal, 22 June 2009.

[atz10] L. Atzori, A. Iera, G. Morabito, “The Internet of Things: A survey,” Computer Networks, vol 54, no. 15, pp. 2787-2805, October 2010.

[bal07] Hitesh Ballani and Paul Francis: “CONMan: A Step towards Network Manageability,” In SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications, pages 205-216, New York, NY, USA, 2007. ACM.

[ban15] Y. Banouar, S. Reddad, C. Diop, C. Chassot and A. Zyane, “Monitoring solution for autonomic Middleware-level QoS management within IoT systems,” 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), Marrakech, 2015, pp. 1-8.

[bas75] F. Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. “Open, closed, and mixed networks of queues with different classes of customers,” J. ACM 22, April 1975.

[bay97] B. Baynat, “Théorie des files d’attente. Des chaînes de Markov aux réseaux à forme produit,” Hermes Sciences Publications, 1997.

[ben14] M. Ben Alaya, Y. Banouar, T. Monteil, C. Chassot, K. Drira. “OM2M: Extensible ETSI-compliant M2M Service Platform with Self-configuration Capability”. Procedia Computer Science, vol. 32, no. 0, pp. 1079 – 1086, 2014. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014). June 02 – 05, 2014. Hasselt, Belgium.

[ben15] M. Ben Alaya, T. Monteil, “Frameself: an ontology-based framework for the self-management of m2m systems,” Concurrency and Computation: Practice and Experience 2015; 27(6):1412–1426.

[bg15] A. Banks and R. Gupta., “MQTT Version 3.1.1 Errata 01,” OASIS Approved Errata, 10 Décembre 2015.

[bir84] Birrell, A. D. and Nelson, B. J., “Implementing remote procedure calls,” ACM Transactions on Computer Systems, pp. 39-59, March 1984.

[boo10] Booth, C., “Chapter 2: IP Phones, Software VoIP, and Integrated and Mobile VoIP,” Library Technology Reports. 46 (5): 11–19, 2010.

[bou11] I. Bouassida, “Gestion dynamique des architectures pour les systèmes communicants collaboratifs,” Thèse de doctorat, INSA de Toulouse - Université Toulouse III, 2011.

[box70] George Box and Gwilym M. Jenkins, “Time Series Analysis: Forecasting and Control, Holden-Day Inc., San Francisco, Calif., 1970.

[box94] George Box, Gwilym M. Jenkins, and Gregory C. Reinsel, “Time Series Analysis: Forecasting and Control,” third edition. Prentice-Hall, 1994.

[bra88] F.J.Brandenburg, “On polynomial time graph grammars,” Proc. STACS 88, Lecture Notes in Computer Science 294, Springer-Verlag, Berlin, pp.227-236, 1988.

[bro03] Enterprise Integration Patterns, “Hub and Spoke [or] Zen and the Art of Message Broker Maintenance,” November 2003.

[bro87] Brockwell, P. J. and Davis, R, “Time Series: Theory and Methods,” Springer-Verlag, New York, 1987.

[cha08] Ranganai Chaparadza: “Requirements for a Generic Autonomic Network Architecture (GANA), suitable for Standardizable Autonomic Behaviour Specifications for Diverse Networking Environments,” International Engineering Consortium (IEC), Annual Review of Communications, 61, 2008.

[cha09] I. Cha, Y. Shah, A. U. Schmidt, A. Leicher, and M. Meyerstein, “Security and trust for m2m communications 1,” 2009.

[chu10] Church S. and Pizzi S., “Audio Over IP,” Focal Press., p. 191, 2010.

[cla03] David D. Clark, Craig Partridge, and J. Christopher Ramming: “A knowledge plane for the Internet,” In SIGCOMM, pages 3-10, 2003.

[con00] A. R. Conn, N. I. M. Gould, Ph. L. Toint, “Trust-region methods. MPS-SIAM series on optimization SIAM and MPS,” Philadelphia, 2000.

[cor92] CORBA OMG, disponible à http://www.corba.org, 2017.

