A Modular Framework for Dynamic QoS Management at the Middleware Level of the IoT: Application to a OneM2M Compliant IoT Platform

Clovis Anicet Ouedraogo, Samir Medjiah, Christophe Chassot

A Modular Framework for Dynamic QoS Management at the Middleware level of the IoT

Application to a oneM2M compliant IoT platform

Clovis Anicet Ouedraogo [1], Samir Medjiah [1,2], Christophe Chassot [1,3]
[1] CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France
[2] Univ. Toulouse, UPS, LAAS, F-31400 Toulouse, France
[3] Univ. Toulouse, INSA, LAAS, F-31400 Toulouse, France
{ouedraogo, medjiah, chassot} @ laas.fr

***Abstract*—The Internet of things (IoT) has evolved exceptionally in recent years; enabling a large number of heterogeneous devices to be interconnected to users via the Internet. This new concept promises in a few years to interconnect billions of devices, which will generate many challenges on the infrastructure supporting these communications. One of these challenges is the satisfaction of the different QoS requirements of the applications. To address this challenge, we identified two bottlenecks with respect to the QoS, which are the networks and the intermediate entities (i.e. middleware) allowing the applications to interact with the devices. In this paper, we propose a modular framework to ensure the QoS of applications at the middleware-level through QoS-oriented mechanisms deployed dynamically and autonomously on the middleware entities. The benefits of this framework are presented through test scenarios in the vehicular transportation domain.**

Keywords— Internet of Things; Quality of Service; Middleware; Modular Framework; Dynamic deployment; Autonomic Computing;

I. INTRODUCTION

The Internet of Things (IoT) is a new paradigm that particularly relies on new communication architectures and protocols, including the middleware (MW) level. To ensure interoperability between connected objects and applications, the oneM2M initiative proposed a distributed REST-based middleware service [1]. Through this MW, applications can collect data and/or trigger commands thanks to distant sensors and actuators through simple (e.g. http) requests (see Fig. 1).

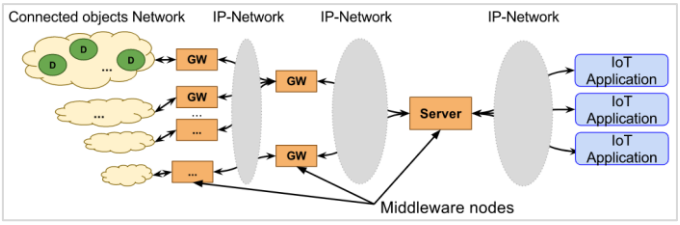

Fig. 1. Overview of an IoT Middleware

These different applications have different requirements in terms of Quality of Service (QoS), and particularly bounded response times. These needs face two bottlenecks: the traversed IP networks, and the intermediate entities implementing the MW layer. In this paper, we address the problem of the bottleneck represented by the MW layer with regard to the QoS requirements of IoT applications.

To tackle this QoS problem, the approach which is explored in this paper is based on the dynamic (i.e. at design-time but also at runtime) and autonomous (i.e. without human interactions) deployment of software modules implementing QoS-oriented mechanisms at the MW level. This deployment is achieved seamlessly for the applications and the connected objects. Basically, the considered mechanisms act on the applicative traffic with the aim to differentiate their processing within the considered MW nodes.

Modular programming is a software design technique [2] that emphasizes separating the functionality of a program into independent and interchangeable modules, such that each one contains everything necessary to execute only one aspect of the desired functionality. A module interface expresses the elements that are provided and required by the module.

To implement the proposed approach, we consider in Fig. 2 a typical deployment infrastructure for MW-based IoT applications. Each application is accessed using a user terminal such as computers or smartphones. An IoT MW is linking the application to the physical sensors and actuators. The application backend as well as some MW entities are deployed in public or private Cloud. Some MW entities may be deployed within the Service Provider private infrastructure using dedicated nodes (typically for IoT gateways) or generic deployment nodes such as COTS hardware. Moreover, an autonomic manager is supposed to provide the adequate modules deployment policies thanks to logical effectors, on the basis of the information (QoS metrics, resource state, etc.) retrieved from the logical sensors. Let us precise here that the paper is not focused on the autonomic manager.

The contributions of this paper include (1) the design and the implementation of the QoS management modules, (2) the architecture allowing their integration in MW entities, and (3) their performance evaluation in terms of benefits and costs through an illustrative scenario coming from the vehicular transportation domain.

