Towards Autonomic Middleware-Level Management of QoS for IoT Applications

Yassine Banouar[1,3(✉)], Saad Reddad[3], Codé Diop[2,3], and Christophe Chassot[2,3]

[1] Université de Toulouse, UPS, Toulouse, France
[2] INSA, Villeurbanne, France
[3] CNRS-LAAS, Toulouse, France
{banouar,reddad,diop,chassot}@laas.fr

Abstract. The Internet of Things is expected to bring large and promising spectrum of social goods in various domains. Several new challenges arise or are to be reconsidered within the IoT systems supporting these goods, among them the Quality of Service (QoS) issue. The goal of this paper is first to introduce our approach for an autonomic Middleware-level QoS management of IoT systems. As a contribution at the second maturity level of the autonomic computing paradigm such as defined by IBM, it is then to propose and validate, within an emulation testbed platform, a proof of concept-oriented architecture of a monitoring component allowing detecting QoS degradation symptoms. We also demonstrate the benefits that could be gained from simple network-inspired QoS-oriented adaptation actions.

Keywords: Internet of Things · Qos · Middleware · Autonomic computing · Monitoring

1 Introduction

Now, the Internet includes not only computers but also all kinds of communicating and more or less smart objects. This new extension is called *Internet of Things* (IoT), it will allow bringing a large and promising spectrum of *social goods* in various domains such as health, safety, etc. Within IoT systems, several challenges are to be considered, among them the *QoS* issue (i.e. the ability of the service to ensure non-functional properties such as bounded response time).

The QoS issue has been addressed many times in the field of the "classical" Internet [1, 2]. This issue becomes again relevant within IoT systems. While conventional services usually involve two end-hosts and intermediate routers, IoT services refer to "activities" involving much many hardware/software entities; their interconnection imposes the use of *communication middleware,* such as the open source *OM2M* platform [5], enabling discovery of connected devices, abstraction of network heterogeneity, etc. Providing such middlewares with QoS-oriented capabilities then becomes a necessity that is still under research study. In this context, the first contribution of this paper is to introduce our vision of an autonomic QoS management at the Middleware level, following the IBM autonomic computing (AC) paradigm [11]. Towards this objective, the second contribution takes place at the second level of maturity of the AC such as defined by IBM; it consists of developing monitoring solution aimed at detecting QoS degradations, helping the administrator in his/her decision to execute adaptation actions.

The rest of this paper is organized as follows: Sect. 2 presents the architecture principles towards a QoS-oriented autonomic management of the OM2M middleware. Section 3 details the functionalities and architectural principles of the proposed monitoring solution. Section 4 presents how a “proof of concept” implementation of our monitoring solution has been tested on an emulation platform, together with the benefits that could be gained from a simple network-inspired QoS-oriented action performed by the administrator when a degradation symptom is detected.

2 Architecture Principles Towards QoS-Oriented Autonomic Management of Middleware Layer for IoT Systems

A structured architectural model of an IoT system is proposed in [3, 4], which includes three levels: *Application Level, Network Level* and *Perception Level*. As subpart of the Application level, the *Middleware* level is aimed at hiding the details of various underlying technologies. In an IoT perspective, it abstracts heterogeneity of physical objects by providing homogeneous representation facilitating their handling by IoT applications. It is also aimed at offering several services such as information/services/user access rights/devices management. The contributions exposed in this paper have been done using the ETSI compliant OM2M open source middleware platform [5]. OM2M is a RESTful platform running on the top of the OSGi layer, making it modular and highly extensible via plugins, offering specific ETSI-M2M compliant *service capabilities*.

Managing QoS in dynamic IoT environment contexts is a complex task [6–10], which makes now compulsory autonomous management of QoS-oriented actions. In this context, our final goal consists of upgrading the OM2M platform following the *Autonomic Computing* model defined by IBM [11], based on the MAPE-K cycle (upper part of Fig. 1a). [11] defines five successive levels of maturity to go from a manual management of a given system to a fully autonomic system (Fig. 1b). At the second level, which is targeted in this paper, monitoring tools can be used to collect metrics from the system to detect anomalies (or “symptoms”), helping to reduce the time taken to collect and synthesize information. Human skills are however required to analyze the detected anomalies and execute corrective actions.

Several plans of management have to be considered within OM2M, each one requiring sensors/effectors in order to be monitored/(re)-configured (lower part of Fig. 1a). The first plan deals with the OM2M software: its goal is to collect metrics and to manage actions that could be performed to improve QoS. As OM2M is a JAVA-based platform, the second plan deals with the Java Virtual Machine, which offers all the required resources for OM2M execution (threads, CPU, memory, etc.). The third and fourth plans deal with computing resources and may concern the physical machine level or the virtual machine level in a cloud-based deployment.

