UNIVERSITÀ DI PISA
Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
Ingegneria dell'Informazione

Tesi di Dottorato di Ricerca

Design and performance evaluation of advanced QoS-enabled service-oriented architectures for the Internet of Things

Giacomo Tanganelli

Anno 2016

UNIVERSITÀ DI PISA

Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in Ingegneria dell'Informazione

Tesi di Dottorato di Ricerca

Design ad performance evaluation of advanced QoS-enabled service-oriented architectures for the Internet of Things

Autore:

Giacomo Tanganelli *Firma*__________

Relatori:

Prof. Enzo Mingozzi *Firma*_________

Anno 2016

Sommario

L'Internet delle cose (IoT) sta velocemente diventando realtà, l'abbassamento dei prezzi e l'avanzamento tecnologico nel campo dell'elettronica di consumo sono i due principali fattori trainanti. Per questo motivo, nuovi scenari di applicazione nascono ogni giorno e quindi nuove sfide da affrontare. Nel futuro saremo circondati da molti dispositivi intelligenti che monitorerranno e agiranno sull'ambiente fisico. Questi dispositivi intelligenti saranno le fondamenta per una pletora di nuove applicazioni intelligenti che forniranno agli utenti finali nuovi servizi evoluti. In questo contesto, la qualità del servizio (QoS) è stata identificata come un requisito non funzionale fondamentale per il successo dell' IoT. Infatti, nel futuro Internet delle Cose, avremo applicazioni diverse, ognuna con specifici requisiti di QoS, che necessiteranno di interagire con un insieme finito di dispositivi intelligenti ognuno con determinate capacità di QoS. Questa mappaturatra richieste e offerte dovrà essere gestira per soddisfare gli utenti finali.

Il lavoro di questa tesi si focalizza sui meccanismi che permettono di fornire la QoS in ambito IoT sfruttando un approccio inter-livello. In altre parole, il nostro obiettivo è fornire un supporto alla QoS che, da un lato, aiuti le architetture di back-end nel gestire il vasto insieme delle applicazioni IoT, in cui ogni applicazione ha dei requisiti QoS diversi, mentre dall'altro lato, vogliamo arricchire la rete di accesso aggiungendo la capacità di gestire richieste con parametri QoS direttamente sui dispositivi intelligenti.

Abbiamo analizzato le piattaforme che già forniscono un certo livello di QoS e, basandoci sullo stato dell'arte, abbiamo derivato un nuovo modello specificatamente pensato per sistemi IoT. Quindi abbiamo definito le procedure necessarie per negoziare il livello di QoS richiesto e per farlo rispettare. In particolare ci siamo concentrati sul problema della selezione dei dispositivi che si presenta quando più dispositivi possono fornire contemporaneamente lo stesso servizio.

Infine abbiamo considerato il livello di accesso fornendo diverse soluzioni atte a gestire il supporto alla QoS a diversi livelli di granularità. Abbiamo proposto una soluzione totalmente trasparente che utilizza tecniche di virtualizzazione e di proxying per differenziare tra differenti classi le applicazioni fornendo perciò un trattamento di prioritizzazione basato sulle suddette classi. Quindi siamo andati oltre e abbiamo sviluppato un sistema di QoS che si basa direttamente sul protocollo IoT chiamato Constrained Application Protocol (CoAP). Abbiamo strutturato il sistema di QoS per migliorare il paradigma denominato Observing che è estremamente importante soprattutto se consideriamo le applicazioni industriali che potrebbero ottenere un notevole beneficio da assicurazioni di tipo QoS.

Abstract

The Internet of Things (IoT) is rapidly becoming reality, the cut off prices as well as the advancement in the consumer electronic field are the two main training factor. For this reason, new application scenarios are designed every days and then new challenges that must be addressed. In the future we will be surrounded by many smart devices, which will sense and act on the physical environment. Such number of smart devices will be the building block for a plethora of new smart applications which will provide to end user new enhanced service. In this context, the Quality of Service (QoS) has been recognized as a non functional key requirement for the success of the IoT. In fact, in the future IoT, we will have different applications each one with different QoS requirements, which will need to interact with a finite set of smart device each one with its QoS capabilities. Such mapping between requested and offered QoS must be managed in order to satisfy the end users.

The work of this thesis focus on how to provide QoS for IoT in a cross-layer manner. In other words, our main goal is to provide QoS support that, on one hand, helps the back-end architecture to manage a wide set of IoT applications, each one with its QoS requirements, while, on the other hand, enhances the access network by adding QoS capabilities on top of smart devices.

We analyzed existing QoS framework and, based on the status of the art, we derive a novel model specifically tailored for IoT systems. Then we define the procedures needed to negotiate the desired QoS level and to enforce the negotiated QoS. In particular we take care of the Thing selection problem which is raised whenever more than one thing can be exploited to obtain a certain service.

Finally we considered the access network by providing different solutions to handle QoS with different grain scale. We proposed a totally transparent solution which exploits virtualization and proxying techniques to differentiate between different class of client and provide a class based prioritization schema. Then we went further by designing a QoS framework directly on top of a standard IoT protocol called Constrained Application Protocol (CoAP). We designed the QoS support to enhance the Observing paradigm which is of paramount importance especially if we consider industrial applications which might benefit from a certain level of QoS assurances.

A mio fratello e ai nostri
interminabili discorsi,
la domenica al tavolo da pranzo.

A Ilaria che mi é stata accanto,
che mi ha capito,
che mi ha tollerato,
che mi ha reso felice,
sempre.

Agli amici veri,
che ti strappano un sorriso,
anche quando sembra impossibile.

Ai miei genitori,
che mi hanno sempre
aiutato e spronato,
grazie mille.

Contents

List of Figures

List of Tables

1 Introduction

The recent advancements in embedded computing and sensor technologies are turning the Internet of Things (IoT) into reality. Many solutions commercially available today exploit networked smart objects to provide end-users with advanced services connected to the physical world. Such solutions are however often vertical, isolated, systems based on ad-hoc HW/SW realizations which are not able to cooperate with each other to share common smart object capabilities. Isolation is not the only drawback: from a software developer perspective, the lack of a common software fabric to interact with smart objects entails great limitations on software portability and maintenance [1].

To overcome such limitations, a layered horizontal approach is by far more appropriate and desirable, since it eases the integration of heterogeneous existing systems, and also facilitates the development of IoT applications based on an unified interface to a converged infrastructure. In fact, several horizontal IoT platforms have been recently designed and developed exposing standard interfaces to access smart objects. Most of these solutions are characterized by a centralized cloud-based approach, which yields the usual benefits in terms of scalability (potentially infinite computation and storage capacity), ease of maintenance, time to market and low development costs. On the other hand, running an IoT platform in a cloud infrastructure deployed far from where the smart objects are physically located may result in a sub-optimal choice for many classes of IoT applications, e.g., Machine-to-Machine (M2M) ones, which typically have a limited scope in time and space (data need to be processed only when and where it is generated), require simple and repetitive closed-loop interactions, and often must respond with stringent latency guarantees to avoid service disruption.

For this reason new cloud-based architectural designs are being considered to account for the distinctive features and characteristics of IoT components, i.e., support of real-time QoS-aware interactions and collection of fresh and accurate context information. In particular, such novel hierarchical architectures extend the cloud to the edge through a capillary infrastructure that moves computing and storage services closer to sensors and actuators. A notable example of these new approaches is the so called *fog* computing paradigm [2][3].

In this work we design an all-in-one QoS solution for horizontal IoT architecture. Our main goal is to provide QoS support that, on one hand, helps the back-end architecture to manage a wide set of IoT applications, each one with its QoS requirements, while, on the other hand, enhances the access network by adding QoS capabilities.

In order to provide QoS support for the back-end architecture we must consider that the wide set of future IoT applications will raise new challenges. As an example, latency (besides dependability) is a critical factor for applications such as real-time sensor monitoring in personal health-care or public safety systems [4]. On the other hand, other applications like, e.g., road traffic management applications for urban mobility, though less sensitive to delay bounds, may nevertheless benefit from receiving some form of soft real-time treatment, at least for a subset of their provided services (e.g., urgent alert notifications) [5]. Moreover, applications involving streaming of multimedia content like video surveillance that consume higher bandwidth will require full support from the platform not only to guarantee an acceptable level of service but also to avoid saturation and waste of network resources.

In this context, efficient support for heterogeneous QoS requirements is a non-trivial challenge. The number of shared communication and computational resources that will be highly heterogeneous and constrained in a variety of manners represent a complex field to operate: smart things are constrained devices in terms of computation, storage and energy (since they may be battery operated). Moreover, things are more volatile and dynamic: continuous changing context and intermittent availability are the two main factors that differentiate IoT services supported by smart objects, i.e., battery powered devices perhaps in movement, from traditional services running on fixed powerful hosts.

However the integration of different sensing and actuating systems into one single service infrastructure allows applications to benefit from high IoT service availability: different equivalent smart things may provide similar services - called equivalent services in the rest of this work - with common functionalities but different QoS and cost (e.g., different smart cameras may provide, if appropriately steered, the same view of a given area from different directions).

In order to handle this heterogeneity a comprehensive QoS management framework is needed. The framework must operate at different time scales, i.e., resource provisioning and runtime, and comprises a general QoS model, SLA-based negotiation and admission control, and optimized resource provisioning.

Thus we propose a new QoS framework, which could be integrated into existing IoT architectures, that try to achieve the following objectives: (i) guaranteeing scalability over large IoT deployments characterized by heterogeneous devices, (ii) supporting M2M applications with a wide range of QoS requirements, (iii) exploiting the large number of smart objects that are expected to be connected providing equivalent services in order to allow the definition of efficient resource allocation algorithms.

To support large IoT deployments we focus on architectures which follows the fog paradigm by implementing a distributed architecture made of interconnected gateways.

Support for heterogeneous applications is guaranteed through Service Level Agreements that allow the negotiation of the desired QoS including also hard real-time requirements. A SLA negotiation framework is included in the design to expose to applications an interface to negotiate the desired QoS level.

Finally, due to the unique features exposed by smart things the standard algorithms used to handle resource allocation are not suitable. For this reason we move forward by developing a novel efficient service selection algorithm which take into account the energy consumption and perform the selection of things matching application requests, whilst guaranteeing to meet the respective QoS requirements.

Taking into account the access network we focus our research on the application layer. In recent years a considerable effort has been carried out to integrate the IoT into the Web. The Web of Things (WoT) envisions a full integration of smart objects within the World Wide Web: objects implementing a REST interface are seamlessly accessed by applications through the same successful RESTful paradigm adopted to access web resources [6]. With the aim of making the WoT real the IETF Constrained RESTful Environments (CoRE) Working Group has recently specified the Constrained Application Protocol, CoAP [7]. Defined to bring the RESTful paradigm to constrained devices, CoAP is not just a lightweight version of HTTP with reduced overhead and limited complexity in order to fit the limited capabilities of constrained devices. While preserving the original paradigm, CoAP extends HTTP by introducing a set of functionalities to provide features specifically tailored to Machine-to-Machine (M2M) applications. Among them, it is worth to mention the observing operation, which provides applications with the ability to specify an interest on the status of a resource in order to obtain unsolicited updates whenever the latter changes [8].

The structure of current deployments is, however, far from supporting the long-term WoT evolution. The latter will need large deployments that can efficiently satisfy concurrent requests also implementing security and QoS features to enforce access control and service differentiation, respectively. In this context the variety of applications exploiting smart objects for heterogeneous purposes will demand support for easy customization to implement third-party functionalities, e.g., for protocol translation or custom features. In fact, the embedded computing systems implementing smart objects may have several constraints in terms of computation, storage, communication, and time operation, if battery powered. To scale to the expected levels of concurrency and implement additional features, intermediate devices are needed to appropriately manage concurrency of access.

Thus we propose a framework to address the challenges of large-scale WoT deployments. The proposed solution aims, on one side, at ensuring scalability to multiple concurrent requests, and, on the other, at guaranteeing expandability through the implementation of QoS policies for request differentiation. Deployed on intermediate devices, e.g., gateways, the proposed solution makes use of the functionalities offered by CoAP proxies to transparently isolate clients, i.e., applications, from servers. Virtualization is included in the design to allow implementation of future custom functionalities that can be introduced to manage the requests from one (or a group of) application(s).

In fact, we can exploit virtualization techniques to differentiate applications by exposing, through a virtualized gateway, a different set of smart things. This solution is totally transparent and powerful. However, to obtain a fine grained control of QoS capabilities we have to break this rule and introduce software logic on both sides of the interactions.

As an example suppose to have a smart thing, e.g. a temperature sensor, that can be queried by two different applications. The first application wants a fresh value every 5 seconds while the latter one just wants to store data for later historical processing, so a fresh message every 10 minutes is enough.

Obviously both applications should receive updates based on their needs so such different requirements must be negotiated between the applications and the smart device. On the other hand, the smart devices will be, usually, a constrained devices which cannot handle complex interactions.

To address this issue we propose a solution that leverages on gateways to add QoS support directly between smart devices and applications. The role of each gateways is to limit the number of message exchanged by the constrained devices and to manage complex interactions on behalf of constrained devices. Moreover, each gateway may acts as a point of access to a set of smart devices and it is the best candidate for hosting security and QoS managers.

Another point to take into account is the meaning of "freshness". Constrained devices usually work in Low power Lossy Networks (LLN) where delays may occur due to congestion more often then in traditional networks. Such delays affect the freshness of the information dispatched to the customers by the gateway.

Coming back to the previous example, an application which requires data every 5 seconds may not be interested anymore if the information that he receives is older that 5 seconds. This approach is outlined also by the Data Distribution System (DDS) architecture [9], where clients specify the maximum acceptable delay from the time the data is produced until the time the data is inserted in the receiver's application cache. The gateway that manages the interactions must take delays into account in order to avoid the dispatching of older information according to clients needs.

In order to overcome this issues we propose a solution to integrate the QoS support over the CoAP protocol. In particular we developed a standard solution that enhance the Observing feature of CoAP [8] in order to provide to clients notifications based on their requirements. To achieve this we partially exploited the CoRE Interfaces draft [10] for the negotiation phase, while we leverage on the CoCoA work [11] to monitor continuously the delays in the LLN network in order to avoid the dispatching of old information.

Finally we also consider the IoT field from the prospective of the developer. We develop a CoAP library called CoAPthon. CoAPthon is specifically designed to offer software developers an easy-to-use programming interface that can simplify application development. Its goal is to provide a tool for fast development of IoT application and rapid prototyping of IoT systems. In fact, CoAPthon is fully implemented in Python in order to exploit its simple easy-to-learn syntax that looks very similar to pseudo code, and the built-in portability of the language. Initially proposed as a teaching language, Python has evolved today into a general-purpose language suitable for easy development of real-world applications [12], also already adopted in the context of IoT applications in both academic projects [13] and real deployments [14]. Its portability, instead, allows IoT applications to run without modifications on heterogeneous embedded systems. Such requirement is mandatory to handle the large heterogeneity of architectures and capabilities of systems adopted today for prototyping and small-scale deployments, such as UDOO [15], Raspberry PI [16], and Arduino YUN [17]. To the best of our knowledge, CoAPthon is the first full-fledged CoAP library implemented in Python.

The rest of the manuscript is organized as follow: in Chapter 2 we present a generic architecture (called Things as a Service) which has been used as the reference architecture to integrate the QoS support into existing IoT frameworks. Chapter 3 provides a detailed description of the negotiation framework involved between the reference architecture and applications. In Chapter 4 we present the two-step QoS procedure used to reserve and allocate requests to resources while in 5 the detail of the resource reservation algorithm is presented. Then in Chapter 6 we start to analyze the access network introducing our virtualization framework for the Web of Things. Chapter 7 go further by taking into account the integration of the QoS support on top of the CoAP protocol while in Chapter 8 we present our python implementation of the CoAP protocol. Finally Chapter 9 depicts the conclusions of our work.

Part I

QoS in the Fog layer

2

Things as a Service Architecture

The integration of existing heterogeneous vertical M2M systems into a new horizontal platform has been recognized as a key factor to boost the development of the Internet of Things vision. In this scenario, many different project aim at developing a framework to enable integration of different M2M systems exposing physical objects to applications through a novel service-oriented interface, the Things as a Service model. In this chapter we present a generic Thing as a Service architecture which has been used as the reference architecture for the work presented in Chapter 3, Chapter 4 and Chapter 5.

We started with a deep analysis of existing reference models and architectures for IoT, then we derive our architecture which can be seen as an abstract model for all the existing IoT architecture. In particular we design a distributed framework composed by different interconnected gateways which has been also referred in the literature as fog computing [2].

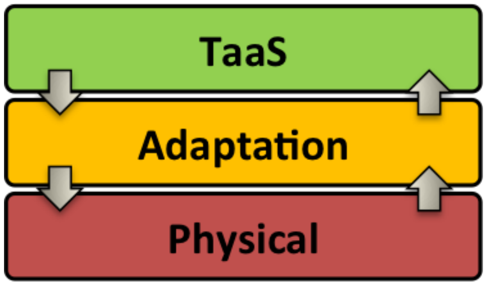

Figure 2.1: Functional model.

Each gateway is an independent entity and runs the overall stack as in Fig. 2.1. The Functional model follows a layered approach composed by three layers: the Thing as a Service layer (TaaS hereafter for short), the Adaptation layer and the Physical layer, respectively. This layered approach as been chosen to be as more general as possible on one side, but on the other hand to separate functionalities in order to better describe capabilities.

The Adaptation layers has been defined to enable the integration and access of different existing IoT/M2M systems. It guarantees transparent access to physical objects regardless of their physical model or their location. Differences among the physical systems are conformed at this layer, which exposes a common interface to the TaaS layer. This interface is deployed as a set of APIs, which are used by the TaaS layer to access the functionalities offered by IoT/M2M systems in a uniform manner.

The TaaS layer, by definition, enables the service layer to access things as a service. TaaS is implemented in a distributed manner: each gateway runs a TaaS local component, which connects to its peers to provide access to things regardless of their location. An application requiring access to one thing interacts with its TaaS local component, which represents its unique interface towards the things. The local component is then responsible for accessing the thing through its own Adaptation layer, if the thing is connected to the local network, or for forwarding the service request to the TaaS local component of the gateway where the thing is connected. Instead of providing direct access to the uniform interface provided by the Adaptation layer, TaaS is included in the design not only to provide location independent access but also to enrich data with context information in order to allow context-aware discovery and access.

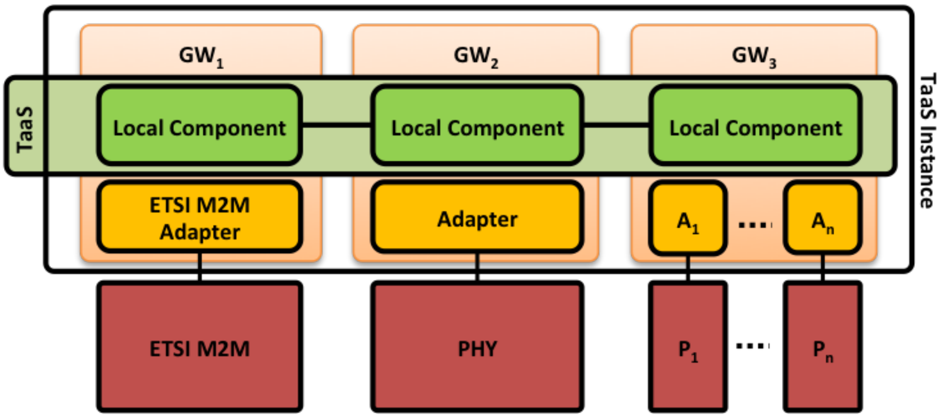

Figure 2.2: Functional model instance.

As already pointed out gateways are interconnected as in Fig. 2.2 forming the so called instance. In this way, when an application interacts with the TaaS layer deployed on a gateway, it can access to all the IoT/M2M systems within the instance regardless where such system is physically connected.

Finally, let us introduce the following assumptions:

- A thing service is uniquely identified by a thingServiceID (TSID).
- A Thing is uniquely identified by a thingID (TID).

Specific requirement of the proposed platform is the exploitation of the possibilities offered by equivalent things. As a result of the integration of different systems, large IoT networks are expected to be characterized by a large number of equivalent things, which can potentially provide the same services.

3 QoS Negotiation in Things as a Service architectures

Quality of Service support is a non-functional requirement of paramount importance for applications with stringent requirements which will be common in the M2M field. In this chapter we present our negotiation framework which allows Machine-to-Machine (M2M) applications to negotiate the required Service Level Agreement. The goal is to expose to applications a standard interface which can be exploited to negotiate the desired QoS selecting one of the service classes defined by the framework.

For an overview of Things as a Service architectures we refer the reader to Chapter 2.