The remainder of this paper is organized as follows. Section II states the considered problem as well as the related state of the art. Section III presents the integration architecture of the QoS

management modules into the oneM2M-compliant middleware entities. Section IV describes the design principles of these modules. Section V presents the performances emulation of our solution. Finally, Section VI concludes the paper and gives insights on our envisioned perspective works.

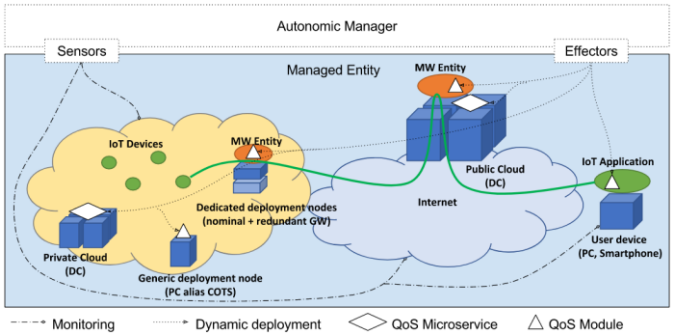

Fig. 2. Overview of the deployment approach

II. Contribution Positioning and Related Work

As part of the standardization efforts, various MW-level service layers have been proposed in the context of the IoT. These MW layers include several functional and nonfunctional services for the management of IoT devices. However, these standard-based MWs do not propose any solution for the QoS management at the MW level. Indeed, QoS is considered a result of the underlying networks [1].

However, several specific (i.e. not standard) solutions [3] have been proposed. [4] proposes to enhance the MW WuKong [5] for the QoS management. It introduces the concept of quality score that considers multiple QoS metrics (response time, reliability, etc.). Adequate physical devices are selected as well as their optimal deployment is decided in order to achieve the highest quality score. The limit of this approach lies in the fact that applications' QoS requirements are not taken into account dynamically. In the project MiLAN [6], Heinzelman proposes a MW that manages both nodes and the network. Depending on the application description and its expressed QoS requirements, the MW configures both the network and the MW nodes to meet these requirements. However, MiLAN needs a specific state diagram of every application scenario and for every WSN context, which is very complicated in the dynamically changing IoT context. Other solutions such as [7][8] rely on the integration of the MQTT protocol [9] for QoS management. This protocol includes a basic form of QoS management. It offers three levels of message delivery guarantee: QoS Level 0: Messages are delivered in a best-effort fashion without receive acknowledgment; QoS Level 1: Messages are guaranteed to be delivered at least once; and QoS Level 2: Messages are guaranteed to be delivered exactly once. Let us also note that the last specification of the oneM2M standard [10] proposes to integrate the MQTT protocol. MQTT does offer a certain guarantee for message delivery. However, in the IoT context, applications can have different requirements additionally to the simple message delivery guarantee.

III. Software Architecture for a Seamless Integration of QoS Mechanisms in a oneM2M-compliant MW

A. *Seamless integration in the oneM2M-compliant OM2M MW*

The oneM2M standard is expected to prevail as the main IoT MW architecture since it enables and facilitates the interoperability at different levels (Fig. 3). At the communication level, the IoT entities are able to "talk" to each other, i.e. applications/objects are able to communicate with other applications/objects independently from their access network technologies or communications protocols. At the data level (i.e. semantic interoperability), entities are able to "understand" each other. This is achieved through the semantic extension, present in oneM2M since its release 2 [1].

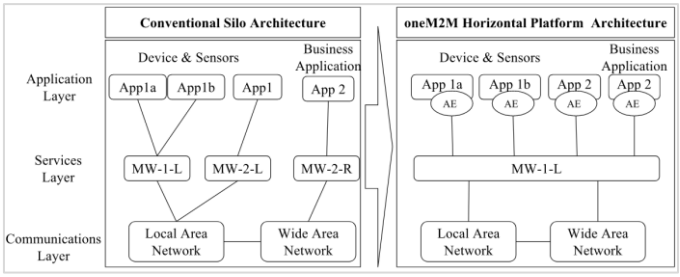

Fig. 3. oneM2M's horizontal architecture (providing a unifying framework for silo applications)

The oneM2M functional architecture identifies logical entities dubbed "MW nodes" (typically server / gateways of previous Fig. 1), each one offering a portion of the MW service. It is a resource-oriented architecture (RoA) where the functionality of the system is exposed by means of APIs. Each of these entities is composed of software modules that implement each one of the node's features. Thus, based on this modular architecture of the MW node, we propose to integrate the new QoS-oriented mechanisms as modules. These modules can be incorporated dynamically at design or run time in a seamless fashion without any modification of the original MW node.