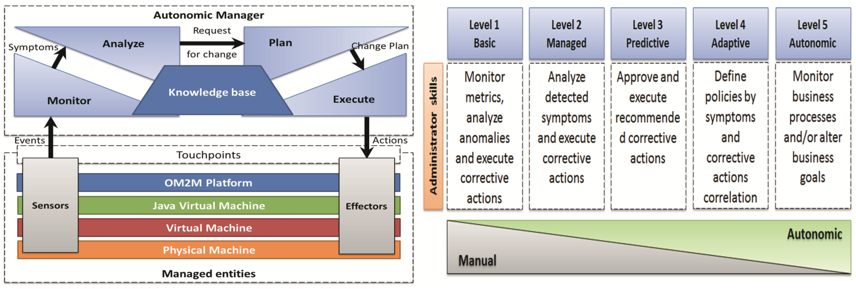

Fig. 1. (a) Autonomic architecture for OM2M QoS management – (b) Maturity levels towards Autonomic behaviour

3 Architectural Principles of the Monitoring Component

The monitoring component is aimed at *collecting metrics* (*events*) and *generating symptoms* identifying QoS degradations. It is based on two main functionalities: the *observation of the monitored system* through sensors, and the *detection of symptoms,* through events aggregation, correlation and filtering actions (Fig. 2).

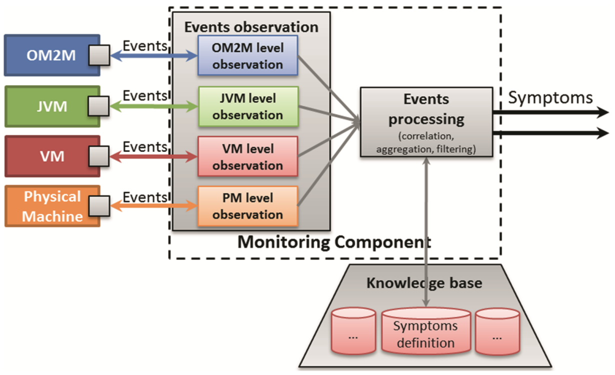

Fig. 2. Architecture of the monitoring component

The *observation function* is performed thanks to logical sensors integrated in the managed entity for events collection. Metrics are collected at four levels; at the OM2M level, such as: execution time, RTT, losses, plugin state, concurrent requests, web server size. At the JVM level (on which OM2M is running), for metrics related to the resources used by the OM2M platform (used memory, used CPU, running threads or number of loaded classes). And at the virtual/physical machine level (on which the JVM is running), with metrics related to machine state, total memory load, CPU load, used disk percentage, etc.

Starting from the events collected by the observation function, the *symptoms detection function* is aimed at detecting *patterns* identifying symptoms that have been pre registered in a knowledge base. To identify these patterns, *complex event processing*

(CEP) is a technique that allows discovering complex events, by deduction, analysis and/or correlation of elementary events. Among the different tools implementing CEP, Esper (http://www.espertech.com/products/esper.php) is an open source Framework that we have used in our study.

4 Validation of the Monitoring Component

4.1 Testbed Platform and Measurement Scenarios

Our validation approach of the monitoring component (that we claim as a basic proof of concept) is based on an *emulation testbed*, which allows testing a real OM2M platform, confronted to an emulated traffic. The emulation platform (Fig. 3) is provided as a set of web services consisting of injectors generating traffic, and of a controller whose main function is to configure the injectors following a defined emulation scenario. To avoid congestions at the sender side, injectors and controllers are launched on different machines. The traffic is sent to a BeagleBone Black gateway executing the OM2M software.

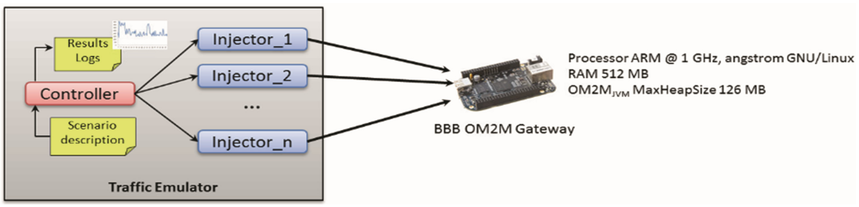

Fig. 3. Overview of the emulation platform

The first injector (Injector_1) is supposed to generate a critical applicative traffic corresponding to the periodic data from a critical sensor in a health care-oriented case study, for which an RTT of 300 ms (threshold value) is required. Due to space limits, we only consider a simple pattern consisting of four successive events indicating an increasing RTT upper than the required threshold value, the last one being twice upper the first one.