3.1 Motivation

Quality of Service support has been identified as a key non-functional requirement to enable many IoT-related application scenarios. Although several proposals have been presented in order to enable end-to-end QoS in constrained environments, all these works focus on a particular technology or address a specific sub-problem and do not propose a solution to handle the problem entirely. An example is the field of Wireless Sensor Networks (WSN) in which many models for QoS have been developed for the MAC layer [18], [19] or the routing protocol [20], [21]. The future IoT world, however, will go beyond current WSNs with a new generation of devices such as actuators or smart cameras that differ significantly from traditional sensors.

Several proposals aimed at introducing QoS support have been proposed for Service Oriented Architectures (SOA) where QoS is a crucial requirement. A first QoS framework for SOA is presented in [22], where applications request services with certain requirements and the framework manages to find a possible allocation, among the set of registered service, in order to fulfill QoS requirements. In [23] authors propose a QoS framework supporting also Real-Time requirements in a distributed heterogeneous environment. Approaches defined in the context of SOA systems lack of support for constrained devices that introduce new non-trivial issues and would require significant modifications.

To the best of our knowledge, a first attempt to address such issues has been done in [24], where the use of web services for sensors integration is proposed. However, this approach aims at implementing a service-oriented middleware directly on the nodes, which is not always feasible due to their constrained environment. To overcome these limitations, in [25] an adaptable middleware is proposed; the middleware functionalities can be configured to reduce their complexity in case of constrained devices such as sensors. The proposed solution exposes a SOA interface to applications in which a flexible QoS support is provided by means of Service Level Agreements (SLAs) between the applications and the middleware. The solution proposed, however, is specifically tailored to WSNs.

In the context of distributed real-time systems, in [26] a framework for dynamic resource allocation and re-distribution is presented, however it lacks of admission control functionalities that are important especially in constraint environments. To partially overcome this issue, authors of [27] propose a middleware that implements an admission control and a load balancer. The latter in particular is responsible for optimizing resource allocation at run time by migrating tasks between processors, if necessary.

We saw the lack of a uniform middleware which try to address the following requirements:

- Provide support for negotiating the QoS required by applications.
- Not grant the service to applications not reaching an agreement during negotiation.
- Provide a re-negotiation procedure which might be requested by the framework in order to handle internal status changes.
- The framework shall be based on open technologies and standard protocols.
- A common set of predefined Service Level Agreement (SLA) templates shall be included in the system in order to enable the framework to consider a wide set of QoS requirements.
- Time critical services shall be supported with the definition of a real-time class of service.

3.2 QoS Model

In order to guarantee a QoS that can be negotiated by applications, a uniform QoS model has to be defined first. The categorization of the requirements is necessary to derive a classification adopted by the system to support applications efficiently. In this following we first provide an overview of the state of the art on QoS models, and then we present the classification derived for our framework.

3.2.1 Overview of existing QoS Models

A QoS service model and relate requirements for M2M applications was developed in [28] and [29] with specific reference to cellular networks as access technology. In particular, authors of [29] group M2M applications into five categories: mobile streaming, smart metering, regular monitoring, emergency alerting, and mobile POS (Point Of Sales). Each category poses different QoS requirements to the underlying network:

- Mobile streaming traffic involves continuous video transmission at high data rate with lower priority than other traffic. On one hand, video requires high bandwidth, soft real-time delivery and low jitter. On the other hand, the traffic is error tolerant.
- Smart metering traffic is characterized by large sporadic burst of packets with a request-response pattern. Its priority is low and the transmission can be rejected in case of network congestion. Its transmission has to be reliable but the traffic is delay-tolerant.
- Regular Monitoring traffic is characterized by small periodic packets(the period is in the order of seconds). Its priority is low and it does not have real-time requirements. Transmission reliability, instead, is critical.
- Emergency Alerting traffic is the most critical category.It is characterized by bursts of data which require the highest priority. The packet size is not predictable and a real-time and reliable transmission is needed.
- Mobile POS traffic is characterized by bursts of data with low priority. Real-time transmission is not required but reliability is a crucial requirement.

Starting from this traffic categorization, the authors uniform QoS requirements in order to cover both H2H and M2M services. The categorization is based on the main features of three types of services: conversational, data transferring and emergency alarming, characterized respectively by real-time transmission, data accuracy and trans- mission priority. Based on this three macro types seven service categorizations ranging from real-time to best-effort service have been developed.

In [30] three service models are defined based on the following factors: interactivity, delay and criticality: Open Service Model, interactive, non real-time and non mission-critical; Supple Service Model, interactive or non-interactive (according to the user subscription), soft real-time and mission-critical; Complete Service Model, non interactive (continuous flow of data), hard/soft real-time and mission-critical.

Since the framework takes into account different heterogeneous scenarios it is important to consider a wide set of QoS requirements. However, M2M applications have their core functionalities relying on sensors and actuators which are constrained devices in terms of computational and communication capabilities. In the field of Wireless Sensor Networks (WSN) several studies have been carried on to highlight application QoS requirements.

In [31] a QoS management system with requirements designed specifically for WSNs is presented. In particular, energy consumption is considered as it is one of the most important QoS metrics for WSN. Different sensors, due to their own characteristics, have different requirements: battery powered sensors need a reduced data sampling frequency while sensors without energy limitations can use a higher data sampling frequency. However, in order to have a fine grained characterization of sensors, other capabilities are also taken into consideration besides energy operation. In particular, a sensor can be single or redundant. A single sensor can be polled by different sources at the same time; in this case the middleware has to manage the requests with a certain priority following the set of QoS metrics. A redundant sensor, instead, is a virtual sensor whose information is provided by a group of physical sensors which can provide an equivalent information. In this case, the middleware can issue the request to any physical sensor belonging to this group thus applying a load balancing policy. Another differentiation is related to sensor data: on one hand, multimedia sensors have strict delay requirements but are packet loss tolerant, on the other hand monitoring sensors do not have stringent real-time requirements but rely on reliable transmission. Eventually, the authors present a categorization based on sensor transmission type: Event driven, Query driven, and Continuous. Each category is mapped to different QoS requirements:

- Event driven: Medium access delay, Reliability, Energy consumption, Flexibility;
- Query Driven: Medium access delay, Reliability, Energy consumption, Flexibility;
- Continuous: Collision rate, Energy consumption, Interference/ Concurrency.

In [32] the authors categorize applications in three major classes with different requirements: inquiry tasks, control tasks and monitoring tasks. Inquiry tasks require service timeliness and reliability. Monitoring tasks require reliability but the service is delay tolerant. A three-layer taxonomy is proposed: the Application and Service Layer which contains all services, the Network Layer which manages network functionalities and provides QoS support, and the Perception Layer, which is used for system monitoring. At the application layer, the QoS is intended by the user point of view and the focus is on Service Time, Service Delay, Service Accuracy, Service Load and Service Priority. At the network layer, QoS requirements are related with the network itself, but, in general, main indicators are: bandwidth, delay, packet loss rate and jitter. A method for passing QoS requirements from the upper layer (customers' requirements) to the lower layer (resource allocation and scheduling) is proposed. QoS communication and translation is another critical point: QoS requirements are different at each layer and depend on the corresponding level of abstraction. Authors partially overcome this problem by adopting a cross-layer approach and dividing services into four classes: Control, Guaranteed Service; Query, Guaranteed Service/Differentiated Service; Real-Time monitoring, Differentiated Services; Non Real-Time monitoring, Best Effort. Network Layer and Perception Layer use a QoS broker which is responsible for adapting QoS requirements received from the Application Layer.

3.2.2 Service Classes

A significant trend emerges from the analysis of existing QoS models: the wide-range of application requirements is handled by means of a very detailed classification that results in numerous service classes. These approaches necessarily increase the complexity of the infrastructure without fully satisfying M2M applications which often require ad-hoc QoS assurances.

For this reason, in the our framework a simple schema composed by three classes of services has been adopted, in order to reduce the complexity of platform management functionalities. At the same time, applications are allowed to customize their QoS requirements through a dynamic negotiation procedure. The three classes of services adopted are the following: Real-time, Assured services and Best-effort.

The Real-time class is designed for applications with hard response time requirements where timing responses are usually mission-critical, e.g., surveillance alarm system, health-care monitoring, industrial control. The negotiation phase is based on parameters, such as response time or service period, expressed in a deterministic manner. The platform must respect the QoS guarantees provided to this class of applications strictly.

The Assured services class instead is for applications with soft response time requirements. These applications usually tolerate some out-of-contract interaction, for this reason the negotiation procedure is based on probabilistic requirements. This class can be used by interactive application - ticketing or user information gathering – or may be used by tracking applications for logistic.

Finally, the Best-effort class is used by applications that do not require any guarantee such as an application for historical data collection.

3.3 Negotiation Framework

The framework has to allow applications to negotiate QoS through a standard protocol. The literature about services and, in general, Service Oriented Architecture (SOA) is rich and can provide standard solutions. In particular, a key feature in SOA systems is the service negotiation procedure. The WS-Agreement [33] and WS-Agreement-Negotiation [34] are the de-facto standards for SLA agreement negotiating, establishing and managing in the Web Service field. It is worth to mention that the WS-Agreement and the WS-Agreement Negotiation standards are already implemented by the WSAG4J [34] project. The implementation is Java-based, it is publicly available, and the code is well documented and stable.

The structure of the QoS framework is illustrated in Fig. 3.1. QoS negotiation capabilities are provided to applications through a standard interface which has been implemented by means of the WS-Agreement-Negotiation protocol.

Two main advantages characterize the WS-Agreement-Negotiation protocol: flexibility and interoperability. QoS support provided by the framework goes beyond the classic approach adopted in SOA architectures, i.e., QoS functionalities have to take into account the characteristics of the things and the unique requirements of M2M applications. To this aim, a flexible negotiation interface is needed to support future technologies with requirements that might not be already defined. On the other hand, the use of a standard protocol for QoS negotiation between the applications and the TaaS layers assures interoperability by definition while guaranteeing also a higher level of implementation efficiency.

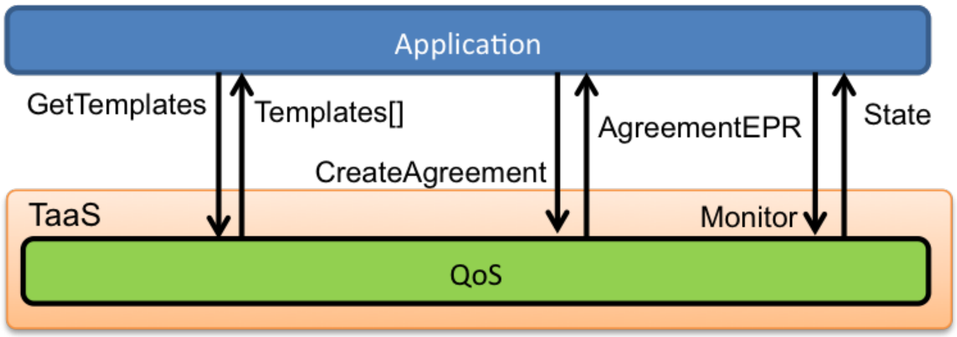

Figure 3.1: QoS framework with negotiation interactions.

In detail, the WS-Agreement protocol defines the message exchange by two end-points in order to create a service level agreement – see Fig. 3.1. In the first step, the service consumer requests all the available templates from the service provider. The consumer selects a template and creates a new agreement offer. Then, it sends the offer to the service provider to create a new agreement. The service provider replies with a confirmation or a rejection message, the confirmation contains an Agreement Endpoint Reference (AgreementEPR) generated to uniquely identify the committed agreement. An offer describes the service as well as the guarantees required for each service. Offers, templates and, in general, agreements are defined by an XML document with a specific schema outlined in Fig. 3.2.

An agreement schema is composed by two main parts: Context and Terms Compositor. The Context provides information on both consumer and provider and optionally an expiration time that defines how long an agreement (and associated services) is valid. One or more Terms Compositor are specified to describe the services. The Terms Compositor is used to structure all the different terms related to each service by creating a so called Terms Tree. It is important to highlight that a Terms Compositor can contain zero or more other Terms Compositors. Each Terms Compositor is composed by Service Terms, a Service Term describes the service and gives a pointer to reference it and, finally, defines and evaluates the guarantees of the WS-Agreement. It is worth to note that multiple Service Terms can describe a single service in an agreement. In fact, each Service Term describes a different aspect of the service.

The WS-Agreement Negotiation standard is an improvement built on top of the WS-Agreement. It gives to the consumer and provider, involved in the process of establishing an agreement, the capability of negotiating by means of offer (as the basic WS-Agreement) and counter offer.

In order to enable QoS negotiation within the platform, specific templates are defined. As an example, Fig. 3.3 shows the template included for Thing Service negotiation between an application and the TaaS layer. This template, compliant with the WS-Agreement specifications, defines a specific schema to describe the Service Description

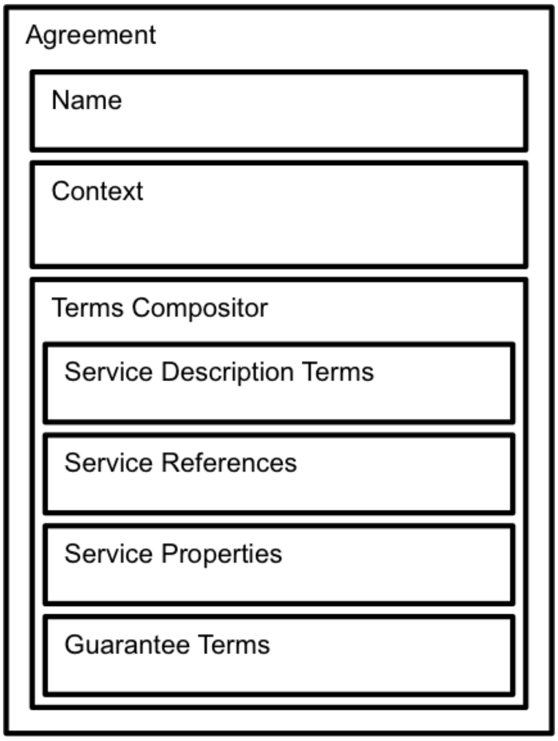

Figure 3.2: Agreement schema.

Terms required for the Thing Services invocation. The template follows the structure of the WS- Agreement standard: a Context section (<wsag:Context> tag) which contains the template name and the template id followed by a Service Description Term section (<wsag:ServiceDescriptionTerm> tag) which defines the terms of the Thing Service (<TSA:ThingService> tag). This section, in turn, includes the transaction ID needed to identify the Thing Service (<TSA:Definition> tag) and the QoS parameters, (<TSA:QoS> tag). In this example, a simple set of QoS parameters which can be included are defined:

- MaxResponseTime, used to specify the Response Time of the Thing Service. In detail, it indicates the maximum delay between a Thing Service invocation and its response, measured at the service layer.
- MinAvailability, used to specify the availability of the Thing Service.This parameter is associated to each Thing Service and is, in principle, a static parameter. However, since the environment can change unpredictably the QoSMonitoring functionality must control and adjust this parameter continuously.
- MaxRate, is the maximum rate a Service can invoke the thing service, in other terms, a minimum inter-request time between two different requests from the same service.

```xml
<?xml version="1.0" encoding="UTF−8"?>
<wsag:Template wsag:TemplateId="1"
xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws−agreement">
  <wsag:Name>Framework−Template</wsag:Name>
  <wsag:Context>
    <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
    <wsag:TemplateId>1</wsag:TemplateId>
    <wsag:TemplateName>Framework−Template</wsag:TemplateName>
  </wsag:Context>
  <wsag:Terms>
    <wsag:All>
      <wsag:ServiceDescriptionTerm wsag:Name="THING"
        wsag:ServiceName="THINGSERVICE">
        <TSA:ThingService
          xmlns:TSA="http://iet.unipi.it/schemas/TSA">
          <TSA:Definition>
            <TSA:transactionID>
              $TRANSACTIONID
            </TSA:transactionID>
          </TSA:Definition>
          <TSA:QoS>
            <TSA:MaxResponseTime>
              $MAXRESPONSETIME
            </TSA:MaxResponseTime>
            <TSA:MinAvailability>
              $MINAVAILABILITY
            </TSA:MinAvailability>
            <TSA:MaxRate>
              $MAXRATE
            </TSA:MaxRate>
          </TSA:QoS>
        </TSA:ThingService>
      </wsag:ServiceDescriptionTerm>
    </wsag:All>
  </wsag:Terms>
</wsag:Template>
```

Figure 3.3: Service Layer – TaaS Layer negotiation template.

3.4 Conclusions

In this chapter the QoS negotiation framework which allows applications to negotiate the desired QoS of the things exposed as a service is presented. A QoS model for M2M application is derived from the existing QoS models to be used within an open negotiation framework integrated in the system as an interface exposed to applications for QoS negotiation. Considered the layered architecture of the reference platform, the same open and standard interface can be adopted in order to extend the proposed framework to any existing platform following a standard integration procedure.

4

QoS Reservation and Allocation: a differentiated time-scale approach

Let's start by considering the reference architecture presented in Chapter 2 with the QoS negotiation framework outlined in Chapter 3. The QoS negotiation framework alone is obviously not enough to enforce and monitor the negotiated QoS requirements. To this aim a pervasive QoS framework must be included in the architecture. Thus, once the negotiation phase is performed, a SLA is established and applications could invoke thing services with the negotiated QoS level. However, in order to accept or deny a specific request the framework must verify if the requirements can be fulfill. For this reason we need a specific QoS procedure which is involved during the negotiation phase but also during the invocation phase. In detail, the framework must reserve resources during the negotiation phase and select the thing service at runtime among the possible set of equivalent thing services in order to fulfill the committed agreement. The latter function is particularly relevant in the IoT field, due to the unique characteristics of smart thing, a thing may be connected during the first phase but at runtime in the invocation phase it may be disconnected. To solve this issue we design a two phase procedure called Reservation and Allocation respectively.

4.1 The two-phase QoS Procedure

In our QoS framework we propose a two-phase procedure, namely, reservation and allocation. The reservation phase is handled by a sub-component called QoSBroker. The QoSBroker manages the QoS negotiation, performs admission control and, most importantly, manages resource reservation by exploiting equivalent thing services. It also generates Agreement End Point References (AEPRs) to authorize thing service invocation. The allocation phase, instead, is managed by another subcomponent called QoSDispatcher. The QoSDispatcher performs allocation of resources at time of thing service invocation. The QoSDispatcher can optimize the allocation by means of a number of parameters, e.g., in terms of energy efficiency. The reservation and allocation procedures are tightly connected. However, while the allocation procedure may be involved in each thing service invocation, the reservation procedure is executed only once at time of negotiation.

To avoid data inconsistencies, race conditions, and long response times, which can affect functionalities implemented in a distributed manner over large deployments, critical system functions are provided through a centralized point of decision, which however can be implemented in a distributed manner for scalability and resiliency over a subset of nodes. For the sake of simplicity, we assume hereafter that this functionality is provided by a single gateway called Designated GW. The choice of which GW become the Designated GW is out of the scope of this work, however we can assume, without lack of generality, that a distributed election procedure is adopted between gateways.

To reflect this design the QoSBroker and the QoSDispatcher are divided into two different sub-component with different scope: local and global, respectively. In the reservation phase we rely on a QoSGlobalBroker and on a set of QoSLocalBrokers. The former is one for TaaS instance residing on the Designated GW, while one instance of the latter is deployed in every gateway. The same approach is adopted for the QoSDispatcher (QoSGlobalDispatcher and a set of QoSLocalDispatchers). The local components manage, in term of QoS, the thing services provided by things that are directly connected to the gateway where the component is deployed. The global components, instead, have a global view of all the thing services available in the TaaS instance and are involved only when a global view is required. The overall architecture is shown in Fig. 4.1. Below we provide a detailed description of the two main procedures that are involved to provide QoS support to applications.

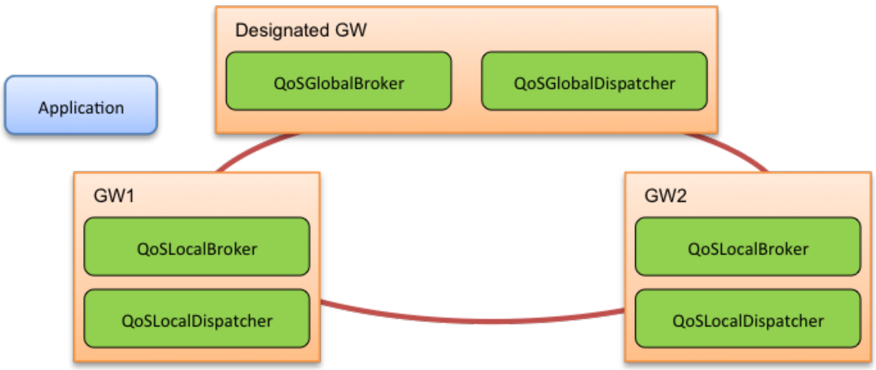

Figure 4.1: Deploy diagram.