As shown in Fig. 4, the integration location of these QoS modules into the communication model of a oneM2M MW node, is justified by the constraint imposed by the different application level protocols (HTTP, CoAP, MQTT) supported by the node. Each of these application protocols has a different message (request or response) structure. We have chosen to propose our QoS modules in a protocol-agnostic way. Thus, new applications protocols are automatically supported as long as the MW node support them.

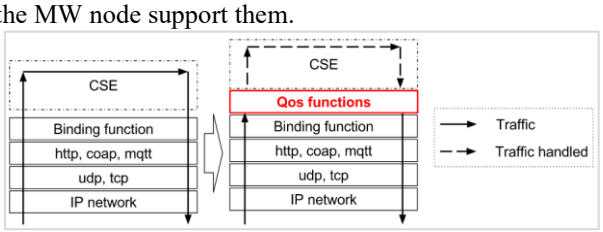

Fig. 4. Integration of QoS management modules into the oneM2M communication model of a MW node

The communication diagram shown in Fig. 5 represents the communication model before (1) and after (2) the integration of the QoS oriented modules.

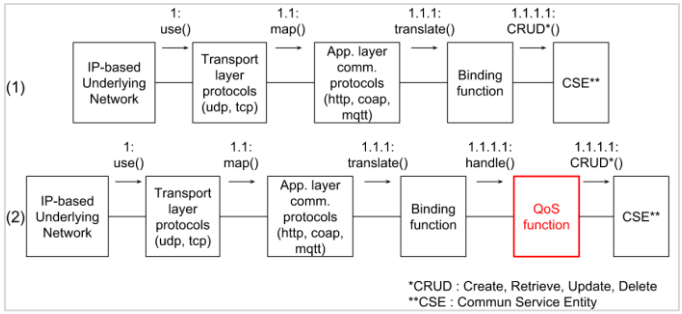

Fig. 5. Basic communication diagram of a oneM2M node before and after integration of QoS-oriented modules (QoS function)

This integration approach can be implanted in all open-source implementations of the oneM2M standard such as OCEAN Mobious [11], IoT-DM [12], or OASIS SI [13], since they share the same modular architectural style. In the following section, we apply our approach to the open source Eclipse OM2M platform [14], initially developed at LAAS-CNRS [15].

B. Application to Eclipse OM2M

OM2M nodes are built following a modular architectural style based on the OSGi standard [16]. Thanks to this implementation, it is possible to integrate our QoS mechanisms in the form of OSGi modules. Our integration approach is achieved so that the OM2M node maintains its modular design and can operates without these new modules (i.e. seamless integration).

An OM2M node (in-cse or mn-cse) is composed of the following modules:

- ***Core module:*** The *Core* module is responsible of the processing of generic requests and responses (i.e. protocol-agnostic messages). It implements features such as Registration, Discovery, Re-routing, Notifications, etc.
- ***Binding modules***: The *Binding* modules act as translators of protocol specific messages to generic messages and vice versa. A binding module is necessary for every supported protocol (HTTP, CoAP, MQTT, etc.)
- ***Persistence modules***: The *Persistence* modules are responsible for implementing the data storage strategy. There is on interface module, and as many as supported storage locations (in-memory, file or server databases)
- ***Interworking Proxy Entity (IPE) modules***: Similar to the *Binding* modules, they provide translation of generic messages into non-IP (Bluetooth, ZigBee, Z-Wave, etc.) messages and vice versa.

To achieve this integration, we had to consider two options: (1) to re-implement the Binding modules and *Interworking Proxy Entity* modules of a node in order to add a new interface to be used for the communication with our QoS modules. Such a modification would have resulted in a new version of Eclipse OM2M; or (2) to use the OSGi feature "Proxying Service" [17] which allows to intermediate an OSGi service. We have chosen the second option which enables the integration of our mechanisms without affecting the oneM2M standard being implemented through Eclipse OM2M. Furthermore, this option has the advantage not to change any element of the current implementation of OM2M.