An Esper description of the pattern is provided hereafter:

$$\begin{gathered} select\ *\ from\ Event\ match\_recognize\ (measures\ A\ as\ value1,\ B\ as\ value2,\ C\ as\ value3,\ D\ as\ value4 \\ pattern\ (A\ B\ C\ D)\ define\ A\ as\ A.value > THRESHOLD,\ B\ as\ (A.value < B.value), \\ C\ as\ (B.value < C.value),\ D\ as\ (C.value < D.value)\ and\ D.value > (2 * A.value)) \end{gathered}$$

The defined scenarios (see Table 1) are aimed at studying the impact of disruptive traffics (Injectors_2 and 3) on the sensitive flow (Injector_1) for which a QoS has to be maintained. Each injector is characterized by a number of HTTP requests (R), a request method (e.g. POST, GET), a destination and a periodicity (P) in second (0 = concurrent requests). Timestamps (t_1, t_2 and t_3 (in seconds)) are collected to evaluate the evolution of the RTT and the remaining inputs.

Table 1. Scenario testbed

Scenario	Controller			Injector 1		Injector 2		Injector 3		Observed metric
	t_1	t_2	t_3	R	P	R	P	R	P	
1	0	–	–	200	0.5	–	–	–	–	$\text{RTT}_{\text{Inj\_1}}$
2	0	1	–	200	0.5	200	0.5	–	–	$\text{RTT}_{\text{Inj\_1}}$
3	0	1	1	200	0.5	200	0.5	200	0.5	$\text{RTT}_{\text{Inj\_1}}$
4	0	20	40	260	0.5	300	0	200	0	$\text{RTT}_{\text{Inj\_1}}$

4.2 Results Analysis

Scenarios results are provided on Fig. 4. Due to space limits, we focus on the evolution of the $\text{RTT}_{\text{Inj\_1}}$.

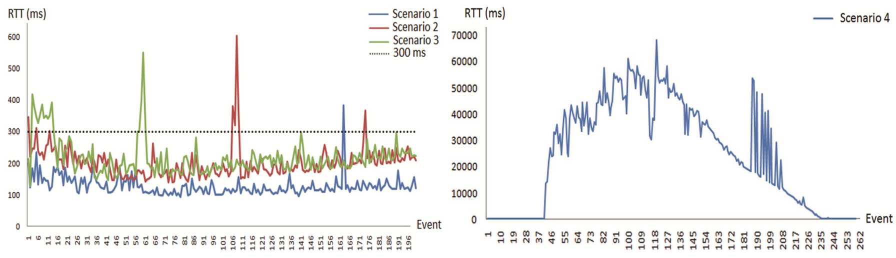

Fig. 4. (a) $\text{RTT}_{\text{Inj\_1}}$ evolution for scenarios 1, 2 and 3 – (b) $\text{RTT}_{\text{Inj\_1}}$ evolution for scenario 4

For the first three scenarios (Fig. 4a), the $\text{RTT}_{\text{Inj\_1}}$ increases slightly when adding disruptive traffic, without leading to degradation symptom detection; only some isolated violation of the required threshold may be observed. Differently, in scenario 4 (Fig. 4b), the $\text{RTT}_{\text{Inj\_1}}$ of the observed traffic reaches much higher and repeated "out of threshold" values that lead to a symptom alert (around the 40th event).

Next section describes the benefits of a simple adaptation action that could be done by the administrator, once notified of this alert.

4.3 Benefits of Adaptation Action in Response to Symptom Detection

Once notified with a QoS degradation symptom, the administrator has to apply QoS-oriented adaptation action(s). Before executing some adequate action(s), he/she has to analyse the potential causes of the observed symptom(s) and then decide about the action(s) to be performed. Let us recall that within a fully autonomic system, such *Analysis* and *Plan* steps should be done by the system itself (transparently for the administrator). These two steps being out of the scope of the AC maturity level targeted in this paper (level two), we suppose here that the administrator has to decide by him/her-self for the execution of a simple adaptation action that consists in activating/de-activating a proxy at the entry of the gateway, allowing discarding part of the incoming traffic (typically the disruptive traffic).

Figure 5 illustrates the benefits of this adaptation action supposed to be performed after the 75th event (the time taken between the symptom detection by the monitoring component - around 40th event - and the execution action of the administrator being not null). At that time, the administrator activates the proxy, in order to discard the traffic coming from the injectors 2 and 3.