4.2 Reservation

The reservation procedure is involved when an application wants to negotiate a set of thing services each one with certain QoS requirements. The applications can access thing services from any gateway that is part of the TaaS instance; however, from the TaaS point of view, one and only one of the gateways is the application's gateway – GW1 in sequence diagrams. First of all, the application sends an agreement request to its gateway. The agreement request encapsulates the TSID required by the application. The request is handled by the QoSLocalBroker that forwards it to the QoSGlobalBroker. The QoSGlobalBroker validates the agreement request and, if it is feasible, reserves resources for the application, performing so the admission control functionality. Then the QoSGlobalBroker replies with the AEPR, which is also stored by the QoSLocalBroker of the GW1. The AEPR is forwarded back to the application in order to enable the application to invoke the negotiated thing services. For a detailed sequence diagram see Fig. 4.2.

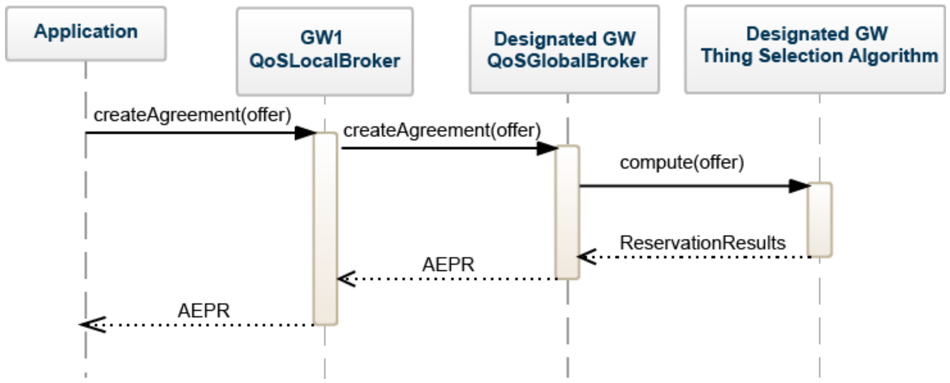

Figure 4.2: Reservation.

It is worth to mention that, in this phase, we do not perform the mapping between thing services and things. In fact, the time between the negotiation and the invocation phase is unknown, thus if we perform the selection in this phase this might result in a suboptimal allocation or incur in unpredictable errors due to the highly dynamic environment. However, the QoSGlobalBroker has the view of all the committed agreements, so it checks if the new agreement can be satisfied without violating previously accepted agreements. In other words, the QoSGlobalBroker checks if there is at least one allocation schema that can be adopted in order to fulfill the requirements of every agreement plus the new one. If the answer is positive, the new agreement is accepted; otherwise, the agreement is rejected and the application is notified. The application can obviously start a new negotiation phase with less stringent QoS requirements.

4.3 Allocation

The allocation procedure is performed every time an application invokes thing services. Applications can invoke thing services several times, thus the selected things can change between two different invocations. However, there is a drawback in this approach, which is the overhead associated to each invocation. In order to overcome this limitation, we can perform the allocation phase only every γ invocations reducing the computational overhead. Thus, there is a trade-off, modeled by the γ parameter, between the optimal selection of things and the overhead associate to each allocation. To the sake of simplicity in the rest of the paper we assume $\gamma = 1$, however the same considerations can be written for larger values of γ.

Due the distributed nature of the TaaS, we can have two different types of allocation scenarios: local allocation and global allocation. The dispatcher uses the list of equivalent thing in order to discriminate between the two different procedures. The local allocation is adopted when all the thing services requested are provided only by things directly attached to the same gateways of the application. In this case, shown in Fig. 4.3, the overall process starts and ends in the local gateway without any external interaction. For the sake of explanation, we consider the case in which an application asks only for one thing service. First of all, the application invokes the thing service (TSID) with also the AEPR previously retrieved. These data are forwarded to the QoSLocalDispatcher, which validates the AEPR with the help of the QoSLocalBroker. If the QoSLocalDispatcher is authorized, it resolves the requested TSID to a TID that can be used to provide the thing service by exploiting the equivalent things list.

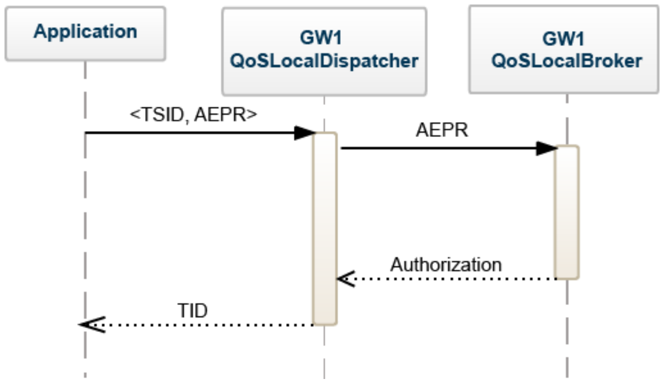

Figure 4.3: Local allocation.

The global allocation is performed when the list of equivalent things contains at least one thing not attached to the application's gateway. In this case, the QoSLocalDispatcher must delegate the allocation procedure to the QoSGlobalDispatcher. The global allocation is split in two more cases: one that takes place when the things involved are attached to multiple gateways, and another one that takes place when all things, in the list of equivalent things, are attached to only one remote gateway. The first case is explained in Fig. 4.4. After the authorization interaction, the QoSLocalDispatcher forwards the TSID to the QoSGlobalDispatcher that replies back with the selected TID. Finally, the Thing Services modules interact and the results are sent back to the application.

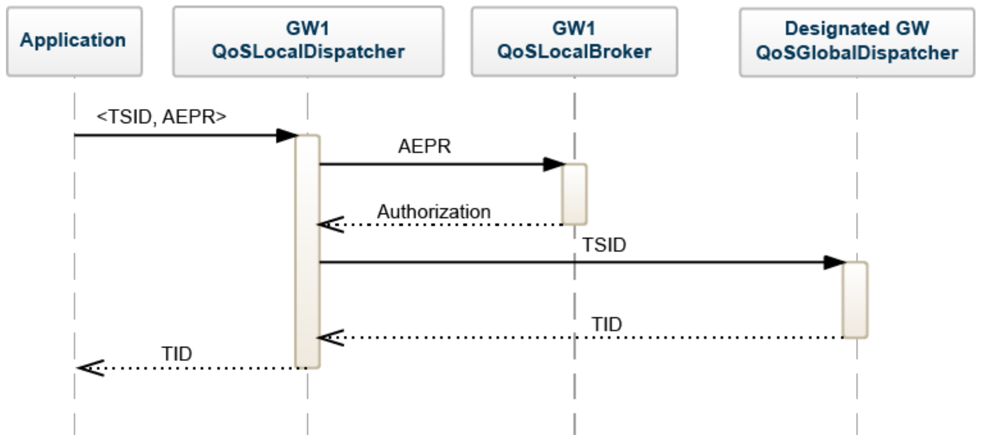

Figure 4.4: Global allocation, first case.

The second case, instead, is a mix of the local and the global allocation, because if the application is attached to the remote gateway the overall process will be accomplished with the local allocation procedure. However, because the gateway involved is not the application's gateway a central point of coordination is needed. The overall process is shown in Fig. 4.5 and is similar to the first global allocation procedure, however when the QoSGlobalDispatcher receives the allocation request it must forward such request to the QoSLocalDispatcher of the remote gateway. The remote instance of the QoSLocalDispatcher performs the allocation process following the local allocation procedure and then replies back with the selected TID. The QoSGlobalDispatcher sends the response to the origin QoSLocalDispatcher that forwards the TID to the Thing Service module. After the interaction, between the Thing Service modules involved, the results are sent back to the application.

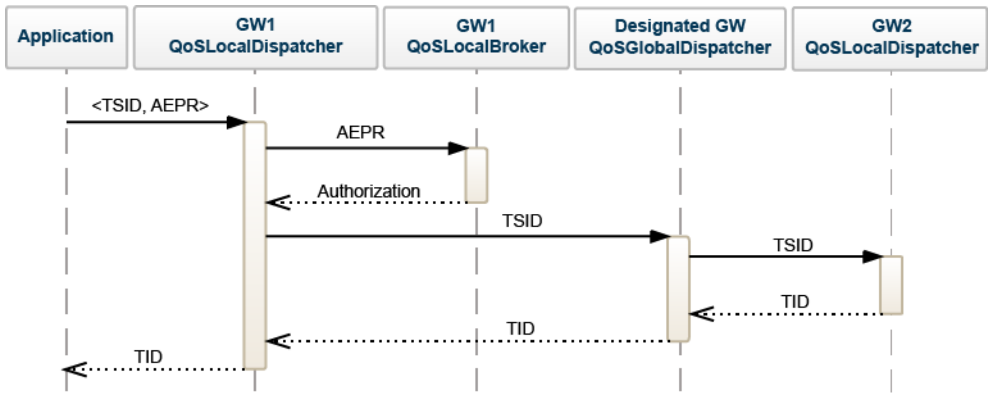

Figure 4.5: Global allocation, second case.

4.4 Conclusions

In this chapter a distributed QoS procedure specifically designed for Thing as a Service architectures is presented. The proposed solution aims at introducing in large heterogeneous IoT systems functionalities for QoS enforcement. Specific attention to scalability is given in the design at any rate. The proposed framework establishes a fertile soil for the definition of algorithms and schedulers that can guarantee efficient resource allocation, which will be presented in the next chapters.

5

Energy-Efficient QoS-aware Thing Service Selection

The two-phase QoS procedure presented in Chapter 4 base its decision on the Thing Service Selection algorithm. In this Chapter we present an energy efficient allocation algorithm that takes into account the unique features of smart things. Moreover the algorithm fully exploits the concept of equivalent services in order to extend the lifetime of battery powered devices. The contribution is twofold: (i) an energy-efficient thing allocation problem is formally defined as an instance in the class of generalized assignment problems; and (ii) a time-efficient heuristic algorithm is proposed that is shown, through numerical analysis, to find a solution close to the optimal one in a time suitable for implementation in a real system.

5.1 Related Work

Research efforts aimed at deploying large-scale IoT systems are only recent. For this reason, although the QoS-aware service selection problem has been widely studied in traditional platforms, solutions specifically designed for IoT are still missing. To the best of our knowledge, [35] is the only work proposing a QoS aware scheduling designed for service-oriented IoT platforms. A multi-layered scheduling model is proposed to evaluate the optimal allocation that meets the QoS requirements of applications. Different solutions are deployed at different layers to manage different system resources, such as network resources and IoT services. However, the proposed approach cannot be used for on-line IoT-service selection, since it is mainly suited for off-line provisioning and planning. Our approach, instead, is designed for run-time service selection and takes into account real-time requirements of applications with stringent deadlines. In the field of SOA several solutions for QoS-aware service selection have been proposed. In particular, in Web Service architectures, support for QoS is a challenge considering the requirements of both users and service providers [36]. In [37] authors propose a QoS broker for web service composition that aims at finding the best combination in order to maximize the end-user satisfaction. The proposed approach is based on a utility function that takes into account only end-users without analyzing constraints from server providers. The first work that tries to meet both end-user and server provider requirements is [38], where the authors propose a QoS-aware adaptive load balancing strategy. The proposed solution, however, does not consider hard real-time requirements but only customer fixed priorities. Finally, in [39] the authors present a heuristic that aims at finding the optimal allocation that minimizes the overall response time. The proposed solution, however, provides only probabilistic guarantees that cannot support hard real-time applications.

Solutions proposed in the field of SOA and Web Service architectures are not suited for IoT deployments, as IoT systems are composed of service providers characterized by constrained capabilities with limited battery capacity. On the other hand, such characteristics are considered by design in solutions proposed in the field of Wireless Sensor Networks (WSN), in which several solutions for task assignment have been proposed. In [40] the authors present a novel methodology to assign tasks to sensors in order to minimize the overall energy consumption. The WSNs is considered composed of heterogeneous devices with different capabilities. The proposed solution aims at minimizing the overall energy consumption through a greedy approach: each task is assigned to the sensor that consumes less power. The proposed approach, however, is also specifically designed for WSNs and leverages a depth knowledge of the sensors and their connections, assumptions that do not apply to the general IoT field, which is usually agnostic to network structure and hardware capabilities.

5.2 Assumptions

We base our work on the reference architecture presented in Chapter 2. In particular in this chapter we address the Thing Service selection problem, for this reason we focus only on the Designated GW which runs the Reservation algorithm while, the QoS procedure presented in Chapter 5, is hidden for the sake of simplicity.

The Designated GW acts as the central point of decision and fully exploit the concept of equivalent things presented in Chapter 2, to this aim, the QoS Broker leverages on the Context Engine and on the QoS Monitor.

Context, defined as “any information that characterize the situation of an entity” [41], enhances the description of a thing service, e.g., acquisition of the temperature, with any additional relevant information related to it, e.g., its location, its freshness, etc. Context information is processed by the Context engine, the entity within the framework responsible for collecting and managing all the context information related to things. The Context engine implements also a context-based service look-up functionality: it determines, for each application service request, the list of equivalent things that can provide that service. This is done by analyzing both the current context information associated to things, and the context information associated to the required service. Algorithms for context management and analysis are however outside the scope of this work.

The status of the system, instead, is maintained by a QoS monitoring entity, which provides information about the battery level, as well as the computational, communications, and storage capabilities of things.

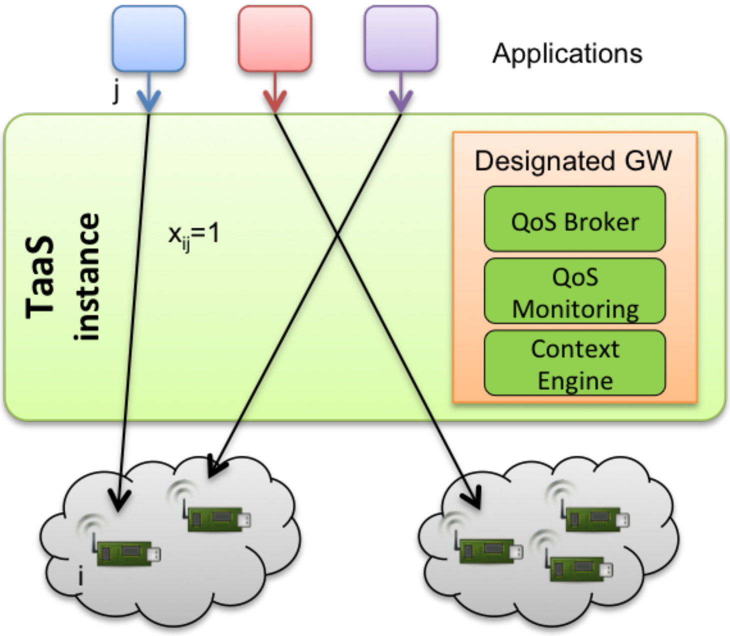

Figure 5.1: Conceptual model.

Based on the information about the status of things provided by the QoS monitoring entity, and the set of equivalent things per service request determined by the Context engine, the QoS broker determines if all requests can be satisfied. It then allocates one thing to each service request so as to minimize the energy consumption in case of energy-constrained devices, thus maximizing the lifetime of the system. In this chapter, we specifically focus on this allocation problem for a fixed set of requests and available things, and develop a heuristic algorithm to find a solution close to the optimal one.

As other requests are negotiated or the system status changes, thing selection and allocation to application requests need to be re-computed (transparently to applications) in order to continue guaranteeing the commitment to meet QoS requirements.

5.3 Problem Formulation

We now formally define the thing allocation problem addressed in this chapter. We are given a system comprising n things, each exposing a subset of IoT services, and a set of k requests for service invocation. Each service j is assumed to be invoked periodically with period p_j, e.g., in order to retrieve periodic updates from sensors or to send periodic commands to actuators. Thing i is assumed to be a constrained device capable of satisfying only one service invocation at a time. Moreover, a thing may be battery-powered. Let b_i be the battery capacity on thing i, i.e., the amount of available energy before service allocation (possibly equal to $+\infty$, if not battery-powered). Each invocation of a service j on thing i has a fixed execution time t_{ij}, including both communication and computation times, and a fixed energy cost c_{ij}, representing the overall amount of energy needed to accomplish the invocation of service j on thing i (including also energy consumption due to communication). The cost of execution of a service on a thing has a different impact depending on the initial battery level of the thing. In order to make a fair comparison among costs, we consider costs of execution over a common (hyper-)period h and normalized with respect to the available energy b_i. The hyper-period h is computed as the least common multiple among all request periods p_j; the normalized energy cost f_{ij} of executing service j on thing i is given by

$$f_{ij} = \frac{h}{p_j}\frac{c_{ij}}{b_i} \tag{5.1}$$

Not all services can be invoked on any thing, but the same service can be invoked on multiple equivalent things. Equivalence among things is based on context information associated to thing services, which we assume is provided by m_{ij} as follows

$$m_{ij} = \begin{cases} 1 & \text{if service} j \text{ can be invoked on thing} i \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Without losing generality, we assume that each service request can be executed by at least one thing, i.e., for any j, there is at least one i such that $m_{ij} = 1$.

The utilization u_{ij} of allocating requests for service j on thing i is finally defined as

$$u_{ij} = \begin{cases} \frac{t_{ij}}{p_j} & \text{if } m_{ij} = 1 \\ +\infty & \text{otherwise} \end{cases} \tag{5.3}$$

The thing allocation problem is then to allocate the k requests to the n things to minimize the maximum (normalized) energy cost among things over a hyper-period h, whilst guaranteeing that all service invocations are completely executed before their implicit deadline, i.e., the arrival of another invocation of the same service. Formally:

$$\min \left(\max_{1 \le i \le n} \sum_{j=1}^{k} f_{ij} x_{ij} \right) \tag{5.4}$$

s.t.

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j \in K \tag{5.5}$$

$$\sum_{j=1}^{k} f_{ij} x_{ij} \le 1, \quad i \in N \tag{5.6}$$

$$\sum_{j=1}^{k} u_{ij} x_{ij} \le v, \quad i \in N \tag{5.7}$$

$$x_{ij} \in \{0,1\}, \quad i \in N, \quad j \in K \tag{5.8}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if request} j \text{ is allocated to thing} i \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

and $K = \{1, ..., k\}, N = \{1, ..., n\}$.

Constraint (5.5) ensures that each service request is assigned to only one thing, whereas constraint (5.6) bounds the energy consumption of each thing in the hyper-period h to its battery capacity b_i. Finally, constraint (5.7) bounds the overall utilization of each thing to a schedulability limit v to guarantee that the implicit deadline of each invocation is satisfied. The limit v is a bound based on the well-known sufficient condition for the schedulability of a set of periodic tasks on a single CPU [42].

The problem is an Integer Linear Problem that results to be an instance of the Agent Bottleneck Generalized Assignment Problem (ABGAP) [43], a variation of the well-known Generalized Assignment Problem (GAP) defined in the literature and known to be NP-hard. To the best of our knowledge, there is no well-known general algorithm specifically designed to solve ABGAP. Stemming from heuristics proposed to solve GAP and BGAP in [44] and [45], respectively, we developed a novel greedy polynomial-time heuristic algorithm to solve ABGAP, and applied it to the problem defined by (5.4)-(5.7)

5.4 The RTTA Algorithm

In this section we describe the proposed heuristic, named Real Time Thing Allocation algorithm (RTTA), to solve the ABGAP problem defined in the previous section. The input to RTTA are: the number of things n, the number of requests k, the normalized energy cost matrix $F = \{f_{ij}\}$, the utilization matrix $U = \{u_{ij}\}$, a precision threshold ε, and, finally, an optional priority matrix $P = \{p_{ij}\}$. The latter is used to steer the thing allocation procedure Feas described in detail below. The output is: a boolean $isFeasable$, which takes the $True$ value if at least one allocation exists, the allocation vector y that maps service requests to things, and the corresponding residual battery vector z.

The rationale behind RTTA is to iteratively search for the first feasible allocation that guarantees the highest minimum level of residual battery for all things. To this aim, RTTA leverages a procedure Feas that, given a threshold θ, finds an allocation so that the residual battery on each thing after service invocation is no lower than θ. On every iteration, the threshold θ is decreased until a feasible solution is found with an acceptable precision level measured by ε. More specifically, a binary search strategy is used to reduce the time needed to execute the overall procedure. At any iteration, $upper$ and $lower$ give the current upper and lower bounds of threshold θ, respectively. When the difference between $upper$ and $lower$ is less than the input parameter ε, the algorithm stops - see Fig. 5.2.