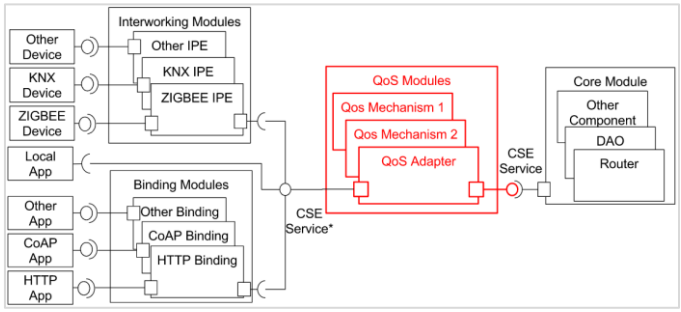

Fig. 6. Internal structure of an OM2M node integrating QoS-oriented modules

As shown in Fig. 6, the main element of this architecture is the "adapter" module. This component is specified following a design pattern [18]. It intermediates the OSGi service between the "core" module and the "binding" modules. Depending on its configuration (coming from the autonomic manager), it also decides to pass the request message through zero or several modules before reaching the "core" module. The same applies for the response message. An example is illustrated in Fig. 7.

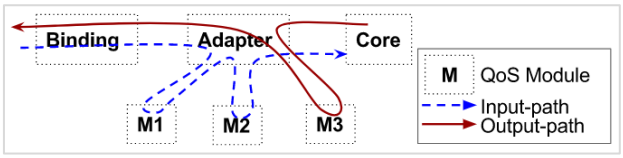

Fig. 7. Adapter module of the proposed architecture

IV. Design Principles of the QoS-Oriented Modules

In this section, we present the design of our QoS-oriented modules that implement mechanisms inspired from QoS management at the IP level coming from IETF DiffServ working group [19]. The DiffServ architecture is based on a simple model where traffic entering a network is first conditioned, then assigned to a class of behavior. Each class is uniquely identified by a code that can be found as a "mark" in the IP packets. The packets are then routed following the behavior associated to the code of the class to which they belong (e.g. packets marked A will be systematically delayed if the node load is higher that a given value). Following these same principles, five QoS modules acting on the applicative traffic (i.e. on the http requests/responses) have been designed and implemented in JAVA: The Adapter, the Classifier, the Dropper, the Delayer, and the Scheduler.

A. The Adapter

The algorithm of the Adapter module is inspired from [18]. When a message is received by this module, the input policy is checked in order to pass the message into every module indicated in the input policy and in the specified order (lines 1 to 7). The input policy being applied, and if no module has dropped the message, it is passed to the "core" module for

normal processing (line 8). The output policy is finally applied and similarly to the input policy (lines 9 to 17).

If any module drops the message (request as in line 5 or response as in line 14), an error message is issued and sent to the "binding" module for its transmission to the request issuer as specified by the oneM2M standard.

Algorithm 1: Adapter

```
      Input: message; modules; input-policy; output-policy; CSEService.
      Output: message.
00:   begin
01:   for (i=0 to sizeof(input-policy)) then
02:         id ← input-policy[i]
03:         service ← module[id]
04:         message ← service(message)
05:         if (message==null) then
06:         return error_message
07:   end for
08:   message ← CSEService(message)
09:   for (i=0 to sizeof(output-policy)) then
10:         id ← output-policy[i]
11:         service ←module[id]
12:         message ← service(message)
13:         if (message==null) then
14:         return error_message
15:   end for
16:   return message
17:   end
```

B. The Classifier

The Classifier module offers message classification and marking services. It first tries to identify the class of the received message (line 1). If a class is identified, the message is then marked with the associated tag (lines 2 and 5).

Algorithm 2: Classifier

```
      Input: message; pattern.
      Output: message.
00:   begin
01:   class ← findClassOf(message)
02:   if (class is identified) then
03:         tag ← pattern[class]
04:         Mark(message, tag)
05:   end if
06:   return message
07:   end
```

C. The Dropper

The Dropper module offers a service of packet rejection following a given rejection percentage. Upon the reception of a message, the Dropper module first identifies the priority of the message which associated to its mark (line 1), then calculates the percentage of the previously rejected messages with the same priority (line 4). If this percentage is still lower than the specified one in the QoS policy, then the message is rejected and a null message is returned to the Adapter. Else, the message is returned without any modification to the Adapter (lines 5 to 8).

D. The Delayer

The Delayer module offers to delay messages based on their priority. It first identifies the priority of the message (line 1), and then waits the required delay corresponding to the identified priority (lines 3 to 6). After the elapsed delay, the message is returned without modification to the Adapter module.