[cpk09] Ranganai Chaparadza, Symeon Papavassiliou, Timotheos Kastrinogiannis, Martin Vigoureux, Emmanuel Dotaro, Alan Davy, Kevin Quinn, Michal Wódczak, Andras Toth, Athanassios Liakopoulos, and Mick Wilson, “Towards the Future Internet - A European Research Perspective,” chapter “Creating a viable Evolution Path towards Self-Managing Future Internet via a Standardizable Reference Model for Autonomic Network Engineering,” pages 313-324. IOS Press, 2009.

[cro09] Doug Crockford, “Google Tech Talks: JavaScript: The Good Parts,” 7 February 2009.

[cug12] G Cugola and A. Margara, Processing flows of information: From data stream to complex event processing. ACM Computing Surveys, 44(3), article 15, 2012.

[dds04] OMG, “Data-Distribution Service for Real-Time Systems,” Version 1.0, December 2004.

[dio15] C. Diop “An Autonomic Service Bus for Service-based Distributed Systems,” Thèse de doctorat, Université Fédérale de Toulouse, Avril 2015.

[dua11] Ren Duan, Xiaojiang Chen, and Tianzhang Xing, “A qos architecture for iot,” International Conference on and 4th International Conference on Cyber, Physical and Social Computing, China, 2011.

[ecall] eCall project, https://ec.europa.eu/digital-single-market/en/news/ecall-all-new-cars-april-2018, accès en Janvier 2017.

[ehe17] Europa, http://europa.eu.int/information_society/eeurope/ehealth, accès en Février 2017.

[ehr90] H. Ehrig and H.-J. Kreowski, “Graph grammars and their application to computer Science,”. Springer-Verlag, 1990.

[eic15] C. Eichler, “ Modélisation formelle de systèmes dynamiques autonomes : graphe, réécriture et grammaire,” Thèse de doctorat, Université de Toulouse, 2015.

[ekl90] H. Ehrig, M. Kor, and M. Löwe, “Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts,” In 4th International Workshop on Graph Grammars and Their Application to Computer Science, pages 2437, Bremen, Germany, March 1990. Springer-Verlag.

[emm00] Emmerich, W., “Software engineering and middleware: a roadmap”, Conference on The future of Software engineering, pp. 117-129, 2000.

[eng89] J.Engelfriet, “Context-free NCE graph grammars,” Proc. FCT '89, Lecture Notes in Computer Science 380, Springer-Verlag, pp.148-161, Berlin, 1989.

[eng97] J. Engelfriet and G. Rozenberg “Handbook of Graph Grammars and Computing by Graph Transformation,” chapter Node Replacement Graph Grammars, pages 194. World Scientific Publishing, 1997.

[esb02] Lapeira, Raul. “ESB is an architectural style, a software product, or a group of software products?”. Artifact Consulting. 2002

[esper] ESPER, accessible à http://www.espertech.com/products/esper.php.

[ets102] ETSI TR 102 732 v1.1.1. “Machine-to-Machine communications (M2M); Use Cases of M2M applications for eHealth,” September 2013.

[ets118] ETSI TS 118 109, “HTTP Protocol Binding,” v1.0.0, February 2015.

[ets690] ETSI TS 102 690, “Machine-to-Machine communications (M2M); Functional architecture,” v2.1.1, October 2010.

[etz10] ETZION O., NIBLETT P., “Event Processing in Action,” p. 384, 12 August 2010.

[fie00] Roy Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. Thèse de doctorat, soutenue en 2000.

[fie00] R. T. Fielding & R. N. Taylor, “Architectural styles and the design of network-based software architectures,” p. 151, Doctoral dissertation: University of California, Irvine. 2000.

[fiw17] Fiware, accès à https://www.fiware.org/

[fle01] P. Flegkas, P. Trimintzios, G. Pavlou, I. Andrikopoulos, C. Cavalcanti, “Policy-based extensible hierarchical network management”, Proc. Workshop Policies for Distributed Systems and Networks (Policy 2001), Spinger-Verlag, 2001-Jan.

[fus12] RedHat Fuse, http://www.redhat.com/about/news/press-archive/2012/6/red-hat-to-acquire-fusesource, accès en January 2017.

