Cluster Comput
DOI 10.1007/s10586-017-0861-0

An ideal IoT solution for real-time web monitoring

Pedro Diogo[1] · **Nuno Vasco Lopes**[2] · **Luis Paulo Reis**[1,3,4]

Received: 28 November 2016 / Revised: 12 January 2017 / Accepted: 6 February 2017

Abstract For the internet of things (IoT) to fully emerge, it is necessary to design a suitable system architecture and specific protocols for this environment. The former to provide horizontal solutions, breaking away the current paradigm of silos solutions, and thus, allowing the creation of open and interoperable systems; while the latter will offer efficient and scalable communications. This paper presents the latest standards and ongoing efforts to develop specific protocols for IoT. Furthermore, this paper presents a new system, with the most recent standards for IoT. Its design, implementation and evaluation will be also described. The proposed system is based on the latest ETSI M2M specification (ETSI TC M2M in ETSI TS 103 093 V2.1.1. http://www.etsi.org/deliver/etsi_ts/103000_103099/103093/02.01.01_60/ts_103093v020101p.pdf, 2013b) and the MQTT protocol (IBM, Eurotech in MQTT V3.1 Protocol Specification pp 1–42, http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf, 2010). With this solution it is possible to show how we can create new applications to run over it and the importance of designing specifically tailored for IoT communication protocols in order to support real-time applications.

Keywords IoT · Standards · Horizontal solution

✉ Pedro Diogo
pedrodiogoum@gmail.com

Nuno Vasco Lopes
nvlopes@dsi.uminho.pt

Luis Paulo Reis
lpreis@dsi.uminho.pt

[1] DSI/EEUM – Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Guimarães, Portugal

[2] Department of Information Systems, ALGORITMI Centre, University of Minho, Guimarães, Portugal

[3] Centro ALGORITMI, Universidade do Minho, Guimarães, Portugal

[4] LIACC – Laboratório de Inteligência Artificial e Ciência de Computadores, Porto, Portugal

1 Introduction

The internet of things is a new technology which is rapidly evolving. Gartner, the world' s leading information technology research and advisory company, said, in December 2013, that the internet of things will grow to 26 billion units in 2020, resulting in 1.9$ trillion in global economic value-add through sales into diverse end markets [1]. Similarly, Cisco also said that it will create, from 2013 to 2022, a 14.4$ trillion of value at stake for companies and industry [2]. However, as the IoT is not just about connecting things to the Internet, but a complex technology composed of a seamless intercommunication and coordination of objects, data, processes and services, there is a need to make use of standardized open architectures and protocols to make all of these interactions as easy and efficient as possible. Furthermore, its complexity covers almost the whole open systems interconnection model (OSI), with different protocols being adapted and created to meet the needed requirements for the technology to emerge. because of this it is difficult to define a single solution for every application domain. Furthermore, not every IoT system needs the same protocol stack and/or overall architecture—given the specific application, some protocols and/or architectural guidelines should be adopted to meet the requirements of that particular application. However, for IoT to be fully expanded and have smarter and more rich applications, it is crucial to design scalable and interoperable systems. Its richness lies on this paradigm: a complex vast

Published online: 18 May 2017

coordination of sensed data (typical constrained devices), its processing (cloud services, for instance) and overall coordination of the corresponding action triggers which might occur at different IoT systems. As a simple example, one can think of a seamless interconnection of a simple wearable device monitoring a person's heart rate (typical current silos and proprietary applications) with another IoT system—a clinical facility, as an example. This concept is illustrated in Fig. 14. The figure shows the ETSI M2M' s entities called SCL (Service Capability Layer) and its standardized reference points mId, mIa and dIa makes all of this interconnection possible.

In essence, there is a need to define a way of intelligently interconnect different IoT systems. This is possible by adopting the latest, specifically defined for the IoT, protocols and standards and by interworking different technologies. As a mean to achieve such a thing, a set of abstraction layers combined with a RESTful API are the basis needed to build such a system. This eases both the integration of each system on the web—crucial for the technology to fully emerge as it could offer a simple UI for the end user -, and the development of richer applications, by crossing data in an already well established way (web services) with different IoT systems and with overall already existing Web applications and services. As mentioned before, different application scenarios need different requirements, like security, latency and a specific messaging pattern, for instance. With this in mind, it is crucial for IoT systems to incorporate proper bindings in order to satisfy each application scenario. It would then be the developer's choice to chose which protocols to use on its own application.