Algorithm 3: Dropper

```
      Input: message; pattern.
      Output: message.
00:   begin
01:   Initialize p ←message.priority
02:   if (p ≠ null) then
03:         if (p is a key of pattern) then
04:         percentage ← rejected[p]/total[p]
05:               if percentage<pattern[p] then
06:                     rejected[p] ← rejected[p]+1
07:                     message← null
08:               end if
09:         end if
10:   end if
11:   total[p] ← total[p]+1
12:   return message
13:   end
```

Algorithm 4: Delayer

```
      Input: message; pattern.
      Output: message.
00:   begin
01:   Initialize p ←message.priority;
02:   if (p ≠ null) then
03:         if (p is a key of pattern) then
04:               delay ← pattern[p]
05:               wait(delay)
06:         end if
07:   end if
08:   return message
09:   end
```

E. The Scheduler

The Scheduler module redefines the message scheduling based on their priority. It operates through two processes: (1) the first process enqueues the received message in an internal queue, and delivers this message when it progresses to the head of the queue (lines 1 to 11), (2) the second process schedules (reorders) the messages within the queue according to their priority (lines 13 to 17)

Algorithm 5: Scheduler

```
      Input: message; pattern.
      Output: message.
00:   begin
01:   Initialize p ←message priority;
02:   if (p ≠ null) then
03:         if (p is a key of pattern) then
04:         add message to queue
05:         wait for the message to be the head of the queue
06:         message ← head of the queue
06:         remove the head of the queue
07:         end if
08:   end if
09:   return message
10:   end
```

```
      Input: queue.
      Output: N/A
00:   begin
01:   while (true) then
02:         if (queue is not empty) then
03:               sort queue by priority
04:         end if
05:   end
```

V. Implementation and Evaluation

The efficiency of the proposed approach is evaluated through a scenario (case study) dealing with the vehicular

transportation area. Within this scenario, the first goal is to measure the expected benefits induced by a QoS policy consisting (to face a QoS degradation) in the dynamic deployment of several QoS modules on the considered MW entities. The second goal is to evaluate the cost associated to the performed deployment. The scenario is presented in the next section A. Section B and C are devoted to the evaluations associated to the two targeted goals.

A. Presentation of the scenario

We consider three Infrastructure-to-Vehicle (I2V) applications [20] having different QoS requirements (see Table I).

TABLE I. CONSIDERED IOT APPLICATIONS

App.	Description	Rate	Latency	Loss
A [20]	Traffic Signal Violation Warning	10 req/s (minimal)	100 ms (Allowable)	0% (Allowable)
B [20]	Free-Flow Tolling	10 req/s	50 ms (Allowable)	N/A
C [20]	Just-In-Time Repair Notification	10 req/s	N/A	0% (Allowable)

The considered architecture is shown in Fig. 8 and described through Table II. The applications are emulated through stochastic HTTP traffic injectors where 1500 requests are sent to the server and 1000 to the gateway, 500 requests from each application (A, B, and C). The MW level entities (server and gateway) are real entities whose specifications are provided in Table II.

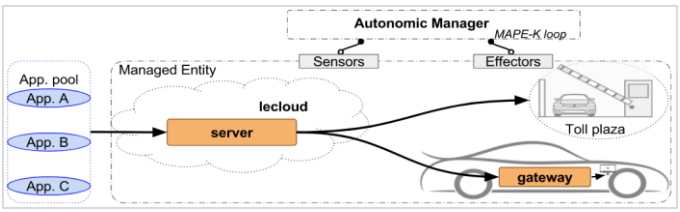

Fig. 8. Considered architecture for scenario

In order to assess the effects of the envisioned policies and implemented through the QoS modules, we have considered the processing time within the considered server and gateway MW nodes.

This scenario is divided into three stages:

1) Stage 1: Detection of an unsatisfied latency constraint and elaboration of a QoS-oriented global policy

The first stage is initiated by an event indicating to the autonomic manager that the latency constraint of application A (supposed to be more important than application A) is not satisfied. The autonomic manager is also notified that the average response times within the server and the gateway are respectively 65ms and 40ms, whatever the applicative traffic. In order to meet the 100ms of end-to-end latency required by the application A, still taking in account as best as possible the QoS requirement of applications B and C, we suppose that the autonomic manager decides to perform a global policy consisting in two successive (local) policies applied on the gateway then on the server (see stages 2 then stage 3). Let us recall here that our goal is not to discuss of the optimality of the chosen policy, but to evaluate the effectiveness of a plausible policy.