[gal05] J.R. Gallego, A. Hernandez-Solana, M. Canales, J. Lafuente, A. Valdovinos, J. Fernandez-Navajas, “Performance Analysis of Multiplexed Medical Data Transmission for Mobile Emergency Care Over the UMTS Channel,” IEEE Trans. Information Technology in Biomedicine, vol. 9, no. 1, pp. 13-22, Mar. 2005.

[gar17] Gartner, http://www.gartner.com/it-glossary/internet-of-things, accès en Septembre 2016.

[ger06] R. Gerven, S. Jaarsma, and R. Wilhite, “Smart metering,” Kema, The Netherlands, July 2006.

[gha16] G. Gharbi, “ Gestion autonomique d'objets communicants dans le cadre des réseaux Machine-à-Machine sous des contraintes temporelles,” Thèse de doctorat, Université de Toulouse, 2016.

[goe06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, “Java Concurrency in Practice,” Print ISBN-10: 0-321-34960-1. 2006.

[gre05] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan and Hui Zhang: “A Clean Slate 4D Approach to Network Control and Management. SIGCOMM Computer Communication Review,” 35(5):41-54, 2005.

[gue06] M. K. Guennoun, “Architectures dynamiques dans le contexte des applications à base de composants et orientées service,” Thèse de doctorat, Université Paul Sabatier - Toulouse III, Décembre 2006.

[hei04] W. Heinzelman, A. Murphy, H. Carvalho, M. Perillo, “Middleware to support sensor network applications,” Network, IEEE, vol. 18, issue 1, pp. 6-14, 2004

[hol89] Holtzman, S., “Intelligent Decision Systems,” Addison-Wesley. 1989.

[ida14] Info-communications Development Authority of Singapore (IDA), “Internet of things roadmap,” 2014.

[idl02] OMG IDL, “CORBA 3.0 - OMG IDL Syntax and Semantics chapter,” July 2002.

[ieee90] IEEE, “IEEE standard glossary of software engineering terminology,” 1990.

[ier14] IERC Cluster SRIA, “Internet of Things: From Research and Innovation to Market Deployment,” 2014.

[ierc14] IERC, Internet of Things: From Research and Innovation to Market Deployment. IERC Cluster SRIA, 2014.

[ish13] I. Ishaq, D. Carels, G. K. Teklemariam, J. Hoebeke, F. V. d. Abeele, E. D. Poorter, I. Moerman, and P. Demeester, “Ietf standardization in the field of the internet of things (iot): a survey,” Journal of Sensor and Actuator Networks, vol. 2, no. 2, pp. 235–287, 2013.

[iso9000] ISO 9000, accès à https://www.iso.org/fr/standard/45481.html, 2015.

[itu01] ITU-T Recommendation G.1010, “End-user multimedia QoS categories,” November 2001.

[itu08] ITU-T E.800 Recommendations, “Quality of telecommunication services: concepts, models, objectives and dependability planning - Terms and definitions related to the quality of telecommunication services,” September 2008.

[jai08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uvgur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik, “Towards a Streaming SQL Standard,” Proc. VLDB Endow., 1(2) :1379–1390, August 2008.

[jmx17] Oracle JMX, accessible à https://docs.oracle.com/javase/tutorial/jmx

[jol17] Jolokia, accessible à https://jolokia.org

[kat13] Katusic, D., Skocir, P., Bojic, I., Kusek, M., Jezic, G., Desic, S., Huljenic, D., "Universal identification scheme in machine-to-machine systems," In: 2013 12th International Conference on Telecommunications (ConTEL), pp. 71–78, 2013.

[ken53] Kendall D. G., "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain," The Annals of Mathematical Statistics, vol. 24, no 3, 1953

[kep03] J. Kephart and D. Chess, "The vision of autonomic computing," Computer, vol. 36, No. 1, pp. 41-50, 2003.

[kep03] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing," IEEE Comp, 36 (1), Jan 2003.

[kle61] L. Kleinrock, "Information Flow in Large Communication Nets," Proposition de thèse de doctorat, Massachusetts Institute of Technology, Mai 1961.

[lin13] K.-J. Lin, N. Reijers, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu, "Building Smart M2M Applications Using the WuKong Profile Framework," 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, pp. 1175-1180, August 2013.