In this paper, the results of a thorough study of the state-of-the-art is presented, discussing which protocols and standards should be adopted and how we could achieve the so desired IoT system. In order to demonstrate how it could be achieved an IoT system has been developed. The developed system uses only open-source hardware and software which is already compliant to the most complete, and already available, IoT architecture to date (ETSI M2M [3]) and the best suited network and application protocols (6LoWPAN adaptation layer [4], CoAP [5] and MQTT [6]). To demonstrate how the health sector could benefit from an open internet of things system architecture, a real-time heart rate data web monitoring testbed was set. As this data must be transmitted in real-time, a set of proper protocols were used (MQTT and Web Sockets). This subset of the system demonstrates the need for standard IoT/M2M systems to bind its functionalities to other proper protocols. As a means to prove it, an experiment was conducted comparing the efficiency of MQTT versus the built-in standardized subscription mechanism found on ETSI's M2M specification.

2 Related work

This section denotes the on going efforts of major entities to develop a standard architecture for M2M and IoT systems and the necessary components to achieve and support the ideal IoT. Entities like ETSI and OMA have developed a full fledge standard architect for IoT (ETSI M2M [7] and LWM2M [8], respectively), while others are focusing on developing guidelines and designing smart objects. These smart objects are a nice practice, as it is an attempt to standardize the way IoT systems consume and interact with common "things", by means of known semantics. While these smart objects alone will not solve IoT's issues alone, its combination with latest architectural standard and specifically tailored for IoT protocols might do so. The following sub-sections will detail each case.

2.1 ETSI M2M

ETSI M2M TC's mission is to deploy an end-to-end architecture, providing an horizontal solution where services can serve multiple vertical applications, while offering interoperability between them, independently of the underlying network and technology. It standard, ETSI M2M, is composed of three main area domains: M2M Device Domain, M2M Network Domain and M2M Application Domain. Across these domains are three standard Reference Points connecting them, by means of a set of Service Capabilities. Figure 1 depicts the high level architecture. The Service Capabilities are present in each SCL (Service Capabilities Layer)—DSCL, GSCL and NSCL (Device, Gateway and Network, respectively). A RESTful architecture style was adopted, where information is represented by resources. The SCL holds this standardized resource hierarchical tree structure. The information present on each resource can then be exchanged between the different SCL over the Reference Points (dIa, mId and mIa), via standardized procedures. As

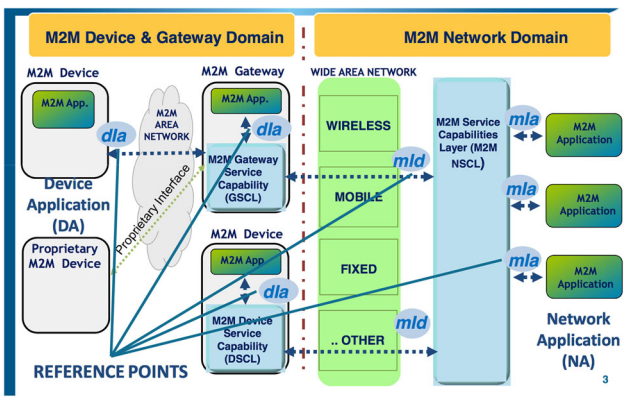

Fig. 1 ETSI's M2M high level system architecture [9]

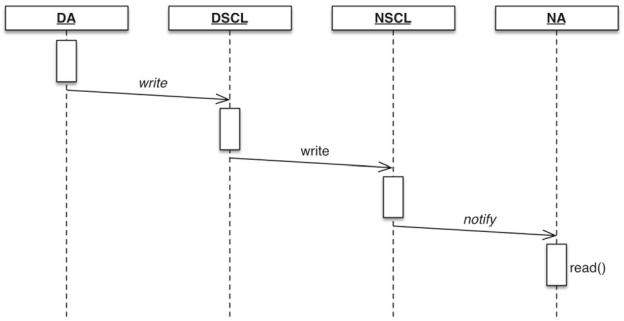

Fig. 2 Simple use of SCL resources to exchange data (adapted from [7])

it follows a RESTful architecture, it is possible to manipulate the information using CRUD methods (Create, Read, Update, Delete) in a Client-Server model.