TABLE II. DESCRIPTION OF THE ARCHITECTURE SCENARIO

Managing Entity				
Element	**Description**			
Autonomic Manager	This entity implements the control loop MAPE-K **Erreur ! Source du renvoi introuvable.** and allows (among other tasks) to decide when, where and how to deploy the implemented modules in order to meet the QoS requirements of the targeted application(s).			
Touchpoints				
Element	**Description**			
Sensors	A logical component allowing to retrieve the processing time in the different nodes of the middleware.			
Effectors	A logical component allowing to add/remove or modify modules into the middleware nodes.			
Managed Entity				
Element	**Description**	**Specifications**		
		RAM (Gb)	CPU (x3.3 GHz)	Disk (Gb)
App. pool	A pool of applications including A, B, C applications.	-	-	-
Cloud	Virtualized environment hosting the "Server" entity.	4	4	40
Server	Entity that represents the set of instances of the infrastructure node specified in the oneM2M standard (i.e. IN-CSE)	1	1	10
Gateway	Entity that represents an intermediate middleware node as specified in the oneM2M standard (i.e. MN-CSE)	0.5	1	10

2) Stage 2: Implementation of a scheduling policy on the gateway (first part of the global policy)

The second stage consists in the implementation of the first part of the policy defined by the autonomic manager. This one has to deploy three different modules on the gateway: An Adapter, a Classifier and a Scheduler. This policy is aimed at prioritizing (on the gateway) the traffic coming from the application A (Fig. 9).

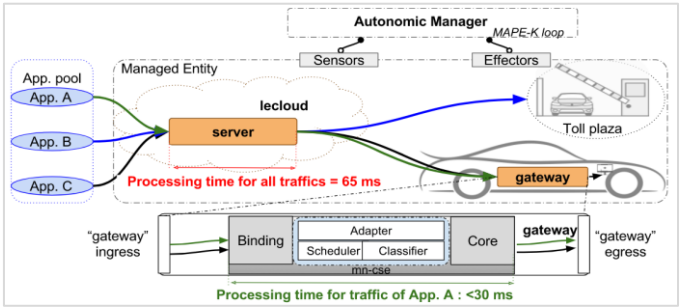

Fig. 9. Second stage of the scenario

3) Stage 3: Implementation of a joint scheduling and dropping policy on the server (second part of the global policy)

The third stage consists in the implementation of the second part of the chosen policy, through the combination of a scheduling policy and a dropping policy within the server. Four modules have then to be deployed by the autonomic manager: An Adapter, a Classifier, a Scheduler, and a Dropper. This policy is aimed at prioritizing (on the server) the traffic coming

from the applications A and B, and if the constraints is not satisfied, to drop the traffic coming from B (Fig. 10).

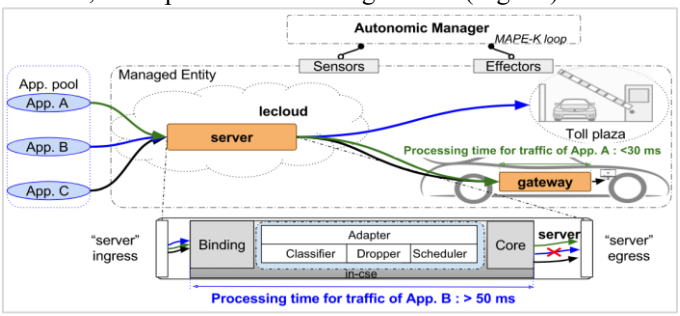

Fig. 10. Third stage of scenario

B. Evaluation of the benefits induced by the defined policies

1) Evaluation of the scheduling policy performed within the gateway

Fig. 11 presents the evolution time of the processing time of traffics A and C within the gateway. The curves represent the moving average over 50 requests. The scheduling policy was deployed around req# 464.

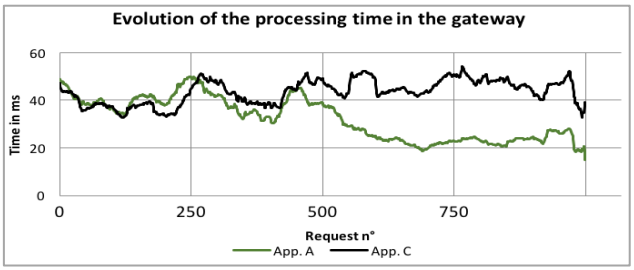

Fig. 11. Evolution of the processing time of traffics coming from applications A and C, within the gateway

We can notice three phases in the evolution of the processing time of the different traffics:

- Phase 1: [req#1 to req#464]: During this phase, the three traffics are handled without differentiation within the MW node. We can notice an average of 40ms for applications A and C. This average for the application A is considered unsatisfying by the autonomic manager (65ms + 40ms > 100ms). It triggers the second stage of the scenario.
- Phase 2 [req#465 to req#480]: The observed peak is related to the deployment of the QoS policy (Adapter + Classifier + Scheduler)
- Phase 3 [req# 481 to req#1000]: This phase corresponds to the reduction and then the stabilization of the processing time for traffic A around an average of 26ms meeting the initial requirements of application A (65ms + 26ms < 100ms).