[lin15] Linux Foundation, "IoTivity Open Source Project Announces Preview Release,"14 January 2015.

[low03] Lowekamp B. B., "Combining active and passive network measurements to build scalable monitoring systems on the grid," Performance Evaluation Review, vol. 30, no. 4, pp. 19-26, March 2003.

[luc02] David Luckham, "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems," Addison-Wesley Professional, 1st edition, 2002.

[luk13] Lukman Ab. Rahim and Jon Whittle, "A survey of approaches for verifying model transformations," Software & Systems Modeling, pages 1–26, 2013.

[mcl02] Scott McLean, James Naftel and Kim Williams. "Microsoft .NET Remoting," Microsoft Press. 2002.

[mic06] Michelson, B., "Event-driven architecture overview," Patricia Seybold Group, Feb, 2006

[mof93] J. Moffett, M. Sloman, "Policy Hierarchies for Distributed Systems Management," IEEE Journal on Selected Areas in Communications, Vol. 11, No. 9, pp. 1404-141, December 1993.

[mor07] Morison R., Balasubramaniam D., Oquendo F., Warboys B., and Greenwood R. M., "An active architecture approach to dynamic systems co-evolution," In Flavio Oquendo, editor, Software Architecture, vol. 4758 of Lecture Notes in Computer Science, pages 2–10. Springer Berlin Heidelberg, 2007.

[nik13] N. Nikaein, M. Laner, K. Zhou, P. Svoboda, D. Drajic, M. Popovic, S. Krco, "Simple traffic modeling framework for machine type communication," International Symposium on Wireless Communication Systems, pp. 1-5, Aug 2013.

[nin13] H. Ning, H. Liu and L. Yang, "Cyber-entity security in the Internet of things," Computer, vol. 46, no. 4, pp. 46-53, 2013.

[ocf16] OCF, "IoT Standards Get a Big Push: Meet the Open Connectivity Foundation (OCF)," 23 February 2016.

[omg06] OMG (Object Management Group). "Unified Modeling Language: Infrastructure". Version 2. Mars 2006.

[one15] oneM2M TS v1.6.1, "oneM2M functional architecture," January 2015.

[one16] oneM2M, TS v1.5.1, "MQTT Protocol Binding," 29 February 2016.

[orb13] IBM, "Object Request Brokers," November 2013.

[oul15] Oulmahdi M, Chassot C, Van Wambeke N., "Transport protocols: limitations, evolution obstacles and solutions for an actual deployment in the Internet," International Journal of Parallel, Emergent and Distributed Systems, Taylor & Francis, 2015, Special Issue: Smart Communications in Network Technologies, Volume 30 (Issue 6), p. 515-535.

[pan13] Nikolaos A. Pantazis, Stefanos A. Nikolidakis and Dimitrios D. Vergados, "Energy-Efficient Routing Protocols in Wireless Sensor Networks: Survey," IEEE communications surveys tutorials, Vol 15, No. 2, Second Quarter 2013.

[pet12] I. Petiz, P. Salvador, and A. N. Nogueira. "Characterization and modeling of m2m video surveillance traffic," In IARIA Int. Conf. on Advances in Future Internet - AFIN, August 2012.

[rab07] "Launch of RabbitMQ Open Source Enterprise Messaging," Press release. February 8, 2007.

[raz16] M.A. Razzaque, M. Milojevic-Jevric, S. Clarke, "Middleware for Internet of Things: A Survey," in Internet of Things Journal, IEEE, vol. 3, no. 1, February 2016.

[red15] IBM, "Node-RED, a visual tool for wiring the internet of things," 2015.

[rfc1057] Sun Microsystems, Inc. "RPC: Remote Procedure Call Protocol Specification, Version 2," RFC 1057. Juin 1988.

[rfc1094] Sun Microsystems, Inc. “NFS: Network File System Protocol Specification”. RFC 1094, March 1989.

[rfc1633] R. Braden, D. Clark, S. Shenker, “Integrated services in the internet architecture: an overview,” RFC 1633, IETF, June 1994.

[rfc2475] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, “An architecture for differentiated services,” RFC 2475, IETF, December 1998.