These SCL allow for management of information related to applications and devices like registration, access rights, security and authentication, data-transfer, subscription/notification and group management [10,11]. All of this is achieved by RESTful operations using HTTP or CoAP, as CoAP binding is also part of the standard, over the standardized resource tree. As a simple example of how this RESTful Architecture works across the SCLs, a diagram is shown Fig. 2 denoting a typical scenario where a sleepy Device (D) sends data to a Network Application (NA). First, the Device writes data to the DSCL through NSCL (UPDATE verb); then, the NSCL notifies the NA of that resource change (notify means either a response of a RETRIEVE or an UPDATE, if a polling or the asynchronous mechanism is used, respectively); at last, the NA reads the new resource change. For the sake of simplicity, the illustrated diagram represents only the events flow, hiding all methods, protocols and responses used in real applications.

The different Reference Points mIa, mId, dIa, and mIm have their own purpose. They all, fundamentally, offer the same mechanisms, but at different domains. dIa offers a generic and extendable mechanism for Device Application (DA) and Gateway Application (GA) to interact with DSCL (Device Service Capability Layer) or GSCL (Gateway Service Capability Layer). This allows the applications to register in the SCL, request read/write permissions on the NSCL, GSCL or DSCL, subscribe and notify to/of specific events and request managing capabilities of groups. mIa functionality is identical to that of dIa, but the interaction is between the Network Application (NA) and the NSCL (Network Service Capability Layer) and it may be used to to request device management actions like firmware update. Likewise, mId follows the same pattern, but over IP connectivity between DSCL/GSCL to NSCL backed by security features. The last Reference Point defined in the Functional Architecture [7], mIm, is a special one used only for inter-domain across different M2M Service Provider; its capabilities differ from mIm when there is no NSCL registration—if this happens, the offered capabilities are some of those offered by mIm like request to read/write information in the NSCL, GSCL or DSCL across two different M2M Service Providers, if properly authorized to do so. The previously mentioned device management capability is possible because of BBF TR-069 [12] and OMA DM [13] compatibility and integration with the standard [3].

2.2 oneM2M

ETSI was not the only SDO (Standards Development Organization) working on a M2M architecture—others had their own solution as well, which would lead to interoperability problems and a slow development of the global M2M market. To avoid any duplication of work between SDOs involved in the same domain, and to agree on a truly global standard M2M architecture, oneM2M was born. In January 2012, seven SDOs—ARIB, ATIS, CCSA, ETSI, TIA, TTA, and TTC -, have agreed on a Global Initiative for M2M Standardization [14]. As they have agreed on an open collaboration with other interested organizations and parties, many others have joined in to contribute in different levels of functionality—OMA, responsible for LWM2M, IPSO Alliance and Continua Health Alliance, as Partners Type 2, are some examples. Six months later, in July 2012, the oneM2M partnership project was established having ETSI TC M2M work being transferred over to form its basis. Fundamentally, oneM2M Partnership Project aims to establish a cooperation "in the production of globally applicable, access-independent M2M Service Layer specifications, including Technical Specifications and Technical Reports related to M2M Solutions" [15].

OneM2M 's ambition to release specific Technical Specifications on different transport layer protocol bindings (like CoAP, HTTP, MQTT and, possibly, XMPP) combined with the inclusion of device management, abstraction layers and semantics view make it a very compelling uniform Standard. The Candidate Release [16] was available for public commenting on August 2014 which already specifies the functional architecture, CoAP and HTTP protocol bindings [17,18] and mappings to device-management related protocols like OMA DM, OMA LWM2M and BBF TR-069 [19,20].