2) Evaluation of the joint scheduling + dropping policy performed within the server

Fig. 12 presents the evolution time of the processing time of traffics A, B and C within the server.

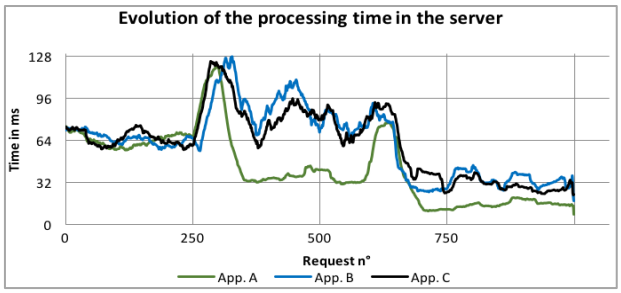

Fig. 12. Evolution of the processing time of traffics coming from applications A, B and C, within the server

We can distinguish 5 phases:

- Phase 1 [req# 1 to req# 264]: During this phase, the three traffics are handled without differentiation. We can notice an average of 65ms (for either A, B, or C). This will trigger the third stage of the scenario.
- Phase 2 [req# 265 to req# 339]: The observed peak is related to the deployment of the QoS policy implemented by a new scheduling (i.e. Adapter + Classifier + Scheduler)
- Phase 3 [req# 340 to req# 585]: This phase corresponds to the reduction and then the stabilization of the processing time for traffic A (36ms) and an increase of the processing time for traffic B and C (average around 80ms). Requirements of application A is thus satisfied. However, since application B is sensitive to delay (65ms + 80ms > 50ms), its requirements are no longer fulfilled. Therefore, the autonomic manager will augment the QoS policy in order to abandon all requests of application B if their waiting time (when leaving the Scheduler) exceeds 40ms.
- Phase 4 [req# 586 to req# 660]: The observed peak in the processing time is related to the deployment of new elements to make evolve the already deployed QoS policy (i.e. deployment the Dropper module only).
- Phase 5 [req# 661 to req# 1000]: This phase corresponds to the reduction and then the stabilization of the processing time for traffic A (15ms), traffic B (36ms but with 27% of dropped requests), and traffic C (30ms).

C. Evaluation of the costs associated to the QoS modules deployment

1) Introduction

Since we aim to dynamically handle QoS requirements of IoT applications, we need to take into account the deployment time of the QoS-oriented modules. The performance measurements presented hereinafter allows to assess the deployment time (including the activation time) of the QoS modules within the considered gateway and server MW nodes. In this test, and in order to not bias the results by additional upload time (linked to the network conditions), the QoS modules are supposed to be already present within the system hosting the MW node as illustrated in Fig. 13.

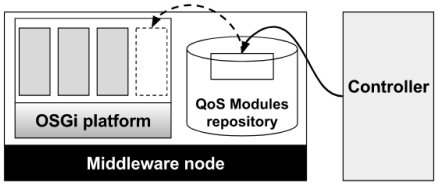

Fig. 13. Software architecture for the performance measurements

Based on this measured time, it is possible to estimate the total deployment time through this simple formula:

$$t_{commissioning(ms)} = t_{loading(ms)} + t_{deployment(ms)}$$

$$t_{commissioning(ms)} = \frac{module_{size(bit)}}{network_{throughput(bit/ms)}} + t_{deployment(ms)}$$

The "Controller" entity depicted in Fig. 13 is responsible for the deployment of modules within the OSGi platform (and eventually remotely). The deployment of every module is repeated over 1000 iterations and the results are presented and discussed in the following sections.