The core of the RTTA algorithm is the procedure Feas which is derived from a classical approach in the literature to tackle assignment problems [45]. The pseudocode of the Feas algorithm is reported in Fig. 5.3. The allocation is based on the values of a priority

```
Algorithm RTTA
Input: $n$, $k$, $P$, $F$, $U$, $\varepsilon$
Output: $z$, $y$, $isFeasable$

$[z,\quad y,\quad isFeasable] \leftarrow Feas(...)$
if $isFeasable = True$ then:
    $lastFeas \leftarrow 0;\quad upper \leftarrow 1;\quad lower \leftarrow 0$
    while $upper - lower > \varepsilon$ do:
        $\theta \leftarrow \frac{(upper-lower)}{2}$
        $[z,\quad y,\quad isFeasable] \leftarrow Feas(...)$
        if $isFeasable = True$ then:
            $lastFeas \leftarrow \theta;\quad lower \leftarrow \theta$
            $\theta \leftarrow \theta + \frac{(upper-lower)}{2}$
        else:
            $upper \leftarrow \theta;\quad \theta \leftarrow \theta - \frac{(upper-lower)}{2}$
    if $isFeasable = False$ then:
        $\theta \leftarrow lastFeas$
        $[z,\quad y,\quad isFeasable] \leftarrow Feas(...)$
```

Figure 5.2: Pseudo-code of RTTA.

matrix P passed as input. In particular, element p_{ij} of P is a measure of the desirability of allocating request j to thing i. All requests are then considered iteratively for allocation. At each step, the next request to allocate, say j, is the one having the maximum difference between the largest and the second largest p_{ij} (for all things i such that constraint 5.7 is met). Request j is then allocated to the thing i for which p_{ij} is a maximum. If a service request for which no feasible assignment is found, i.e., any possible allocation to a thing implies its residual battery level goes below θ, the algorithm returns $isFeasable$ equal to $False$.

Otherwise, a post-processing local optimization procedure is performed in order to improve the optimality of the solution. This is achieved by performing local exchanges: for each request, the procedure verifies that the exchange of a service request j from the selected thing i' to any other thing i^* increases the overall residual battery; if that happens, the allocation is modified to select the thing i^* instead of i'. The final solution represented by y and z is then returned. Several choices are possible for setting the priority values in P. Preliminary numerical tests have shown that good results are obtained when P is set equal to F or U. Both cases are then considered in the final specification of the heuristic solution. The computational complexity of the RTTA algorithm is given by the result below.

Property 1. *The complexity of RTTA is $\mathcal{O}\left(\beta\left(nk^3\right)\right)$.*

Proof. Let us first evaluate the complexity of the Feas procedure. The most expensive phase, on each iteration, is to compute F_j which requires $\mathcal{O}\left(nk\right)$ time. To compute the

```
Algorithm Feas
Input: $n$, $k$, $P$, $F$, $U$, $\theta$
Output: $z$, $y$, $isFeasable$

$N \leftarrow \{1...n\}$; $K \leftarrow \{1...k\}$; $v \leftarrow k\left(2^{1/k}-1\right)$; $isFeasable \leftarrow True$
for $i \leftarrow 1$ to $n$ do: $ci \leftarrow 0$
for $i \leftarrow 1$ to $n$ do: $zi \leftarrow 1$
while $isFeasable = True$ and $K \neq \emptyset$ do:
$d^* \leftarrow -\infty$
  foreach $j \in K$ do:
    $F_j \leftarrow \{i \in N : c_i + u_{ij} < v, \quad z_i - f_{ij} > \theta\}$
    if $F_j = \emptyset$ then:
      $isFeasable \leftarrow False$
      return
    $i' \leftarrow \arg\max\{p_{ij} : i \in F_j\}$
    if $F_j \setminus \{i'\} = \emptyset$ then: $d \leftarrow +\infty$
    else: $d \leftarrow p_{i'j} - \max_2\{p_{ij} : i \in Fj\}$
    if $d > d^*$ then:
      $d^* \leftarrow d; \quad i^* \leftarrow i'; \quad j \leftarrow j'$
  if $isFeasable = True$ then:
    $y_{j^*} \leftarrow i^*; \quad z_{i^*} \leftarrow z_{i^*} - f_{i^*j^*}$
    $c_{i^*} \leftarrow c_{i^*} + u_{i^*j^*}; \quad K \leftarrow K \setminus \{j^*\}$
#Local optimization
foreach $j \in K$ do:
  $i' = y_j$
  $F_j \leftarrow \{i \in N : c_i + u_{ij} < v, \quad z_i - f_{ij} > \theta, \quad i \neq i'\}$
  if $F_j = \emptyset$ then: continue
    $i^* \leftarrow \arg\max\{z_i - f_{ij} : i \in F_j\}$
    $maximum \leftarrow \max\{z_i - f_{ij} : i \in F_j\}$
    if $maximum > z_{i'} - f_{i'j}$ then:
      $y_j \leftarrow i^*; \quad z_i \leftarrow z_{i'} + f_{i'j}; \quad z_{i^*} \leftarrow z_{i^*} - minimum$
      $c_{i'} \leftarrow c_{i'} - u_{i'j}; \quad c_{i^*} \leftarrow c_{i^*} + u_{i^*j}$
```

Figure 5.3: Pseudo-code of Feas.

first and the second max the algorithm needs $\mathcal{O}(nk)$. The while loop (line 8) performs $\mathcal{O}(k)$ assignments, hence by considering also the inner loop (line 10) we obtain a total of $\mathcal{O}\left(k^2\right)$ time. We can conclude that the overall time complexity is $\mathcal{O}\left(nk^3\right)$. The complexity of RTTA is governed by the choice performed on ε. We can derive β according to:

$$\beta = \log_2 \frac{upper - lower}{\varepsilon} \tag{5.10}$$

□

Hence, the overall complexity is $\mathcal{O}\left(\beta\left(nk^3\right)\right)$.

5.5 Performace Evaluation

In this section we report a numerical evaluation of RTTA as compared to the optimal allocation obtained by solving the problem defined by (5.4)-(5.7), by means of a standard optimization solver, i.e., IBM ILOG CPLEX. RTTA was implemented in C++. Experiments were run on a machine with a Linux 64bit operating system. The machine is equipped with an Intel Core i7-4770 CPU @ 3.40GHz, and 16 GB of RAM. The algorithm is evaluated in different scenarios, each one characterized by a number of service requests and available things, and an average number of services exposed by each thing, expressed as a fraction of the overall number of service requests. For each scenario, one hundred different inputs are randomly generated. More specifically, the initial battery levels, computational costs and periods are drawn from a uniform distribution with parameters as reported in 5.1. Moreover, m_{ij}'s, i.e. context information about which services can be invoked on which things, are also randomly generated so that the average number of services per thing characterizing the scenario is fulfilled. Metrics of interest are then estimated for each scenario along with 95% confidence interval.

Parameter	Range
Period	10 - 100 s, step 10 s
Initial battery level	50 - 25 mJ, step 5 mJ
Execution cost	210^{-4} - 610^{-4} mW
Execution time	7 - 22.5 ms

Table 5.1: Experiments Parameters

Numerical experiments have been conducted with several combinations of the number of service requests and things. Results are similar in all considered scenario, therefore we limit in this work to report results to two different scenarios only. The first scenario is characterized by 50 things and 500 requests, while the second consists of 100 things and 500 requests. In both scenarios, the following average number of services exposed by each thing are considered (expressed as a fraction of the overall number of service requests, i.e., 500): 15%, 25%, 50%, 75% and 100%, respectively. To evaluate the performance of RTTA, we consider the residual battery ratio defined as the ratio between the battery level of a thing at the end of the hyper-period and the initial battery level.

Fig. 5.4 and Fig. 5.5 show the average minimum residual battery ratio over the set of different inputs for the first and second scenarios, respectively. The objective of RTTA is to maximize such ratio among all things, hence this metric can provide a measure of how far is the heuristic solution from the optimal one. As can be seen, in all cases the heuristic succeeds into finding an allocation which is very close to the optimum. The difference reduces as the percentage of the average number of service requests that a thing can satisfy increases. This means that RTTA is more efficient when there are more opportunities to allocate a service to thing, i.e., the solution space is larger. On the other

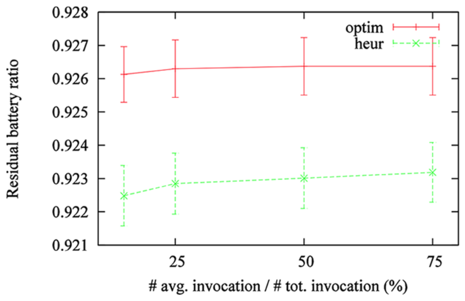

Figure 5.4: Min residual battery ratio (50 things, 500 requests).

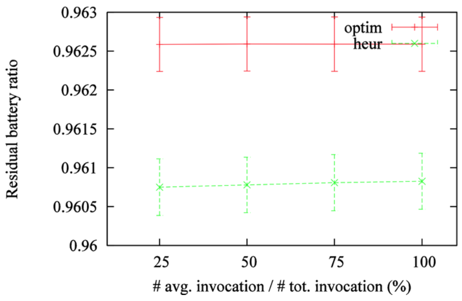

Figure 5.5: Min residual battery ratio (100 things, 500 requests).

hand, when low percentages are considered (e.g., 15%), i.e., the sets of equivalent things are small, we can notice that also the optimal solution decreases, though at a lower rate than the heuristic algorithm.

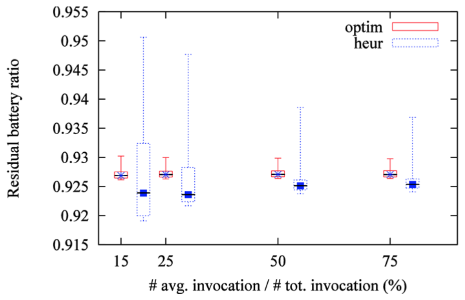

Figure 5.6: Residual battery ratio (50 things, 500 requests).

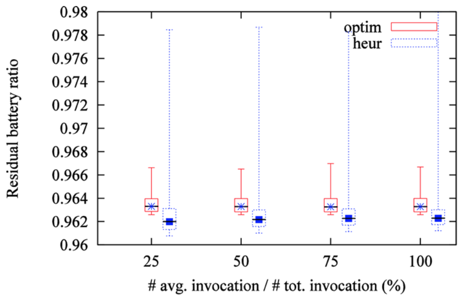

Figure 5.7: Residual battery ratio (100 things, 500 requests).

Fig. 5.6 and Fig. 5.7 illustrate the distribution of the residual battery level in the first and second scenario, respectively. The distribution is shown through the box-plot representation where the bottom and top of the box represent the 25th and 75th percentile, the band in the box the median (50th percentile), while the ends of the whiskers represent the minimum and 95th percentile. As can be seen, even if the objective function tries to maximize the remaining battery of the thing with less energy, the algorithm succeeds in distributing the energy consumption among all the things obtaining uniform battery consumption.

Finally, Fig. 5.8 and Fig. 5.9 illustrate the computation time required by the heuristic algorithm and the optimization tool to find the optimal allocation in the first and second scenario, respectively. As can be seen, the time required by the solver to find the optimal solution is at least an order of magnitude higher than the time required by the proposed algorithm to find an allocation that is comparable to the optimal value. It is also worth noting that the performance of the heuristic algorithm is only slightly affected by the size

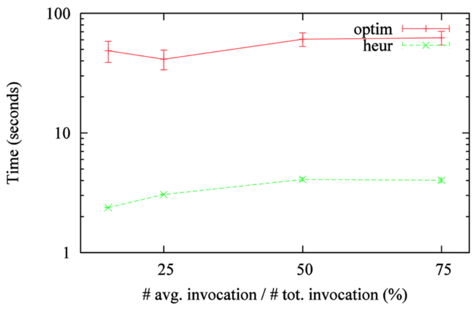

Figure 5.8: Computation time (50 things, 500 requests).

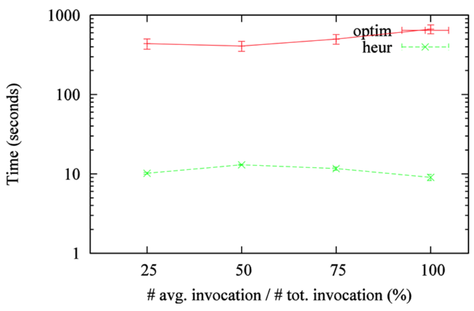

Figure 5.9: Computation time (100 things, 500 requests).

of the problem (100 things in the second scenario vs. 50 things in the first one), whereas this has a much higher impact on the solver, which take almost an additional order of magnitude of time to find the solution.

5.6 Conclusions

In this chapter we tackled the problem of QoS-aware service selection in Things as a Service architectures. First, we formally defined the problem of optimum energy-efficient service selection in which real-time QoS requirements are enforced considering context-information. We then derived a heuristic to solve the problem in a polynomial time in order to allow its implementation in real systems. The heuristic algorithm was validated through simulation which demonstrated that it guarantees an allocation close to the optimal value in polynomial time.

Part II

QoS in the Access layer

6

CoAP Proxy Virtualization for the Web of Things

From this chapter we start to analyze the access network. The future Web of Things (WoT) foresees a web in which applications can seamlessly access physical objects through the same REST interface used today to access web services. A key enabler of this shift is the Constrained Application Protocol (CoAP), a redesign of the popular HTTP protocol that aims at supporting resource-constrained devices for Machine-to-Machine applications. In the following we briefly introduce the CoAP protocol which will be used in al the following chapters in order to give the basis to unaware readers.

Then we present a proxy virtualization framework to support scalability and easy implementation of custom functionalities in large WoT deployments. The functionalities offered by a CoAP proxy are exploited to transparently decouple applications from servers and to guarantee the implementation of custom policies and functionalities through virtualization techniques. A solution based on Linux Containers is implemented in a real testbed made of off-the-shelf hardware and open-source software, demonstrating the feasibility of the proposed design. Experimental results have shown that the response time is highly improved by our solution thanks to central coordination of concurrent requests from different virtual proxy instances. Moreover, to highlight the potential of the proposed framework, a priority-based Quality of Service policy is also implemented and evaluated.

6.1 Motivation

In the attempt of enabling concurrent execution of applications over the same sensor network infrastructure, research work has been carried out extensively in the context of Wireless Sensor Networks (WSN). The first concept has been introduced in [46] where the authors defined the concept of Virtual Sensor Networks (VSN). A VSN is sensor network deployment capable of sharing its sensor capabilities and network resources to support the execution of different tasks and applications, concurrently over the same physical infrastructure. Such abstract initial concept has triggered the definition of different VSN architectures such as the work presented in [47], which presents VITRO, an architecture specifically designed to introduce virtualization techniques in WNS. In [48] the authors propose to improve QoS performance of applications by means of caching strategies. The authors consider a deployment in which a proxy is employed to perform HTTP-CoAP protocol translation and also implements caching policies. In [49], instead, the authors propose modifications to the CoAP observing extension to prioritize notifications between critical and non-critical. Notifications marked as critical are delivered immediately to destination. Congestion control is only marginally handled by CoAP specifications. For this reason, the subject has been studied in [50], in which the authors propose congestion control algorithms to regulate the number of outstanding transactions from the same client in order to minimize the amount of buffer overflows.

The scenario we consider is a WoT deployment composed of a set of smart objects, usually implemented through embedded devices with constrained capabilities. The CoAP protocol is used as the enabler to allow applications to interact with smart objects that expose their services as resources of a CoAP server. An intermediate device, e.g., a gateway, is assumed to act as a bridge between smart objects and the Internet.

A large-scale scenario, where sensors and actuators are exposed to different heterogeneous applications, presents new challenges to ensure scalability, considering the limited memory and computational capabilities of constraint devices that cannot manage efficiently concurrent requests from multiple applications. Moreover, this scenario requires additional features to support heterogeneous applications characterized by different contexts and different technologies that might also be unknown at the time of the deployment. Service differentiation, for example, is required to handle heterogeneous applications with different QoS or security requirements. Service customization, instead, is mandatory to ensure system expandability over a long lifespan: allowing third-party implementations to introduce custom functionalities or implement custom protocol translation is essential to guarantee expandability towards future technologies and protocols.

6.2 Proxy Virtualization Framework

The solution we propose is a proxy virtualization framework that can be implemented on each gateway linking a network of smart objects to the rest of the Internet. The overall idea is to install on the gateway a CoAP proxy that could be virtualized. The implementation of a proxy allows transparent detachment between applications and smart objects. Scalability is accomplished by leveraging the resources offered by the gateway, usually not deployed on constrained hardware, which can provide buffering and additional processing. Since the proxy is responsible for managing all the requests, it can be exploited to implement service request differentiation.

The usage of a proxy does not necessarily require virtualization. Implementation of custom policies and functionalities can be performed directly in the proxy modifying its standard implementation. Such practice, however, does not guarantee easily expandability and does not enable clean installation of third party functionalities, unless allowing external entities to modify the proxy itself. In this context, virtualization is an attractive

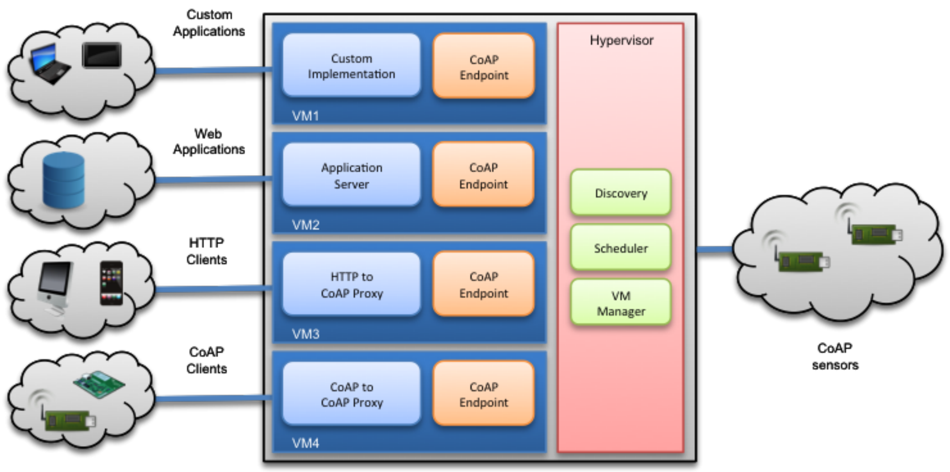

Figure 6.1: Proxy virtualization framework concept.

solution to allow third party customization, which can benefit from strong isolation of the software introduced in the platform and device abstraction provided by the proxy itself.

The overall concept architecture is illustrated in Fig. 6.1. Core of the architecture is a CoAP proxy called proxy hypervisor[1]. Only the hypervisor has direct access to the smart objects and it manages the access to them. A virtualization framework is then introduced in the gateway to deploy a set of virtual proxies, whose interface is exposed to applications, which lack direct access to the interface exposed by the hypervisor. Virtual proxies run inside virtualized environments created and managed by the hypervisor.

External modules are required to implement the CoAP protocol by means of a CoAP endpoint sub-component, hereafter CoAP endpoint for short, in order to communicate with smart things. External modules can be exploited to implement a wide range of functionalities offered to applications such as protocol translation. Each virtual proxy has a dedicated virtual network interface and a specific IP address and is instantiated to manage a group of applications. Within each group, applications know the IP address of the associated virtual proxy and interact only with it to discover resources and issue requests, completely unaware of the framework.

Virtual proxies interact with the hypervisor to implement their operations. The hypervisor exposes towards them the same interface as a reverse proxy, in order to allow virtual proxies to act transparently as they were interacting directly with resources. At the time of activation, each virtual proxy interrogates the hypervisor to discover the available resources and expose them to applications. This procedure follows the Resource Discovery procedure defined in the CoAP RFC [7].

[1] The term hypervisor here is employed to highlight that it controls the access to hardware resources (CoAP servers) and it creates and configure the virtualized containers according to configuration parameters

As the virtual proxy receives a request from an application, it issues the request to the resource exposed by the hypervisor. Eventually, the hypervisor manages the request as any standard reverse proxy: requests for the same device are buffered and dispatched once at a time as the device becomes available (as required by the CoAP standard).

The hypervisor can be exploited to implement service differentiation policies for different groups of applications. Modifications to the standard reverse proxy behavior can be implemented to manage requests coming from different virtual proxies, hence from different groups of applications.

In particular, QoS policies can be enforced implementing different management strategies for CoAP requests coming from different virtual proxies. Moreover, the solution based on virtualization does not require modifications to applications and does not require the hypervisor to be aware of any information specific to application; the hypervisor differentiate its behavior only taking into account virtual proxies.

In order to illustrate an example of the possibilities offered by the proposed framework, a QoS priority schema is proposed. Applications using CoAP to interact with things are grouped into priority classes, each one associated with a virtual proxy. A strict priority policy is implemented in the hypervisor: different queues, instead of a single one, are implemented for each CoAP server, requests for resources hosted on the same devices are queued based on the virtual proxy that issues the request. A fixed priority scheduler is then implemented to dispatch the requests according to a strict priority policy: the requests of a class are scheduled only when there are no requests from higher classes. More complex QoS policies can be easily implemented in the proposed framework, however their definition and implementation is out of the scope of this work.