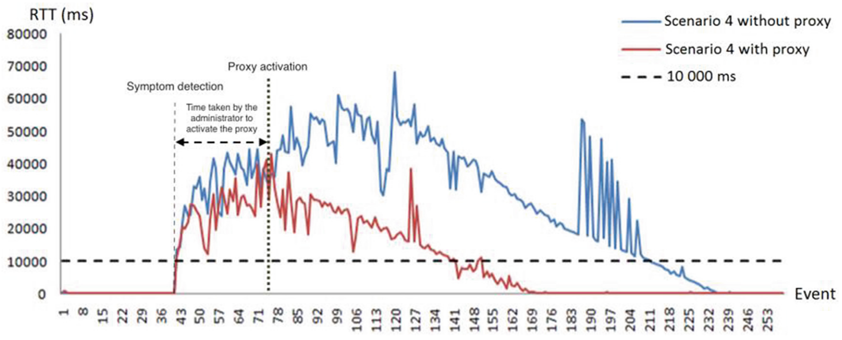

Fig. 5. Comparison of RTT_{Inj_1} without and with the proxy mechanism

Through this scenario, one can notice that the proxy activation leads to a better RTT, still being at a value much higher that the targeted threshold of 300 ms. The targeted RTT is reached around the 169th event, due to the time taken to process the disruptive traffic already in the OM2M gateway when the proxy has been activated.

5 Conclusions and Perspectives

This paper has presented our general vision for an autonomic Middleware-level QoS management of IoT systems. With the aim to target the second maturity level of the AC, the focus has been done on the Monitoring component of the AC paradigm, for which proof of concept implementation principles have been proposed and tested through an emulation platform aimed at stressing the OM2M open source platform. The performed measurements allow concluding that QoS-oriented and resources-oriented symptoms may be detected during execution of the applications (here emulated by traffic injectors), and that simple (network-inspired) adaptation actions allow improving the observed QoS-oriented.

Many perspectives are arising from this work. The enhancement of the set of more complex symptoms is a first perspective. The enhancement of the set of mechanisms to be activated is also under study: instead of activating/de-activating a traffic discarding mechanism, a current improvement of the proposed proxy allows blocking a given percentage of the traffic; similarly, we have also configured a delay-oriented proxy; other mechanisms are going to be proposed, their choice and parameterization depending on the targeted policy and the context. A current study is also to enhance the AC maturity level of our system with the aim (as a first result) to make it transparent to the administrator the step of activating manually the adaptation mechanism to be enforced when a symptom is detected. Finally, enhancing the architectural design of the AC components is also an important perspective towards the deployment of a Middleware-level QoS management system within a real IoT system.

References

1. Braden, R., et al.: Integrated Services in the Internet Architecture: An Overview. RFC 1633, June 1994
2. Black, D., et al.: An Architecture for Differentiated Services. RFC 2475, IETF, December 1998
3. Duan, R., Chen, X., Xing, T.: A QoS architecture for IoT. In: 2011 International Conference on Internet of Things (iThings/CPSCom), and 4th International Conference on Cyber, Physical and Social Computing, pp 717–720 (2011)
4. RICHCLOUD White paper: Internet of things (IOT) (2012)
5. Ben Alaya, M., Banouar, Y., Monteil, T., Chassot, C., Drira, K.: OM2M: extensible ETSI-compliant M2M service platform with self-configuration capability. Proc. Comput. Sci. **32**, 1079–1086 (2011)
6. Skorin-Kapov, L., Matijasevic, M.: Analysis of QoS requirements for e-Health services and mapping to evolved packet system QoS classes. Int. J. Telemed. Appl. (2010)
7. Jin, J., Gubbi, J., Luo, T., Palaniswami, M.: Network architecture and QoS issues in the internet of things for a smart city. In: Proceedings of the ISCIT, pp. 974–979 (2012)
8. Ling, L., Shancang, L., Shanshan, Z.: QoS-aware scheduling of services-oriented internet of things. Indus. Inform. **10**, 1497–1505 (2014)
9. Zhou, M., Ma, Y.: A modeling and computational method for QoS in IoT. In: Proceedings of Software Engineering and Service Science (ICSESS), pp. 275–279, June 2012
10. Ren, W.: QoS-aware and compromise-resilient key management scheme for heterogeneous wireless IoT. Int. J. Netw. Manag. **21**, 284–299 (2011)
11. Kephart, J., Chess, D.: The vision of autonomic comp. Computer **2003**, 41–50 (2003)