2) Results and Discusions

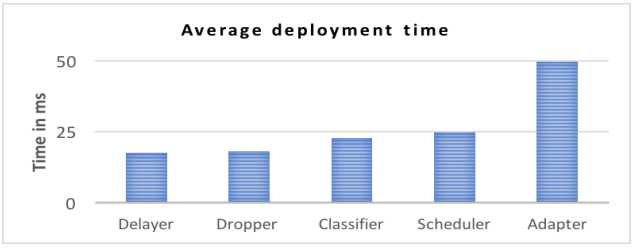

Fig. 14. Average deployment time (ms) of the QoS-oriented modules

Fig. 14 presents the average deployment time of each implemented QoS module. Among all the QoS modules, we see that the Adapter module has the highest deployment time (50ms), followed by the Scheduler (25ms), the Classifier (23ms) and finally the Delayer (18ms). Thus, the deployment time of a QoS policy is linked to the size of the involved QoS modules. Indeed, in our implementation, the size of the QoS modules are as follows: The Adapter module (i.e. OSGi Bundle) is 1Mb, the Scheduler is 12 Kb, the Classifier is 10kb, and finally the Dropper and the Delayer are both 7Kb. For example, overriding the scheduling policy will require around 100ms of deployment time.

Since the Adapter module is required for any QoS policy deployment, we could consider that the Adapter module has to be part of the initial deployment of the OM2M middleware node. Thus, only other QoS modules (that are small in size) are to be deployed during run-time depending on the QoS policy planned by the autonomic manager.

VI. Conclusion and Future Work

QoS management in the IoT is still challenging goal. In this paper, we presented one of the approaches that we envision to meet IoT applications' QoS requirements. We detailed our vision of a modular architectures and how we can use them to manage the IoT applications' QoS requirements at the middleware level. We have also presented the design and implementation of QoS oriented modules, their integration architecture within the oneM2M-compliant OM2M middleware, and the experimental evaluation of both the benefits and costs of the proposed solutions through a scenario coming from the vehicular transportation domain.

Even though, the deployment of the QoS modules into the middleware nods can be considered, this solution cannot fit all the cases. Thus, we also consider, as future work, the dynamic deployment of microservices in which QoS management mechanisms will be implemented taking advantages of both approaches (joint deployment of QoS microservices and QoS modules).

Finally, the contributions presented in this paper fall in our general approach that consists in a dynamic and autonomous end-to-end management of the QoS at the middleware level through the deployment of QoS mechanisms within the middleware nodes (in the form of modules) or outside the MW (in the form of microservices).

References

[1] oneM2M, TS-0002-V2.7.1, "Requirements", August 2016.

[2] S. Yau, and J.-P. Tsai, "A Survey of Software Design Techniques", in IEEE Transactions on software engineering, vol. 12, pp. 713-721, 1986.

[3] Razzaque M A et al. Middleware for internet of things: a survey in IEEE Internet of Things Journal, pp. 70-95, 2016

[4] 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.

[5] K.-J. Lin, N. Reijers, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu, "Building Smart M2M Applications Using the WuKong Profile Framework", in 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.

[6] W. Heinzelman, A. Murphy, H. Carvalho, M. Perillo, "Middleware to Support Sensor Network Applications", Network, IEEE, vol. 18, issue 1, pp. 6-14, 2004.

[7] 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.

[8] IBM, "Node-RED, a visual tool for wiring the internet of things", 2015.

[9] A. Banks and R. Gupta, "MQTT Version 3.1.1 Errata 01", OASIS Approved Errata, December 2015.

[10] oneM2M, TS-0010-V2.4.1, "MQTT protocol Binding", August 2016

[11] The OCEAN Mobius HomePage [online] Available : developers.iotocean.org/archives/module/mobius.

[12] The IOTDM HomePage [online] Available : wiki.opendaylight.org/view/IoTDM:Main.

[13] The IoT Oasis HomePage [online] Available: www.iotoasis.org .

[14] M. B. Alaya, Y. Banouar, T. Monteil, C. Chassot, K. Drira, "OM2M : Extensible ETSI-compliant M2M Service Platform with Self-configuration Capability", in Procedia Computer Science, vol. 32, 2014, pp. 1079-1086.

[15] The OM2M HomePage, [online] Available: www.om2m.org.

[16] The OSGi HomePage, [online] Available: www.osgi.org.

[17] The OSGi Alliance, "OSGi Core Release 6", June 2014, pp 385-387.

[18] E. Gamma, J. Vlissides, R. Johnson, R. Helm, "Design Patterns Elements of Reusable Object-Oriented Software", October 1994, pp. 157-170.

[19] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An architecture for differentiated services", RFC 2475, IETF, Dec. 1998.

[20] The CAMP Vehicle Safety Communications Consortium, DOT HS 809 859, "Vehicle Safety Communications Project Task 3 Final Report Identify Intelligent Vehicle Safety Applications Enabled by DSRC", May 2004.

[21] O. Kephart , D. M. Chess, "The vision of autonomic computing", Computer, v.36 n.1, p.41-50, January 2003