6.3 Framework Implementation

In order to validate the proposed solution, we have prototyped the proposed framework using Linux Container [51], a lightweight operating-system level virtualization framework for running multiple isolated Linux systems on a single host. Among different virtualization techniques, Linux Container has been selected as it provides a high level of isolation and security with good efficiency [52]. Although our implementation leverages a specific virtualization technology, it is worth to highlight that it does not exploit any specific functionality. Any virtualization technique, e.g. other hypervisor-based ones such as KVM[53], can be adopted without requiring modifications to the overall concept, although at the cost of an increased overhead of the virtualization functions.

The overall structure of the implementation is depicted in Fig. 6.2. Core of the implementation is the proxy hypervisor, deployed using the Californium framework [54], a Java-based CoAP library. A standard proxy is deployed with additional customized functionalities to manage Linux Container instances. Containers are configured with one virtual network interface of type veth whose peer device on the host operating system is connected to a virtual bridge (br0) that includes also the physical Ethernet interface. Each virtual interface is configured, at time of container creation, with a different public

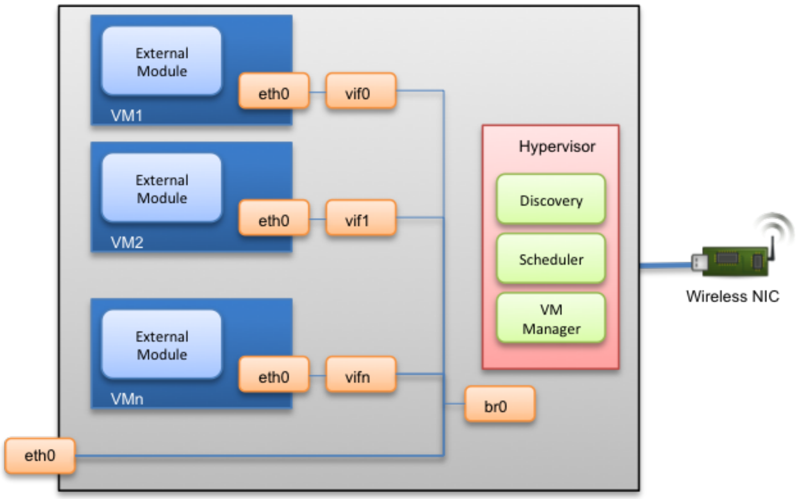

Figure 6.2: Proxy virtualization implementation.

IP addresses in order to allow external applications to reach the services running in each one of them. The hypervisor runs directly on the host operating system and has direct access to the NIC interface that communicates with the sensor network and to the virtual bridge br0 to communicate with the virtual proxies.

Each container could be considered as a standard Linux host that can run the custom code provided by third-parties to manage the requests of a group of applications. The code running on the container can be written on any language and implement any kind of functionality, the only requirement is the deployment of CoAP endpoint functionalities to interact with the hypervisor. The custom code is responsible for implementing an interface towards applications that will be accessible through the public IP address of the container.

As a simple proof-of-concept, we deploy on each container a standard CoAP proxy to realize a simple CoAP-to-CoAP proxy that exposes a simple pass-through interface for applications. To this aim, the CoAP proxy implementation of the Californium framework is exploited.

Finally, the simple QoS prioritization policy, illustrated in Fig. 6.3, is implemented to demonstrate the possibilities offered by the architecture for implementing custom request management policies. The goal is to differentiate the service offered to requests from two different groups of applications, a high priority group and a low priority group. Two buffers for each CoAP server are introduced in the hypervisor to differentiate requests from the two groups of applications. A strict priority-based scheduler is introduced in the hypervisor to select the request to be served towards each CoAP server: requests of the former group are always served first, while requests from the latter are selected only when no high priority requests are waiting.

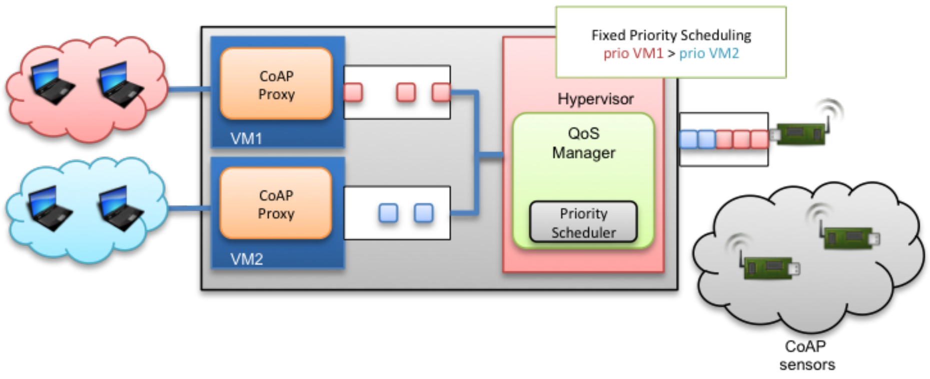

Figure 6.3: Prioritization schema.

6.4 Performance Evaluation

In order to assess the performance of the proposed solutions, our implementation has been evaluated by means of a real-world testbed. In the following, we first illustrate the hardware platform we adopted for our experiments, then we show the results obtained with the platform compared with the results obtained deploying a standard solution.

6.4.1 Testbed design

The testbed is a proof-of-concept platform implemented using off-the-shelf hardware and open-source software freely available. Smart objects are deployed by means of Zolertia Z1 [55] nodes, equipped with an IEEE 802.15.4 wireless interface capable of running Contiki [56] as operating system. Contiki is an operating system specifically designed for con- strained devices and the Internet of Things. A striped networking stack called uIP is implemented to fit the limited memory capacity of devices. However, uIP implements a single packet buffer that represents a performance bottleneck in several scenarios, as shown in the experimental results. The sensor node exposes its resources running Erbium [57], a lightweight CoAP server part of the official standard release of Contiki. In order to simulate processing operations, an active task is triggered by each CoAP request, resulting in a processing delay of around 300ms.

Considering that smart objects are often equipped with a wireless interface for ease-of-deployment (e.g. IEEE 802.15.4 or IEEE 802.11), a gateway is usually deployed as point of access for one or more smart objects towards/from the Internet, performing, if necessary, protocol translation functionalities. In our deployment, a prototype of a gateway that links sensor nodes with the Internet is emulated by means of a laptop. The system is an Intel(R) Core(TM) i5 CPU @ 2.40GHz and 4 GB of RAM. Since the laptop is not equipped with an IEEE 802.15.4 transceiver, a Zolertia Z1 is connected to the laptop to act as border router managing the low level communication between sensors and the laptop. A daemon called tunslip runs in the background to manage the communication

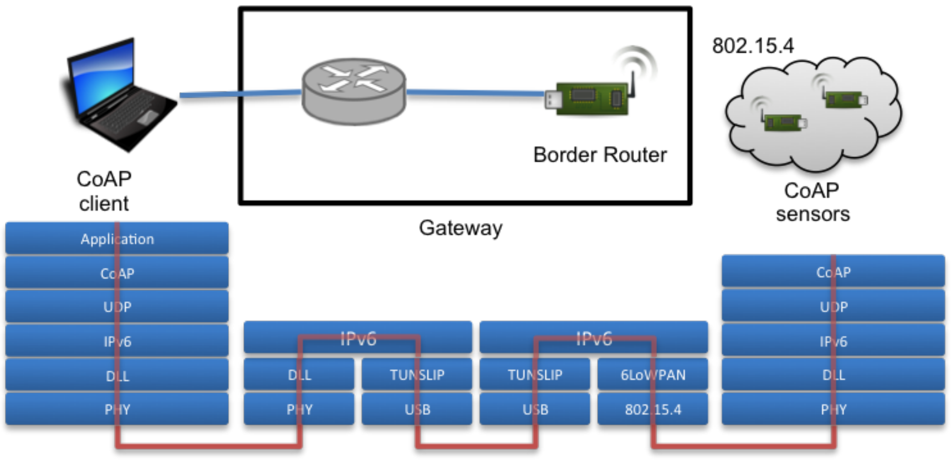

Figure 6.4: Standard solution deployment.

(through the Serial Line IP protocol, SLIP for short) between the Zolertia and the Linux operating system that eventually sees the node as a tun virtual interface [2].

CoAP clients are deployed using the Californium framework and runs on a laptop. Every client issues CoAP requests in CON mode in order to guarantee reliable delivery of information.

6.4.2 Standard Solution

Current approaches towards the Web of Things allow applications to directly access smart devices: things acting as CoAP servers expose a set of resources. Applications interact with constrained devices as CoAP clients in order to retrieve information from smart objects or to send data to actuators.

In this first set of experiments we evaluate the performance of this approach, as a term of comparison for the proposed framework. Fig. 6.4 summarizes the logical representation of our setup for this set of experiments, along with the network stack implemented by each element. The metric request delay, defined as the time between the CoAP request is issued by the client and the time the client fully obtain the corresponding reply, is measured to assess the performance. First, the baseline for the request delay is assessed. To this aim, a simple experiment in which a single client issues 500 requests on the same resource is run. The client is set in order to issue requests in sequence, i.e., a new request is sent only when the previous is satisfied. The resulting average delay is $389.99 \pm 0.042ms$[3], which includes not only the processing delay on the sensor but also the round-trip transmission delay. The low variance of the delay that results from the single client scenario confirms that this standard setup does not present issues in handling

[2] TUN is a virtual network kernel device simulating a network layer device. The device is emulated at layer 3 for routing purposes.

[3] The 95% confidence interval of the average delay is shown

non-concurrent requests. To evaluate the performance with concurrent requests, a set of experiments with multiple clients is run. Arrival of CoAP clients is modeled as a Poisson process (arrival rate λ); each client issues a single CoAP request.

Fig. 6.5 illustrates the cumulative probability distribution function of the request delay with different λ values. As expected, both the delay distribution shifts forward to the right and the tail of the distribution gets heavier as the request rate λ increases; at an average rate of two requests per second, the delay can grow up to $100s$ in some cases. This behavior can be explained by considering the limits of the uIP implementation, which have only one packet buffer shared for both transmission and reception. This represents a bottleneck on the sensor limiting the number of concurrent CoAP requests that can be processed concurrently. As a matter of fact, in case a CoAP request is received while another is waiting, the frames are correctly decoded and acknowledged by the IEEE 802.15.4 radio, but discarded at the IP layer. Since CoAP CON mode is employed, clients retransmit dropped requests until a response is obtained, hence the delay increases dramatically as the number of retransmissions increases. This is confirmed by the delay distribution that is characterized by a multi-modal distribution, which can be explained considering the congestion control defined in the CoAP protocol: when a running request times out a retransmission is scheduled with a back-off delay increased exponentially after every retransmission.

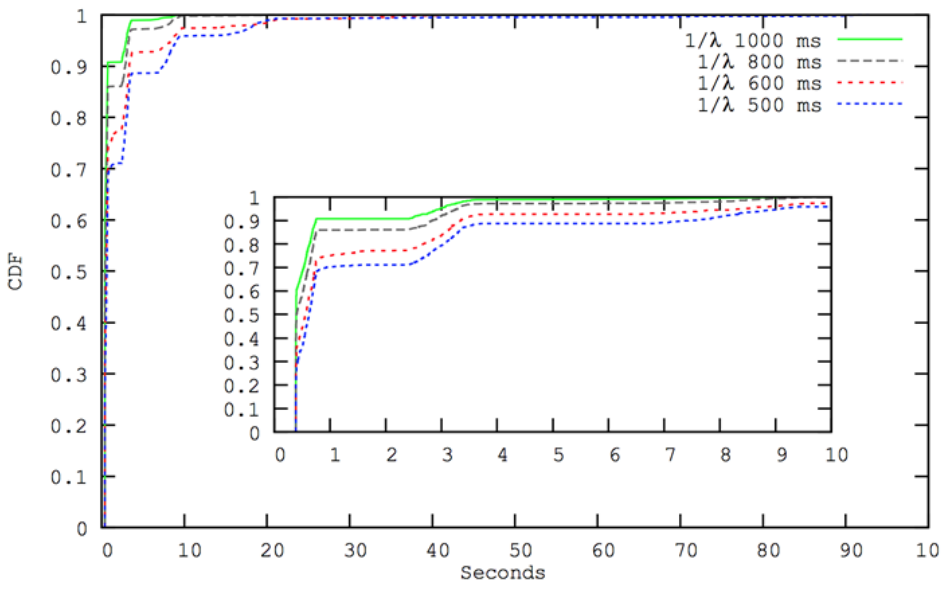

Figure 6.5: Request delay CDF, standard solution.

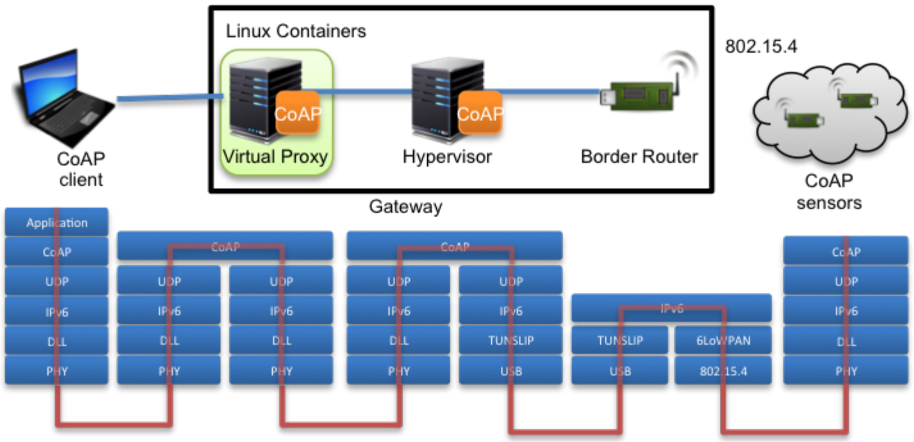

Figure 6.6: Proxy virtualization deployment.

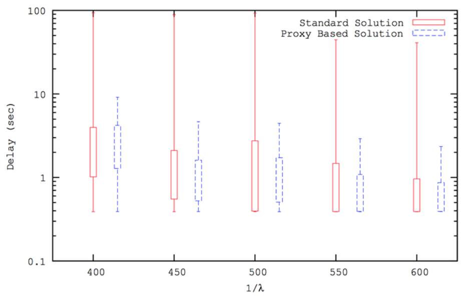

Figure 6.7: Request delay, proxy-based solution.

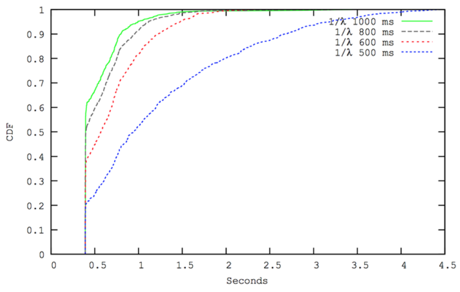

Figure 6.8: Request delay CDF, proxy-based solution.

6.4.3 Virtualized Proxy Solution

The first set of experiments highlighted the limitations of exposing directly smart objects to applications. In this section we present the results obtained with the proposed solution. In regard with the high response delays, the expected result is the minimization of the CoAP request delay achieved through an efficient usage of the limited capabilities of smart objects.

The proposed framework has been implemented as described in Section 6.3 and is illustrated in Fig. 6.6. In order to measure the performance of the proposed architecture, in this first set of experiments, the hypervisor is deployed as a standard CoAP proxy, without introducing any policy for service differentiation. Only one virtual proxy is deployed to manage all the CoAP requests. The custom code implemented within the container is a standard CoAP proxy. As for the experiments presented in Section 6.4.2, arrival of CoAP clients is modeled as a Poisson process with an arrival rate λ.

Fig. 6.7 illustrates the experimental results obtained with our framework compared with the standard solution. A boxplot representation is used: the bottom and top of the box represent the 25th and 75th percentile, the band in the box the median (50th percentile), the ends of the whiskers represent the minimum and 95th percentile. The results show that the maximum experienced delay decreases dramatically.

Fig. 6.8 illustrates the cumulative probability distribution of the delay. Results show how the proposed setup succeeds in avoiding the loss of CoAP requests that caused large delay in the standard solution, as demonstrated by the lack of a multi-modal pattern in the distribution, typical of the exponential back-off. This experiments demonstrates that the improved response delay is a consequence of the proxy. In fact, since CoAP requests are now buffered at the proxy, whose storage capacity is not constrained, no concurrent request is dropped in this case.

6.4.4 Prioritization Schema

Finally, a set of experiments is run to demonstrate the effectiveness of the deployment. A simple scenario running two virtual proxies is deployed to test prioritization capabilities with two categories of applications. Two independent Poisson processes model the arrival of CoAP requests for the two groups of applications; $\lambda_{low} = 1/800$ and $\lambda_{high} = 1/1200$ are adopted as arrival rates of the low priority and high priority applications, respectively. Fig. 6.9 shows the request delay over time. As expected, the requests of the high priority class experiences a delay significantly lower than the low priority.

Fig. 6.10, instead, shows the average delay of high priority requests when different λ_{low} values are considered. As can be seen the average delay of high priority requests is only slightly affected by the rate of the low priority clients. These results, however, demonstrates that a full separation between low and high cannot be achieved. This can be explained considering the priority inversion phenomenon caused by the non-preemptive execution of tasks. Since preemption is not allowed by the things, it can happen that a low priority is in execution when a high priority request arrives. In this case the high priority must wait until the end of the execution of the current request. It is worth to highlight that the additional delay of the high priority requests is equal to the execution of a request in the worst case, as demonstrated by results Fig. 6.9.

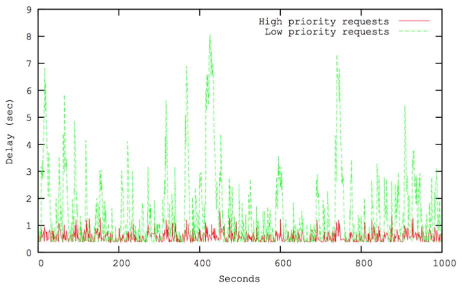

Figure 6.9: Request delay versus time, high versus low priority requests.

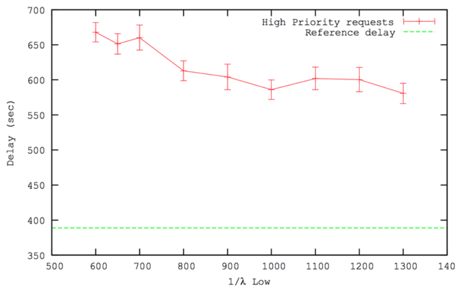

Figure 6.10: Request delay (high priority requests) with different lambda values (low priority requests).

6.5 Conclusions

In this Chapter a proxy virtualization framework for large-scale WoT deployments has been presented. The proposed solution is designed to enable scalable and highly customizable deployments, ready for supporting the large multitude of heterogeneous of applications expected in the future. A possible practical deployment based on Linux Container as virtualization technology is presented. Experimental results carried on by means of a real testbed demonstrated that the proposed solution succeeds in overcoming the limitations of the classic WoT architectures, without introducing significant overhead.

7

QoS-support for the CoAP Observing paradigm

The Internet of Things field is still in an early stage and different protocols are currently under evaluation in this specific scenario. Moreover, the heterogeneity of devices, as well as the different types of communication infrastructure involved, raises many challenges that must be addressed.

In this chapter we will focus on how to provide QoS capability directly within the application protocol. In particular in recent years many application protocols have been developed or adapted to the IoT field and some of them already provide QoS capabilities.

We will consider the CoAP protocol and its extension in the rest of this chapter, due to the fact that it has been designed especially for the IoT/M2M field and is suitable for constrained devices working in constrained networks. Adding QoS capabilities over CoAP could enhance the protocol in order to enable its adoption in new interesting use cases e.g. industrial wireless sensor networks.

In particular we focus on the observing paradigm [8]. The QoS support, in fact, can be exploited to create a new set of applications that use periodic notifications within certain deterministic bounds to provide near real-time services to the end-user.

7.1 State of Art

At the Application level there are two different protocols suitable for IoT which also provides a certain level of QoS: Data Distribution Service (DDS) [58] and MQTT [59].

DDS is a publish/subscribe architecture based on topics. Each Publisher publish its information under a certain topic while each Subscriber register its interest in a certain topic. Applications that want to publish something send data to their Publisher that is in charge to send each “packet” to all the interested Subscribers.

Each Subscriber is then exploited by a different application to read “packets”. Discovery of Publisher/Subscriber are left out of the specification and partially covered by the DDS-RTPS [60], however the overall idea is based on the fact that a Publisher/Subscriber announces its presence to a list of known locators.