[rfc3135] Border, J., Kojo, M., Griner, J., Montenegro, G., et Z. Shelby, “Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations,” RFC 3135, June 2001.

[rfc7252] Z. Shelby, ARM, K. Hartke, C. Bormann. “The Constrained Application Protocol (CoAP),” RFC 7252. Juin 2014.

[rfc7303] Internet Engineering Task Force, “XML Media Types,” RFC 7303, July 2014.

[ric08] L. Richardson and S. Ruby, “RESTful web services,” O’Reilly Media, Inc., 2008.

[riz12] Dimitris Rizopoulos, “Joint Models for Longitudinal and Time-to-Event Data, with Applications in R,” Chapman & Hall/CRC, Boca Raton, 2012. ISBN 978-1-4398-7286-4.

[rmi17] Oracle, “Remote Method Invocation specification,” 2017.

[saa11] Saamer Akhshabi; Ali C. Begen; Constantine Dovrolis, “An Experimental Evaluation of Rate-Adaptation Algorithms in Adaptive Streaming over HTTP,” In Proceedings of the second annual ACM conference on Multimedia systems (MMSys '11). New York, NY, USA: ACM. 2011.

[sha96] M. Shaw and D. Garlan, “Software architecture: perspectives on an emerging discipline,” Vol. 1, p. 12. Englewood Cliffs: Prentice Hall, 1996.

[sir14] A. Sîrbu, S. Caminiti, P. Gravino, V. Loreto, V. Servedio, F. Tria, “A new platform for Human Computation and its application to the analysis of driving behaviour in response to traffic information,” CCS14 Proceedings in Human Computation, 2014.

[sko10] L. Skorin-Kapov and M. Matijasevic, “Analysis of qos requirements for e-Health services and mapping to evolved packet system qos classes,” International Journal of Telemedecine and Applications. July 2010.

[sny10] Snyder, Bruce; Bosanac, Dejan; Davies, Rob, “ActiveMQ in Action,” (1st ed.), Manning Publications, p. 375, March 2010.

[soa06] OASIS standard, “SOA RM - Reference Model for Service Oriented Architecture 1.0,” October 2006.

[sol00] Soley, R. and the OMG Staff Strategy Group, Object Management Group, White Paper, Draft 3.2 – November 27, 2000.

[sta10] M. Starsinic, “System architecture challenges in the home M2M network,” Applications and Technology Conference (LISAT), 2010 Long Island Systems, vol., no., pp.I,7, 7-7 May 2010.

[str06] John C Strassner, Nazim Agoulmine and Elyes Lehtihet: “FOCALE - A Novel Autonomic Networking Architecture, “ In Latin American Autonomic Computing Symposium (LAACS), Campo Grande, MS, Brazil, 2006.

[uk17] Gouvernement du Royaume-Uni, Département de l'énergie et du changement climatique, http://www.gov.uk/smart-meters-how-they-work, accès en Mars 2017.

[van13] Van Wambeke N, Exposito E, Chassot C, Diaz, M., “ATP: A Micro-Protocol Approach to Autonomic Communication,” IEEE Transactions on Computers, vol. 62, Issue 11, November 2013.

[van89] V.van Oostrom, “Graph grammars and 2nd order logic (in Dutch),” M. Sc. Thesis, Leiden University, January 1989.

[wen12] L. Wen-xiang, X. Jun, and J. Hao, “Queuing states analysis on a hybrid scheduling strategies for heterogeneous traffics in iot,” Computer Science & Service System (CSSS), pp.1007-1008, August 2012.

[wey14] Weyrich, M., Schmidt, J.-P., Ebert, C., “Machine-to-Machine Communication,” IEEE Software, August 2014.

[wu11] Wu G., Talwar S., Johnsson K., Himayat N., Johnson K., “M2M: From Mobile to Embedded Internet,” IEEE Communications Magazine, 2011.

[yu14] S.-Y. Yu, Z. Huang, C.-S. Shih, K.-J. Lin, J. Hsu, “Qos oriented sensor selection in iot system,” in: IEEE and Internet of Things (iThings/CPSCom), 2014.

[zen04] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware middleware for web services composition,” IEEE Transactions on Software Engineering, 30(5), May 2004