Finally, the transport layer is considered only marginally, the DDS-RTPS claims to use UDP as it does not introduce any delay (like TCP) and all the needed functionalities, also in term of QoS, can rely on it. However, it does not consider communication problems as well as delays caused by congestion in the network explicitly.

DDS also include a QoS by design with different parameters that can be chosen to select the required QoS level. Each Publisher, in fact, exposes its topics enriched with metadata information that explains the QoS capabilities of the Publisher.

On the other hand, Subscribers seek for a certain topic (exposed by a Publisher) that also satisfies the QoS requirements. In other words, a Subscriber can be associated with a Publisher only if the topic and the QoS properties exposed by the Publisher match the topic and the requirements asked by the Subscriber.

Among other parameters DDS provides three different timeliness properties: DEADLINE, LATENCY-BUDGET, TRANSPORT-PRIORITY. Deadline is used by applications to specify the maximum inter arrival time for data. It is useful when the application wants a periodic stream of data with a fixed predefined period. Latency budget is not something that can be enforced, instead it is a sort of hint to the underlying layer. It specifies the maximum acceptable delay from the time the data is written until the time the data is inserted in the receiver's application cache. It can be read as the maximum latency. Finally Transport Priority is a hint to the lowest DDS layer to set the priority of the underlying transport layer. See Fig. 7.1 for a complete overview.

The main problem of DDS is that it is totally agnostic to the underlying transport layer, from one side this behavior provides a high level of flexibility, but, on the other side, it does not allow a fine-grained control of the delays that can be raised by the network between Publishers and Subscribers. Especially in LLN networks, which are more dynamic than traditional networks, the monitoring of the network is of paramount importance in order to avoid, or at least reduce, the latency experimented which may violate QoS agreements.

MQTT is a Broker-based publish/subscribe messaging protocol that runs over TCP. Senders deliver their messages to the Broker under a certain topic, the Broker forwards messages to Receivers that have been registered for that topic. The delivery protocol is concerned solely with the delivery of an application message from a single Sender to a single Receiver. When the Server is delivering an Application Message to more than one Client, each Client is treated independently.

From the QoS point of view MQTT provides three different level of QoS: At most once, At least once, Exactly once. The At most once profile delivers message based on Best Effort. No response is sent by the receiver and no retry is performed by the sender. The At least once, instead, requires an acknowledge message from the Receiver. Messages are assured to arrive but duplicates can occur. Finally the Exactly once requires a two-way acknowledgment mechanism which ensure that the message is delivered without any duplicates.

It is worth to note that, even if MQTT provides three different level of QoS, the QoS guarantees provided do not involve timeliness properties. MQTT can only ensure a cer-

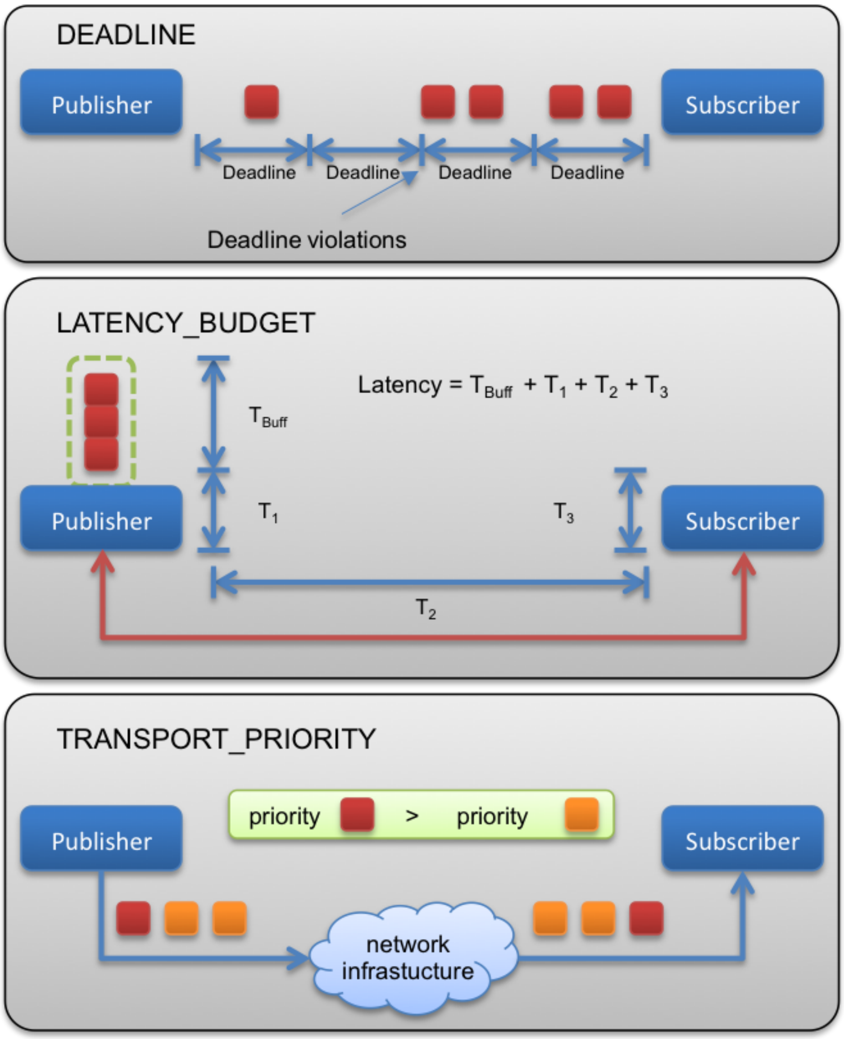

Figure 7.1: DDS QoS architecture

tain level of delivery which may affect indirectly the latency of the information retrieved by Receivers, but it does not provide any negotiation nor monitoring phase.

7.2 Architecture overview

We consider a classic scenario, we have a set of WSNs connected to clients by means of a dedicated gateway which acts as a men in the middle. As already pointed out we focus on how to provide QoS support on top of the Observing paradigm [8], however we would like to provide an overview of a possible real use case.

First of all the gateway discovers all the resources available in its WSN, to achieve this multicast discovery as well as static configuration can be exploited.During the discovery process the gateway populates its remote resources directory which will be used to reply to clients on behalf of the origin servers.

The remote resources directory can be seen as a directory hosting links to resources on remote origin servers, whenever the gateway discovers a resource it creates a new link in its remote resource directory by performing a namespace translation. In other words, each link in the remote resource directory is structured as following:

<origin-server-id>/<resource-name>

thanks to that clients can discover available resources by interacting only with the gateway which acts as a reverse proxy.

Once the discovery procedure ends, the gateway is ready to be used as the intermediary entities between the origin servers and the clients.

In every QoS framework we can identify a negotiation phase. This phase takes place between the service consumer and the service provider, and it is used by the consumer to specify the QoS requirements for the requested service. The service provider, based on the status of the system and on the QoS requirements, can allow or deny the service request.

In our architecture the service provider is the gateway which exploits the CoRE interface draft [10] and especially pmin and pmax for the negotiation phase. The pmin parameter (Minimum period) means that a notifier should wait for at least pmin seconds before sending a new notification message. The pmax (Maximum period) instead, indicates the maximum period between two consecutive notification message sent by the notifier to the client. Clients exploit these two parameters to communicate to the gateway their QoS requirements. Each client, in fact, has its preferred notification period, however it is common that a sort of jitter is tolerated if between certain bounds. By using ranges of notifications periods gives to the gateway a certain level of flexibility when it has to select the best period to be set on the origin server as explained in much detail below.

From a procedural point of view, a client sends a PUT request to the gateway to a URI as the following:

<origin-server-id>/<resource-name>?pmin=5&pmax=10

then it sends a normal observing request to the proxy. The meaning of the messages together is that the client wants a new notification message about the state of the resource every X seconds, where X must be between 5 and 10 seconds.

The gateway collects all the incoming requests for a certain resource and then selects the best period to be set on the origin server following a custom scheduling policy. The overall negotiation procedure is displayed in Fig. 7.2.

After that, when the gateway receive a notification message from the origin server, it checks its internal list of QoS observers for the notified resource through a dedicated Notification Task. In particular, for each client the timestamp of the last notification sent is

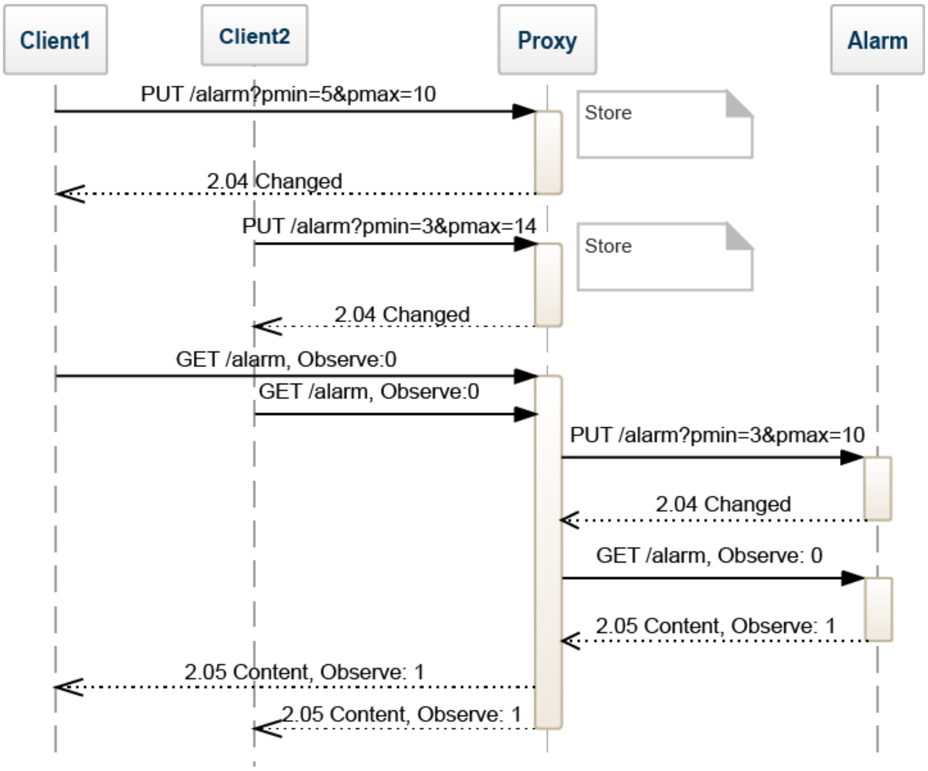

Figure 7.2: Negotiation Procedure

stored so the Notification Task can verify if it is time to send a new notification message or, instead, it is still to early. In the latter case the Notification Task skips the client and proceeds with the others. When all clients have been processed the gateway suspends the Notification Task until a new notification message incomes or a client deadline is quite expired. The internal structure of the Notification Task, in fact, uses a modified sleep function where, on one hand, the duration of the sleeping period is computed according to the minimum remaining time before deadline expiration while, on the other hand, a new incoming notification always interrupt the sleeping period - see Fig. 7.3.

Let's make an example to better explain the concept. Suppose that we have two clients with pmin and pmax as (5,15) (7, 12) respectively. Now let's suppose that the origin server use a notification period of 6, and that both client have been notified at t=0. At t=6, the gateway receive a notification which can be forwarded to the first client, however it is still to early for the second client so the notification thread is suspended. Anyway, the notification thread already has a new notification for the second client that can be sent only after t=7 so the thread go to sleep only for 1 second. At t=7 the notification thread wakes up and sends the notification message to the second client. This process has been developed in order to try to achieve the best QoS for all clients by sending notifications as soon as possible. In the previous example, in fact, the second new notification will be

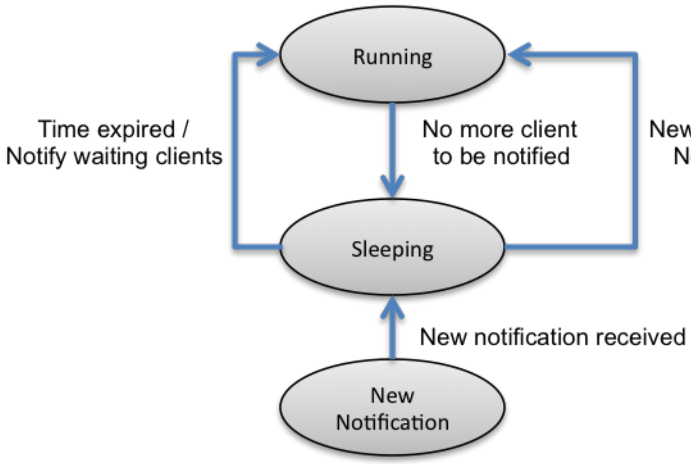
Running
Time expired / Notify waiting clients
No more client to be notified
New notification / Notify clients
Sleeping
New notification received
New Notification

Figure 7.3: Notification Task

supposed to arrive at t=12, which will be enough also for the second client but with a lower quality level.

7.3 Scheduling Policy

In order to guarantee the QoS requirements specified by the clients the gateway collects all requirements and accepts only requests that can be satisfied. In particular the gateway estimates the delay between itself and the origin server through the CoCoA extension [11], and uses it as the limit to accept or reject clients.

The RTT has been chosen as an input to the scheduling policy in order to overcome the CoAP limitations regarding the freshness of data. In other words a CoAP endpoint does not estimate in any way the "age" of a packet upon receive, although delays in the network may affect the freshness of the information. From the CoAP RFC [7]:

> *The mechanism for determining freshness is for an origin server to provide an explicit expiration time in the future, using the Max-Age Option. The Max-Age Option indicates that the response is to be considered not fresh after its age is greater than the specified number of seconds.*

However, the above definition does not consider delays introduced between CoAP endpoints. By using the RTT estimation we can overcome such limitations and derive the received information "age" in order to avoid the dispatching of old information to observer clients.

It is important to outline that the RTT estimation is obtained by exchanging periodic requests, originated by the gateway, with the origin server. In such way the delay computed considers also the processing delay that the origin server may require in order to

```
   Algorithm Scheduler
2  Input: $tasks[n]$, $rtt$, $STEP$
   Output: $period$
4
   $minimum \leftarrow \min\{tasks.pmax\}$
6  $end \leftarrow False$
   while not $end$ and $minimum > rtt$ do:
8    $end \leftarrow True$
     foreach $t$ in $tasks$ do:
10     # Valid  as progression?
       if  $minimum < t.pmin$ then:
12       $tmp \leftarrow 2minimum$
         while $tmp \leq t.pmax$ and $tmp < t.pmin$ do:
14         $tmp \leftarrow tmp + minimum$
         if  $tmp > t.pmax$ then:
16         # Reduce the minimum
           $minimum \leftarrow minimum - STEP$
18         $end \leftarrow False$
           break
20 return $minimum$
```

Figure 7.4: Scheduler.

serve a specific request. Anyway, the period of the evaluation task must be set according to a trade-off between quick detach of delay changes and communication overhead in the origin servers network. In the following we choose a period equal to the double value of the minimum pmin. The Scheduler gets in input all requested ranges as well as the estimated RTT and try to derive a unique period that can fit all requests following the algorithm in Fig. 7.4.

The rational behind the algorithm is to find the best minimum period that can fit inside all the requested ranges. First of all the algorithm starts from the minimum pmax, then it verifies if such period is greater than all the pmins. If it is the case that the algorithm stops and the period is the minimum pmax.

Otherwise, the scheduler verifies if a mathematical progression of the minimum period is within the range of all the requests, in other words, it may happen that the minimum period is to fast for a certain request and will generates a notification too early, however following notifications will arrive within the range of the request and the notification message will be able to be used to notify the client.

The mathematical progression check is an iterative process, it starts by doubling the actual minimum period (Line 12) and verifies if the new virtual period obtained is grater than pmin, if not the virtual period is incremented to check if the third notification will fall within the range (Line 14) and so on until the virtual period is greater than pmin or the virtual period is greater than the allowed pmax (Line 13). If the virtual period obtained is greater than pmax, than it means that the starting minimum period is not suited for all the tasks, the minimum is decremented (Line 17) and the overall process restart from the beginning (Line 5). The algorithm ends when a unique period is find (Success) or when the minimum period that should fit all requests is less than the RTT (Failure). In the latter case, the scheduler remove the request that has the minimum pmax and restart the algorithm until a solution is found or all requests are discarded.

It is worth to note that, obviously, the problem could be solved also by means of the $g.c.d$ between all pmaxs, however it will lead to a suboptimal solution, with our algorithm, instead, we can choose a period greater than the $g.c.d.$ that means less exchange of messages from the origin server and thus saving energy.

7.4 Monitoring

In order to satisfy QoS agreements the gateway must continuously evaluates the delays in the LLN to check the freshness of the information retrieved from the origin server. If the RTT between the gateway and the origin server changes above a certain threshold, the gateway re-evaluates the accepted requests through the scheduler and, eventually, changes the notification period on the origin server or removes the clients that cannot be served anymore.

To better understand the behavior let's make an example as in Fig. 7.5. Suppose we have two clients that are interested in the same resource with pmin=1 and pmax=2, for the first client, and pmin=2 and pmax=12 for the second client. Initially the delay introduced by the LLN is 1 second and the period computed by the scheduler according to Fig. 7.4 is 2. Following, when a notification arrives to the gateway, the gateway infers that the notification has been generated 1 second before. So if the exchange starts at $t = 2$, the gateway will receive the notification message at $t = 3$ and can forward such notification to both clients. The next notification will arrive on the gateway at $t = 5$ and again it can be forwarded to both client because the "age" of the information is only 1 second. Now suppose that the LLN is congested and the delay increase up to 5 seconds. This means that when a notification arrives on the gateway the "age" of the information is 5 seconds, which is too much for the first client. The gateway discovers the new RTT and re-run the scheduler algorithm which will remove the first client because it could not be served with the new RTT.

7.5 Performance Evaluation

In order to evaluate the proposed solution we run several test with different client and different network configurations.

In particular we arbitrary introduces delays between the gateway and the origin servers to emulate congestion in the LLN. To achieve this we use the dummynet software [61] which is a live network emulation tool. It can be used to add custom delays

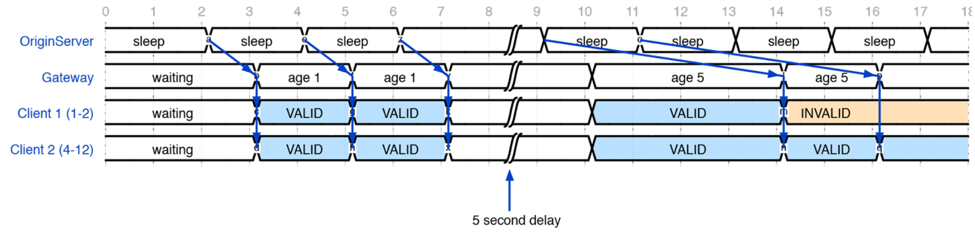

Figure 7.5: Delay increase

in the stack buffers at the kernel level, only for certain ip addresses. The clients, the gateway and the origin servers all run on a Intel 4-core i5 @2.67 GHz processor but on different virtual networks. The network of the origin servers is managed by dummynet in order to introduce arbitrary delays, while the gateway and the clients run on two different virtual networks. In particular the delay between the clients and the gateway has been considered negligible because below 0.1 seconds.

We set up an experiment with 50 pseudo-randomly generated clients, where each client sends an observing request to the gateway for the same resource but with different acceptable QoS ranges. In detail, the period of each client has been chosen following an uniform random distribution between 1 and 20 seconds.

In the first phase of our experiment, which can be seen in Fig. 7.6, all clients are registered in the gateway and all of them receive notifications according to their requirements. In particular the RTT evaluated by the gateway is less than 0.5 sec so a period equal to

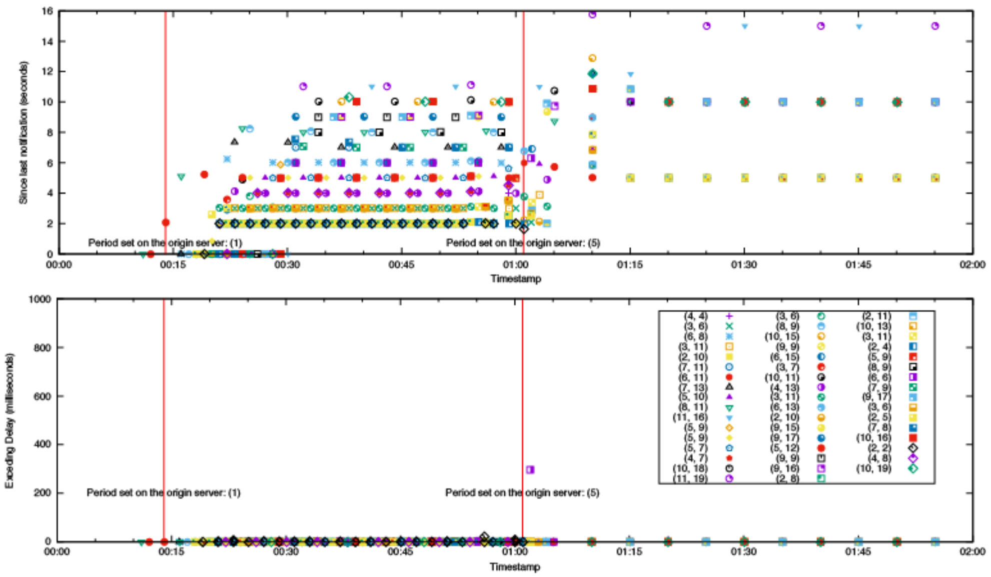

Figure 7.6: Results

1 second can be set on the origin server. It is worth to note that such period has been chosen by the gateway according to the algorithm presented in 7.4. In fact, 1 second is the only period that can be used to accept all clients, due to four clients with configuration parameters set to (4, 4), (6, 6), (9, 9) and (2, 2) respectively - see the graphs key.

After 1 minute a delay of 5 seconds is introduced in the origin servers network and the gateway reacts consequently. All the clients that cannot be served anymore are removed and the period on the origin server is updated according to the scheduler algorithm.

It is worth to note that, thanks to the modified sleep method the client identified by (11, 19), see the last row of the first column in the graph key, always receive the best possible QoS treatment. In the first part of the top most graph in Fig. 7.6 it can be seen that the client receive a notification every 11 seconds (equal to pmin) even if the gateway receives a new notification also on t=12, t=13 etc. In the second part, instead, when the period is set to 5 seconds the above client receives a notification every 15 seconds without violating its deadline, which is, again, the best notification period possible with respect to the delay set into the origin servers network.

Finally, except some transitory phases around the introduction of delays all deadlines are respected for all clients, the transitory phases can be explained by means of the time needed by the gateway to discover the new delay in the origin server network.

7.6 Conclusions

In this chapter we enriched the standard CoAP protocol with by adding the QoS features. We explicitly use the IETF CoRE Interface draft for negotiation purposes in order to provide a standard solution. Specifically, we modified the observing paradigm in order to allow a periodic observing, where each client was able to specify the timeliness properties of the observing relationship, which also considers delays that may occur in LLNs. We demonstrated that the proposed scheduling algorithm can effectively assure the respect of deadlines within the negotiated boundaries as well as the deletion of clients that cannot be satisfied due to delays in the LLN.

8

CoAPthon: Easy Development of CoAP-based IoT Applications with Python

In this chapter we present CoAPthon [62], an open-source Python-based CoAP library, which aims at simplifying the development of IoT applications. The library offers software developers a simple and easy-to-use programming interface to exploit CoAP as a communication protocol for rapid prototyping and deployment of IoT systems. The CoAPthon library is fully compliant with the CoAP RFC and implements in addition popular extensions such as the block-wise transfer and resource observing.

8.1 CoAP implementations

Name	Language	Observe	Block-wise	CoRE	Proxy	Target
Erbium [63]	C	X	X	X	-	Constrained
microcoap [64]	C	-	-	-	-	Constrained
CoAPTinyOS [65]	nesC	X	X	-	-	Constrained
CoAPSharp [66]	C# (Micro .Net)	X	X	X	-	Constrained
node-coap [67]	javascript	X	X	X	Forward	Web
Ruby coap [68]	Ruby	X	X	X	-	Test
iCoAP [69]	Objective C	X	partial	X	-	iOS
Californium [54]	Java	X	X	X	Forward	Backend-Server Embedded Android
libcoap [70]	C	X	X	X	-	Embedded Constrained
txThings [71]	Python	partial	partial	X	-	Backend-Server
openWSN coap [72]	Python	-	-	-	-	Embedded

Table 8.1: CoAP Implementations

Different CoAP libraries have been implemented since the beginning of its specification. Started as a reference implementations aimed at driving the standardization process, some of them successfully became mature software after the protocol standardization. In the following, we provide an overview of the major available CoAP implementations – see Table 8.1. Projects are grouped according to their target environment. For an exhaustive presentation of all CoAP implementations, the interested reader is referred to [73].

The first group of implementations we consider is the group of projects that support constrained devices, such as sensors or micro-controllers. Within this group, Erbium is the most popular project as it implements CoAP for Contiki OS [74], one of the most popular operating systems for constrained devices. The footprint of the library is very small and many configuration parameters are tunable. Other similar implementations are available for other operating systems for constrained devices, such as CoAPTinyOS and micro-coap, tailored for Tiny OS [75] and Arduino [76], respectively. CoAPTinyOS is a CoAP library written in nesC implementing the draft-13 version of the protocol. Micro-coap, instead, is written in C and implements the basic functionalities of the protocol for Arduino devices (e.g. Arduino Mega) on top of Arduino libraries. Although not widely used, the project CoAPSharp is worth of a specific mention as it is the first CoAP implementation in C# that works on the Microsoft .NET micro framework. Although the use of .NET micro framework enables CoAPSharp to run both over constrained devices and regular PCs running Microsoft Windows, the usage of MS Windows as operating system for embedded devices is limited and only at its early stage.

The second group of implementations includes projects that aim at bringing CoAP to more powerful devices, for instance Embedded Systems, such as the ones adopted by makers to build IoT systems, or Backend deployments. In order to guarantee portability among different architectures, the language adopted by these projects is Java, which guarantees platform independent programming. Within this group, the most widely adopted implementation is Californium. The peculiarity of this CoAP library is its modular structure that guarantees easy expandability and customizability. Simple configuration is guaranteed by means of a configuration file that is used to specify the overall behavior and align the software with the underlying hardware. Although Californium is designed for backbone IoT deployments, integrated for example into the cloud [54], the availability of the Oracle Java Embedded platform [77] for embedded devices allows Californium to run also on devices with reduced memory and limited computational capabilities. In addition, the portability of Java allows Californium to be exploited to integrate CoAP functionalities within mobile applications for Android devices.

Another CoAP implementation widely adopted in several projects is libcoap, a CoAP framework written in C. Its lightweight structure not specifically tailored for any OS made this library popular a basis for different other software deployed for both Linux OSs and constrained OSs (Contiki OS). Despite its simple structure, the library does not provide high level APIs, hence it is not easy to be used by software developers, as demonstrated by its limited use for end-user applications.

Finally, three niche CoAP implementations are worth to be mentioned: node-coap, a CoAP implementation written in JavaScript to be use for web applications, Ruby CoAP, a CoAP implementation written in Ruby mainly for testing purposes and iCoAP, a CoAP library written in Objective C to integrate CoAP functionalities into iOS applications. In this context, CoAPthon is the first CoAP implementation that is oriented by design to offer an easy-to-use programming interface to simplify application development for both embedded and backend systems. In comparison to the above mentioned CoAP implementations, CoAPthon is an attractive solution for IoT software developers as it is aligned with the standard version of the protocol and offers the implementation of a large set of features, such as CoRE link format parsing, resource observing and Block-wise transfer. CoAPthon implements also the proxy functionality, in both reverse and forward modes, which are rarely available together in other libraries. In addition to this, CoAPthon is the first mature CoAP implementation developed in Python. To the best of our knowledge, there are two different Python implementations: txThings [71] and openWSN coap [72]. At the time of writing, the former provides only a small subset of the features provided by CoAPthon and it has been implemented with Python 3.4, which is not yet supported by many embedded devices. The latter, instead, is part of a bigger framework called openWSN [78]; the CoAP implementation, however, is still in its early phase and does not handle many of the standard features as well as its extensions.

8.2 CoAPthon

The CoAPthon framework architecture is depicted in Fig. 8.1. The library is built on top of the Twisted framework [79]. Twisted is an event-driven networking engine for Python that implements several application protocols, such as HTTP and FTP, and can be easily extended to implement new protocols based on both TCP and UDP transport layers. Twisted has been selected not only for its small codebase and lightweight implementation, but also for its robustness, as acknowledged by its adoption by many commercial and open-source projects. Its implementation of all low-level networking and management functionalities, such as multi-threading, UDP socket management, and Asynchronous message exchange, is beneficial to CoAPthon since it allows to easily support Multicast CoAP server discovery.

The layered architecture of CoAP is reflected in the CoAPthon architecture illustrated in Fig. 8.1, which presents at the bottom the Message layer and at the top the Request layer. A middle layer called Extension layer is also introduced to implement extended functionalities on top of the basic ones, e.g., resource observing and Block-wise transfer.

The Message Layer implements the Message sub-layer of the CoAP protocol. It is in charge of pairing messages based on the MID header field. In addition, it is responsible for managing the separate mode (enabled by default): when the server receives a CON request a timer is triggered; if the server does not complete the processing of the request before the timer expiration, an ACK message is sent back to notify that the server will replay later.

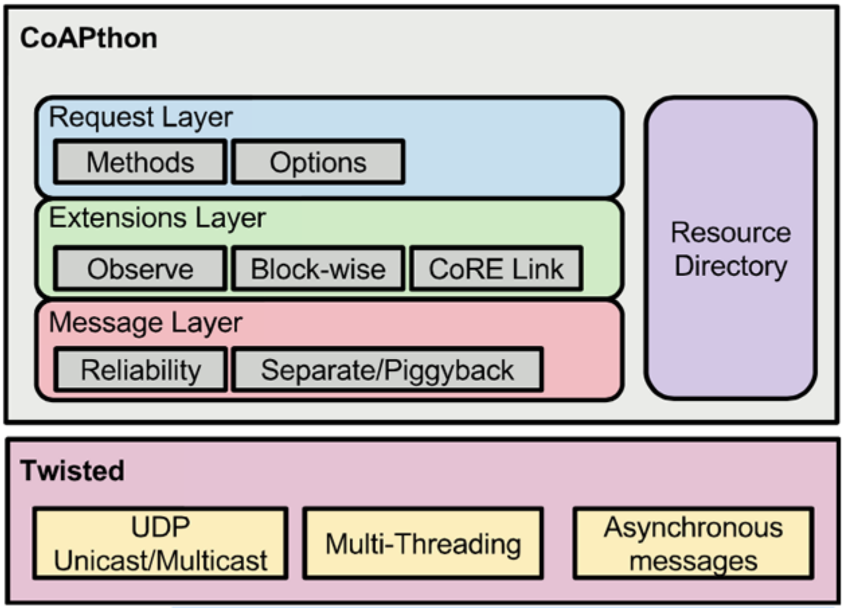

Figure 8.1: CoAPthon implementation architecture.

The Request Layer implements the Request/Response sub- layer of the protocol and is responsible for handling CoAP re- quests to produce responses. To this aim, a Resource Tree is implemented to store information related to resources exposed by a server, and pairs each resource with the corresponding handler function designated to produce CoAP responses. When a request is received, the Resource Tree is accessed to find the resource required by the client, then the associated handler is executed on the method specified by the CoAP request. CoAPthon is implemented following a resource oriented approach, i.e., a resource must implement one handler for each of the CoAP methods (GET, PUT, POST and DELETE) that it wants to expose. If a client requests a method that is not implemented, a “Not Allowed” response is sent by default. In addition, dynamic creation of resources is allowed. An example could be the creation of a new resource as a result of a POST request.

The Extension Layer is a container for all those functionalities that are not included in the basic CoAP specification. At the time of writing, CoAPthon implements the resource observing, the block-wise transfer and the CoRE Link Format parsing features. By default, they are enabled and are automatically activated when needed. When a client requests an observe relationship on a resource that has been set as observable, the library initializes the observe relationship, and will send notifications to subscribers as soon as the payload of the resource changes. Similarly, when the response payload is bigger than a single CoAP message, or when the client explicitly requests a Block-wise transfer, the library activates the block-wise transfer. The CoRE Link feature, instead, is involved in the discovery process, performed by a client on a CoAP Server, and in the creation of resources on CoAP servers. A resource, in fact, can be enriched with CoRE Link attributes, which will be automatically communicated to clients upon discovery requests. It is worth to mention that the Extension Layer is designed with a modular structure to allow further extensions with new features.

8.3 Usage Example

The modularity of the CoAPthon architecture facilitates the development of CoAP endpoints within applications. To this aim, a simple set of APIs has been defined for implementing not only the basic Client/Server CoAP roles, but also Forward and Reverse Proxy functionalities. To ease the developer experience, the library already provides different examples of development, which can be exploited as a basis for more complex custom implementations. In the remainder of this section, we present an example development using the APIs exposed by CoAPthon in order to show how the APIs design jointly with the easy syntax of Python can be used to rapidly develop IoT applications. For the sake of brevity, only the development of a CoAP server is shown: the interested reader is referred to the other examples included in the official distribution.

Let us consider a Raspberry Pi [16] board that has a sensor connected to one GPIO pin. In this example, it is shown how to implement a CoAP server that exposes a resource whose state is the reading value from the sensor. The RPi.GPIO Python library is employed in this example to implement the communication with the GPIO interface.

The definition of a CoAP server is performed in two steps: (i) definition of resources to be exposed; and (ii) definition of the server engine. In order to define a new resource the developer must define a new class, which extends the base superclass Resource. In Fig. 8.2 it is shown how to develop a new resource that replies to GET requests with the status of a specified GPIO pin. The new class 'RPiResource' is defined to initialize the GPIO library and to handle GET requests on the resource through the method 'render_GET'. A similar approach must be followed to define resources that reply to POST, PUT or DELETE requests.

Resources must be linked to a CoAP server. To this aim, a new CoAP server must be defined through the definition of a new class, which extends the base super-class CoAP.

```
class RPiResource(Resource):
  channel = 1
  GPIO.setmode(GPIO.BOARD)
  GPIO.setup(channel, GPIO.IN)

  def render_GET(self, request):
    self .payload = GPIO.input(self.channel)
    return self
```

Figure 8.2: Resource Definition.

Fig. 8.3 shows the procedure to define a new CoAP server. The server is configured with one resource of type 'RPiResource' through the add_resource method. The first parameter of such method must be the URI of the new resource, in this case 'pin1', while the second parameter is an instance of the resource class, RPiResource in the example. As can be seen, the resource is set as observable by simply setting the flag observable to true. Finally, the CoAP server is initialized to listen on localhost to the default CoAP UDP port.

There are practical cases in which a request requires a certain amount of time for processing, e.g., sensors that require a start reading command on a pin and produce a valid result on another pin after a delay. In this case, the separate mode can be employed to acknowledge the request immediately and send the response separately as soon as it is available. The procedure to activate the separate mode is illustrated in Fig. 8.4: the handler returns a reference to a callback function for generating the response ('render_GET_separate' in the example), instead of returning directly the response message. With this configuration, the CoAP server immediately acknowledges the client and then invokes the provided method to generate the response.

Finally, we introduce the approach adopted to trigger observing notifications because of a change of status. Considering that a resource modifies its status according to external events, often detected by external processes, CoAPthon implementation detaches event detection from the observe notification process to ease integration. In particular, notifications are triggered by issuing a PUT request on the resource, allowing external processes to notify the change of status to the CoAP server in a simple and standard manner. To this aim, an observable resource must implement a special PUT handler that updates the resource state according to the PUT payload, triggering the delivery of notifications to subscribers. In case the CoAP server is instead directly responsible for updating the state of a resource, the PUT method can be invoked internally. An example of this approach is provided in Fig. 8.5: the 'callback' function, invoked automatically by the GPIO library when the value on one GPIO pin raises, issues a PUT request on the loopback interface to trigger observe notifications.

```
class CoAPServer(CoAP):
  def __init__(self, host, port):
    CoAP.__init__(self)
    self.add_resource('pin1/', RPiResource("pin1", observable=True))

server = CoAPServer("127.0.0.1", 5683)
reactor.listenUDP(5683, server, "127.0.0.1")
reactor.run()
```

Figure 8.3: Server definition and initialization.

```
class RPiResource(Resource):
  ...
  start_read = 2
  GPIO.setup(start_read, GPIO.OUT)

  def render_GET(self, request):
    return self, self.render_GET_separate

  def render_GET_separate(self, request):
    GPIO.output(start_read, GPIO.HIGH)
    time.sleep(known_wait_time)
    self.payload = GPIO.input(self.channel)
    return self
```

Figure 8.4: Separate Mode.

```
def callback(channel):
  client = HelperClientSynchronous()
  client.put({"path": "coap://127.0.0.1:5683/pin1", "payload": GPIO.input(channel)})

class RPiResource(Resource):
  ...
  GPIO.add_event_detect(channel, GPIO.RISING, callback=callback)
  ...

  def render_PUT(self, request):
    self.payload = request.payload
    return self
```

Figure 8.5: Resource with Observing.

8.4 Performance Evaluation

We have run several experiments in order to assess the performance of CoAPthon against Californium in different scenarios. Our goal is to show that the proposed implementation in Python does not pay a considerable price in terms of performance to favor ease of usage. Our experiments focus on evaluating the performance of the server implementation, which is the bottleneck in a real use case considering that its performance determines the number of concurrent requests that can be handled by a sensor.

In the first experiment, a benchmark of the server implementation is performed. In particular, a CoAP server, implemented in both Californium and CoAPthon, is deployed to expose a resource that returns a simple response with a fixed numeric value. The CoAP server runs on a Raspberry PI Model B, equipped with a single core ARM System on a Chip at 700Mhz with 512 Mb of RAM installed. The behavior of a set of CoAP clients issuing requests at a fixed rate is mimicked through coapbench, a Java benchmark tool that sends continuous CoAP requests at a given rate. Clients run on a PC equipped with an Intel Quad-core 3 GHz processor with 8 GB of RAM. The Raspberry Pi and the PC are connected by means of a 100 Mbps FastEthernet LAN switch. For each experiment, an overall number of 10000 requests are issued.

Fig. 8.6 shows the response delay, measured as the time between the delivery of the request and the reception of the response, with different request rates. The average value is estimated with a 95% confidence interval. As expected, Californium guarantees lower response times than CoAPthon. This is expected since the performance of Python as a programming language favors the ease of programming over efficiency [80]. Although high request rates are unlikely for a scenario involving embedded/constrained devices, it is worth to highlight that the CoAPthon performance does not degrade significantly with rates up to 1000 and 1500 requests per second.

In the second experiment, a benchmark of the performance of the proxy implementation is carried out. In particular, a CoAP forward proxy (the only one implemented in Californium) is installed to expose the resources hosted on 20 CoAP servers, emulated through a process running on a PC with an Intel Dual-core 2.4 GHz and 2 GB of RAM. The proxy runs on a UDOO board, a more powerful embedded device equipped with a Quad-core Cortex-A9 CPU at 1000 Mhz and with 1 Gb of RAM [15]. Clients are emulated using the same methodology of the previous experiment except for the request rate,

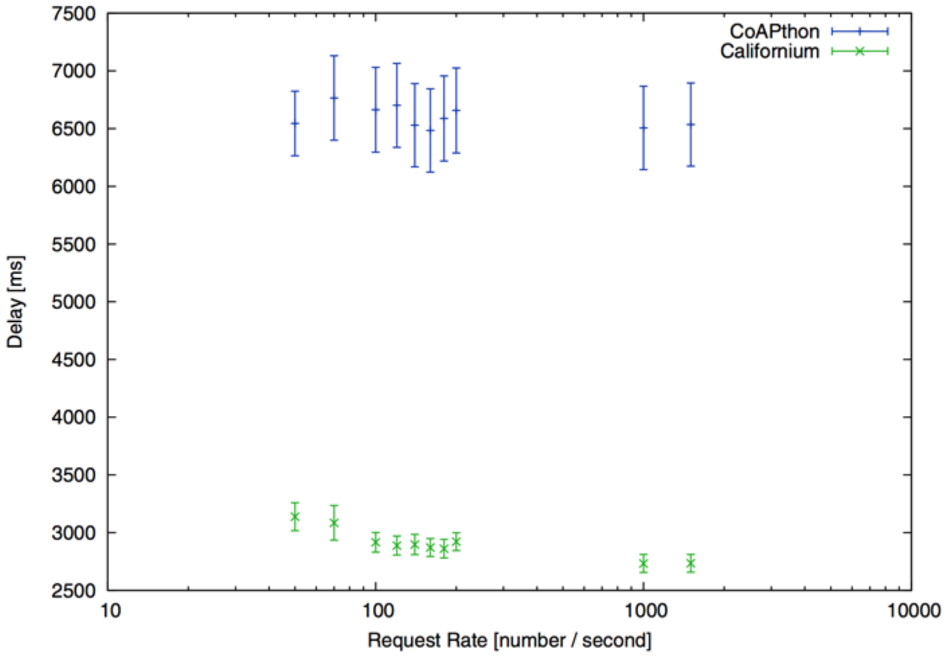

Figure 8.6: Request delay, Client – Server.

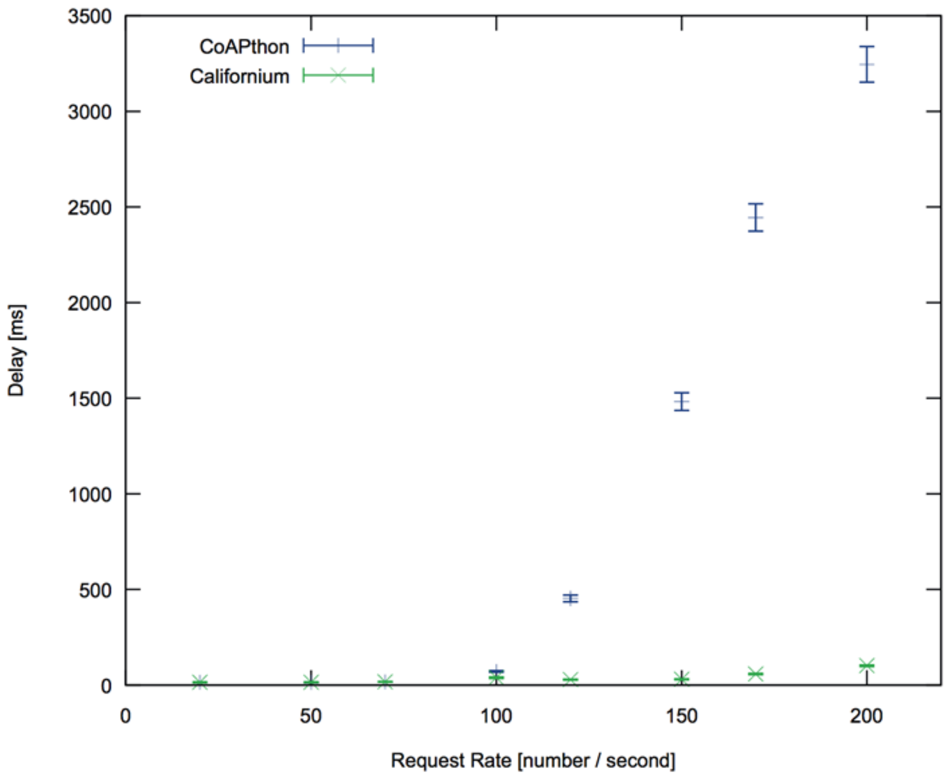

Figure 8.7: Request delay, Client – Proxy - Server.

which follows a Poisson distribution to model the aggregate rate of simultaneous CoAP sessions directed to different servers at the same time. For each experiment, an overall number of 10000 requests are issued.

Fig. 8.7 shows the average delay obtained with increasing values of the average arrival rate λ. As can be seen, proxy implementation in CoAPthon manages a flow of requests up to 100 requests per second However, when the load increases above this threshold, the response delay increases steeply. This behavior can be explained by considering that Python does not fully exploit concurrency because of the well-known issue that characterizes all of its interpreters [81].

8.5 Conclusions

In this chapter, we presented CoAPthon, a Python implementation of the CoAP protocol that aims at enabling the development of IoT application based on the CoAP protocol through a simple and easy to use programming interface. To this aim, different usage examples have been presented to show how a simple CoAP server can be rapidly implemented with CoAPthon, and how different configurations can be implemented with a few changes in the code. The performance evaluation of the proposed framework against Californium through real experiments highlighted a trade-off between ease of development and performance. In particular, it is shown how usability has a price in term of response delay and scalability, which, however, can be considered acceptable in the considered scenario.

9

Conclusions

In this thesis we study the QoS support in the IoT field. In the first part I we provide the Thing as a Service architecture which we use as a reference model for all the QoS considerations in the fog layer. We divide this part of our work in three main sections following an implementation focus approach.

First of all we define the negotiation framework involved between applications and the architecture, thus we design an abstract procedure which is suitable to manage QoS reservation into the fog layer where IoT gateways are distributed close to the access network. Finally we solve the Thing Selection problem by the developing of a new algorithm which takes care of the unique features of smart things. In particular, we solve a Mixed Integer Problem to reduce the energy consumption on smart things in order to maximize the lifetime of smart devices.

Eventually we achieve the following results:

- We derive a novel QoS model tailored for M2M application from the existing QoS models.
- We design a QoS procedure designated for large heterogeneous IoT systems where specific attention to scalability is given in the design at any rate.
- We develop a heuristic algorithm to solve the Thing Selection problem in a polynomial time in order to allow its implementation in real systems.

In the second part of this manuscript II, we focus on the Access Network, and, in particular, on how to provide QoS support over the CoAP protocol.

We designed a new architecture which exploits gateways, that are located between WSNs and clients, to provide the QoS support in a transparent way. To this aim we combined different technologies, such as proxying and virtualization, to enhance the standard CoAP protocol without the needs of any modification to clients or smart devices. Experimental results carried on by means of a real testbed demonstrated that the proposed solution succeeds in overcoming the limitations of the classic architectures, without introducing significant overhead.

Then we enriched the standard CoAP protocol with the QoS features by exploiting existing IETF drafts. Specifically, we modified the observing paradigm in order to allow a periodic observing, where each client was able to specify the timeliness properties of the observing relationship, which also considers delays that may occur in LLNs. We demonstrated that the proposed scheduling algorithm can effectively assure the respect of deadlines within the negotiated boundaries as well as the deletion of clients that cannot be satisfied due to delays in the LLN.

A

Constrained Application Protocol

The Constrained Application Protocol (CoAP) is a lightweight RESTful protocol designed for constrained devices. It is based on UDP, and inherits the same client/server paradigm adopted in HTTP. The protocol has a two (sub-)layer structure: a request/response sub-layer and a message sub-layer, respectively.

The request/response sub-layer, similarly to the one included in HTTP, handles CoAP requests, pairs requests and responses by means of tokens, and invokes application methods to generate responses. The message sub-layer, instead, is responsible for managing the message exchange between endpoints over UDP, implementing reliable delivery if enabled. CoAP defines four different types of messages: Non-Confirmable (NON), Confirmable (CON), Acknowledgment (ACK) and Reset (RST). When unreliable message delivery is enabled, NON messages, which do not require confirmation of reception, are employed. When reliable delivery is enabled, requests/responses are transported within CON messages that, instead, require an acknowledgement through ACK messages. The latter can be sent in two alternative modes: separate and piggyback. In separate mode, the receiver sends an empty ACK message immediately after the reception of a request, and defers sending the actual response in a separate CON message, when it is ready. In piggyback mode, instead, the receiver waits for the response to be ready and sends it back directly encapsulated in the body of the ACK message. The sender is responsible for retransmitting messages that are not acknowledged after a default timeout, implementing an exponential backoff between retransmissions. Finally, RST messages are employed to handle error situations, such as a recipient that cannot process a message.

CoAP includes also the definition of a proxy, which is a CoAP endpoint that can issue requests on behalf of a client. Two different types of proxy are defined: the Reverse Proxy and the Forward Proxy. A Reverse Proxy is employed to transparently expose resources hosted on different servers, as if they were hosted by itself, e.g., to simplify the discovery procedure or to transparently implement custom processing (e.g., caching) or enforce access policies. A Forward Proxy, instead, does not expose any resource but can issue on-demand CoAP transactions on behalf of a client, which specifies the destination URI as a string in the Proxy-Uri Option.

In addition to the CoAP protocol, some extensions and features are currently under definition or have been recently defined by IETF. Among them, three ones are worth to be mentioned: Block-wise transfer [82], Resource Observing [8] and the Constrained RESTful Environments (CoRE) Link Format specification [83].

Block-wise transfer enables the transmission of a big bunch of data without IP fragmentation. Constrained networks usually have a limited maximum frame length that requires large data- gram to be fragmented at the IP layer, resulting in inconvenient communication overhead and additional processing for fragmentation and reassembly. Block-wise transfer allows big payloads to be fragmented directly by the application into a chain of messages, each one sent individually to avoid fragmentation at the IP layer.

Resource Observing is a non-RESTful additional mode of operation implementing a publish/subscribe mechanism that can be exploited to reduce the communication overhead: a client registers its interest in the status of a resource to receive asynchronous updates when the status changes. In detail, a client issues a GET request with the Observe option enabled; the server processes the message and registers an observe relationship, whenever the resource representation changes, the server notifies all the observers sending a normal CoAP response.

Finally, the CoRE link format specification defines a link format to be used by CoAP servers to describe hosted resources and specify possible link relationships with other resources. The language is an extension of the Link Header format defined in HTTP, which includes specific attributes typical of constrained environments.

The CoRE link format is exploited during discovery operations. In particular, CoAP defines two procedures to automate discovery: service discovery and resource discovery, respectively. Service discovery allows clients to discover CoAP servers on the same subnet. CoAP servers join the all-CoAP-nodes IPv6 multicast address, listen to the default CoAP port, and replies to multicast messages advertising their presence. Once a server is discovered, a client can retrieve all URIs associated to respective resources hosted by the server with corresponding attributes through a resource discovery. To this aim, a client issues a GET request to the “/.well-known/core” URI to obtain the representation of such resources in the CoRE Link Format.

References

1. Jan S. Rellermeyer, Michael Duller, Ken Gilmer, Damianos Maragkos, Dimitrios Papageorgiou, and Gustavo Alonso. The software fabric for the internet of things. In *Proceedings of the 1st International Conference on The Internet of Things*, IOT'08, pages 87–104, Berlin, Heidelberg, 2008. Springer-Verlag.
2. Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.
3. S. Abdelwahab, B. Hamdaoui, M. Guizani, and A. Rayes. Enabling smart cloud services through remote sensing: An internet of everything enabler. *Internet of Things Journal, IEEE*, 1(3):276–288, June 2014.
4. Patricia Morreale, Feng Qi, Paul Croft, Ryan Suleski, Brian Sinnicke, and Francis Kendall. Real-time environmental monitoring and notification for public safety. *IEEE Multimedia*, 17(2):4–11, 2010.
5. C. Bennett and S.B. Wicker. Decreased time delay and security enhancement recommendations for ami smart meter networks. In *Innovative Smart Grid Technologies (ISGT), 2010*, pages 1–6, Jan 2010.
6. Erik Wilde. Putting things to rest. *School of Information. UC Berkeley: School of Information*, 2007.
7. Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7252.txt`.
8. K. Hartke. Observing resources in the constrained application protocol (coap). RFC 7641, RFC Editor, September 2015.
9. Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, pages 200–, Washington, DC, USA, 2003. IEEE Computer Society.
10. Zach Shelby, Matthieu Vial, and Michael Koster. Reusable interface definitions for constrained restful environments. Internet-Draft draft-ietf-core-interfaces-04, Internet Engineering Task Force, October 2015. Work in Progress.
11. August Betzler, Carles Gomez, Ilker Demirkol, and Josep Paradells. Cocoa+. *Ad Hoc Netw.*, 33(C):126–139, October 2015.
12. Atanas Radenski. "python first": A lab-based digital introduction to computer science. *SIGCSE Bull.*, 38(3):197–201, June 2006.
13. Stefano Bocchino, Szymon Fedor, and Matteo Petracca. *Wireless Sensor Networks: 12th European Conference, EWSN 2015, Porto, Portugal, February 9-11, 2015. Proceedings*, chapter PyFUNS: A Python Framework for Ubiquitous Networked Sensors, pages 1–18. Springer International Publishing, Cham, 2015.

14. Iqbal Mohomed and Prabal Dutta. The age of diy and dawn of the maker movement. *GetMobile: Mobile Comp. and Comm.*, 18(4):41–43, January 2015.
15. UDOO. http://www.udoo.org/.
16. RaspberryPi. http://raspberrypi.org.
17. ArduinoYun. http://arduino.cc/en/main/arduinoboardyun.
18. Venkatesh Rajendran, Katia Obraczka, and J. J. Garcia-Luna-Aceves. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wirel. Netw.*, 12(1):63–78, February 2006.
19. Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 171–180, New York, NY, USA, 2003. ACM.
20. Linfeng Yuan, Wenqing Cheng, and Xu Du. An energy-efficient real-time routing protocol for sensor networks. *Comput. Commun.*, 30(10):2274–2283, 2007.
21. K. Akkaya and M. Younis. An energy-aware qos routing protocol for wireless sensor networks. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 710–715, May 2003.
22. Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. Qos management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, August 2007.
23. T. Cucinotta, A. Mancina, G.F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. A real-time service-oriented architecture for industrial automation. *Industrial Informatics, IEEE Transactions on*, 5(3):267–277, Aug 2009.
24. Flávia Coimbra Delicato, Paulo F. Pires, Luci Pirmez, and Luiz Fernando Rust da Costa Carmo. A flexible web service based architecture for wireless sensor networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, pages 730–, Washington, DC, USA, 2003. IEEE Computer Society.
25. G.F. Anastasi, E. Bini, A. Romano, and G. Lipari. A service-oriented architecture for qos configuration and management of wireless sensor networks. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8, Sept 2010.
26. N. Shankaran, J.S. Kinnebrew, X.D. Koutsoukas, Chenyang Lu, D.C. Schmidt, and G. Biswas. An integrated planning and adaptive resource management architecture for distributed real-time embedded systems. *Computers, IEEE Transactions on*, 58(11):1485–1499, Nov 2009.
27. Yuanfang Zhang, C.D. Gill, and Chenyang Lu. Configurable middleware for distributed real-time systems with aperiodic and periodic tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 21(3):393–404, March 2010.
28. LOLA European project D3.6. Qos metrics for m2m and online gaming. Technical report, EU FP7, 2013.
29. Rongduo Liu, Wei Wu, Hao Zhu, and Dacheng Yang. M2M-oriented QoS categorization in cellular network. In *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, pages 1–5, September 2011.
30. Marie-Aurélie Nef, Leonidas Perlepes, Sophia Karagiorgou, George I Stamoulis, and Panayotis K Kikiras. Enabling qos in the internet of things. In *Proc. of the 5th Int. Conf. on Commun., Theory, Reliability, and Quality of Service (CTRQ 2012)*, pages 33–38, 2012.
31. Zhou Ming and Ma Yan. A modeling and computational method for qos in iot. In *2012 IEEE International Conference on Computer Science and Automation Engineering*, 2012.
32. Ren Duan, Xiaojiang Chen, and Tianzhang Xing. A qos architecture for iot. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 717–720. IEEE, 2011.
33. Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). In *Open Grid Forum*, volume 128, page 216, 2007.
34. Oliver Waeldrich, Dominic Battré, Francis Brazier, Kassidy Clark, Michel Oey, Alexander Papaspyrou, Philipp Wieder, and Wolfgang Ziegler. Ws-agreement negotiation version 1.0. In *Open Grid Forum*, volume 35, page 41, 2011.

35. Ling Li, Shancang Li, and Shanshan Zhao. Qos-aware scheduling of services-oriented internet of things. *Industrial Informatics, IEEE Transactions on*, 10(2):1497–1505, 2014.
36. Daniel A Menascé. Qos issues in web services. *Internet Computing, IEEE*, 6(6):72–75, 2002.
37. Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.
38. Bas Boone, Sofie Van Hoecke, Gregory Van Seghbroeck, Niels Joncheere, Viviane Jonckers, Filip De Turck, Chris Develder, and Bart Dhoedt. Salsa: Qos-aware load balancing for autonomous service brokering. *Journal of Systems and Software*, 83(3):446–456, 2010.
39. Daniel A Menascé, Emiliano Casalicchio, and Vinod Dubey. On optimal service selection in service oriented architectures. *Performance Evaluation*, 67(8):659–675, 2010.
40. Andreas Reinhardt and Ralf Steinmetz. Exploiting platform heterogeneity in wireless sensor networks for cooperative data processing. *Proc. of the GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, 2009.
41. Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.
42. John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.
43. David W Pentico. Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, 176(2):774–793, 2007.
44. Silvano Martello and Paolo Toth. The bottleneck generalized assignment problem. *European journal of operational research*, 83(3):621–638, 1995.
45. Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.
46. Anura P Jayasumana, Qi Han, and Tissa H Illangasekare. Virtual sensor networks-a resource efficient approach for concurrent applications. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 111–115. IEEE, 2007.
47. Monica Navarro, Marcello Antonucci, Lambros Sarakis, and Theodore Zahariadis. Vitro architecture: Bringing virtualization to wsn world. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 831–836. IEEE, 2011.
48. Rémy Leone, Paolo Medagliani, and Jérémie Leguay. Optimizing qos in wireless sensors networks using a caching platform. In *Sensornets 2013*, page 56, 2013.
49. Alessandro Ludovici, Ernesto Garcia, Xavi Gimeno, and A Calveras Augé. Adding qos support for timeliness to the observe extension of coap. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on*, pages 195–202. IEEE, 2012.
50. August Betzler, Carles Gomez, Ilker Demirkol, and Josep Paradells. Congestion control in reliable coap communication. In *Proceedings of the 16th ACM international conference on Modeling, analysis & simulation of wireless and mobile systems*, pages 365–372. ACM, 2013.
51. Linux Containers. https://linuxcontainers.org/.
52. Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.
53. KVM. http://www.linux-kvm.org.
54. Californium CoAP. http://projects.eclipse.org/projects/technology.californium.
55. Zolertia Z1. http://zolertia.com/products/z1.
56. Contiki OS. http://www.contiki-os.org/.
57. Erbium. http://people.inf.ethz.ch/mkovatsc/erbium.php.
58. Object Managment Group. Data distribution service for real-time systems. Technical report, OMG, January 2007.

59. Edited by Andrew Banks and Rahul Gupta. Mqtt version 3.1.1. Technical report, OASIS Standard., October 2014.
60. Object Managment Group. The real-time publish-subscribe protocol (rtps) dds interoperability wire protocol specification. Technical report, OMG, September 2014.
61. Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, April 2010.
62. CoAPthon. https://github.com/tanganelli/coapthon.
63. Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A low-power coap for contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860. IEEE, 2011.
64. microcoap. https://github.com/1248/microcoap.
65. CoAPTinyOS. http://tinyos.stanford.edu/tinyos-wiki/index.php/coap.
66. CoAPSharp. http://www.coapsharp.com/.
67. node coap. https://github.com/mcollina/node-coap.
68. Ruby coap. https://github.com/nning/coap.
69. iCoAP. https://github.com/stuffrabbit/icoap.
70. Libcoap. http://sourceforge.net/projects/libcoap/develop.
71. txThings. https://github.com/siskin/txthings.
72. openWSN coap. https://github.com/openwsn-berkeley/coap.
73. CoAP Implementations. http://en.wikipedia.org/wiki/constrained_application_protocol.
74. Adam Dunkels, Oliver Schmidt, Niclas Finne, Joakim Eriksson, Fredrik Österlind, and Nicolas Tsiftesand Mathilde Durvy. The contiki os: The operating system for the internet of things. *Online], at http://www. contikios. org*, 2011.
75. Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.
76. Arduino. http://www.arduino.cc/.
77. Java Embedded. http://www.oracle.com/technetwork/java/embedded/ overview/index.html.
78. Thomas Watteyne, Xavier Vilajosana, Branko Kerkez, Fabien Chraim, Kevin Weekly, Qin Wang, Steven Glaser, and Kris Pister. Openwsn: a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.
79. Twisted. https://twistedmatrix.com.
80. Python – Java benchmark. http://benchmarksgame.alioth.debian.org/u32/python.php.
81. Python Concurrency Issues. https://en.wikipedia.org/wiki/cpython#concurrency_issues.
82. Z. Shelby C. Borman. Block-wise transfers in coap. Technical report, Internet- Draft draft-ietf-core-block-17.txt„ 2015.
83. Z. Shelby. Constrained restful environments (core) link format. Technical report, RFC Editor, 2012.

Acknowledgments

This work would not be possible without the advice and support of many people. Foremost, I would like to thank Prof. Enzo Mingozzi for giving me the opportunity of doing my PhD and for the hours spent with me in front of a whiteboard or a bunch of printed graphs. I would also like to express my most gratitude to my college Carlo Vallati for his help and for the advices that allowed me to finish my work. I hadn't been able to do this thesis without his support.

I am also grateful to Prof. Friedemann Mattern and Matthias Kovatsch for the support during my visiting period at Zurich ETH. I would like also to thank my colleges for our coffee break conversations which always bring a smile.

A special thank goes to my family, my father and my mother, which helped me more than what a son could expect. My brother which has been always ready to talk in front of a good bottle of wine. Particular thank to my aunt which helped me to revision this work even if there are tons of "obscure" technical words.

I would like to thank my closest friends: Albe, Biondo, Gaddo, Mario, Menga, Piastra (I know, he is also my brother, but I would like to thank him twice), Toro and Zio for their support and for our dinners/wine-tasting/beer-tasting etc. Truly thank you, I hope that we will be toghether for many many years. Thank also to Simona which shared with me the joy and sorrow of doing a PhD, I'm thinking about sending our Skype chats to PhDcomics.

Last but not least, I thank my girlfriend Ilaria, she supported, understood and helped me during all the PhD, and I know that sometimes it has been really hard. I cannot find the right words to express how huge is my thankfulness.