Its functional architecture is depicted in Fig. 3 and one can easily spot its resemblance to ETSI M2M. Like ETSI M2M, the resources described in the architecture can all be interacted via a RESTful API and its primitives, the service layer messages sent over Mca and Mcc reference points, are then mapped to the appropriate transport layer protocols like CoAP, HTTP and MQTT [22]. Bellow these transport layer protocol bindings is the underlying network to which

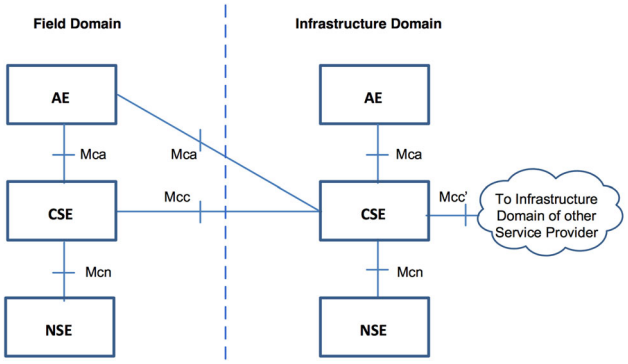

Fig. 3 oneM2M functional architecture [21]

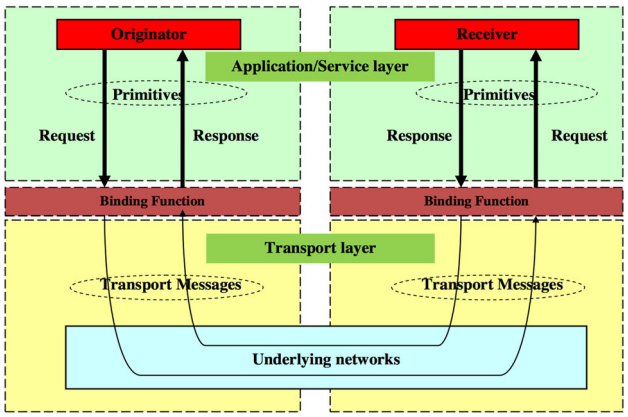

Fig. 4 oneM2M—primitives [23]

oneM2M will also define a set of bindings to. This last feature will be possible via NSE (Network Services Entity) which provides services of the underlying network to the CSEs (Common Services Entity), like device management, location and device triggering. The services provided by CSE are called CSF (Common Services Functions) and these are the key services functions with which AE (Application Entity) and CSEs interact with via the appropriate reference points. The interaction between AE and CSE is made by primitives, as illustrated in Fig. 4. Each CRUD (Create, Update, Retrieve and Delete) operation is mapped to one or more primitives which is then further mapped to transport layer protocols like HTTP, CoAP or MQTT.

2.3 LWM2M

This new standard (Technical Specification Candidate Version 1.0, released in December 2013 [8]) is a fresh new approach to Device Management and was designed having IoT in mind. It is different from OMA DM standard, which is mainly used in Cellular devices, in a sense that it is suitable for any kind of device as long as it uses IP (Cellular, WiFi, 6LowPAN, as examples) and was designed for constrained devices and environments. It is an unique solution that enables both Management and Application data. In essence, OMA Lightweight M2M (LWM2M) is built upon CoAP and DTLS protocol with bindings to UDP and SMS (for Cellular Device management) and offers an extensible Object and Resource model for application semantics which can be published to OMA public registry. This use of known application semantics provides interoperability and abstraction between different IoT/M2M systems.

The Device Management functionalities are as follows:

- *Bootstrapping* automatically connect the device to the right server using key management.
- *Device Configuration* change parameters of the device and network settings.
- *Firmware Update* Over-the-Air (OTA) software updates (to overcome latest security problems, apply patches, enhancements, etc.)
- *Fault Management* automatic error reporting from the device and ability to query it. For debugging purposes such as network unreachability and misconfigured applications and/or services.

And, as it is also a solution for Application's Data, it also allows for:

- *Configuration & Control* in-app configuration of settings (control commands to define new Application's parameters)
- *Reporting* Notification mechanism to alert for new sensor values, alarms and events.

Its architecture is simpler than ETSI's and oneM2M's. Figure 5 illustrates its components and relationships between them.

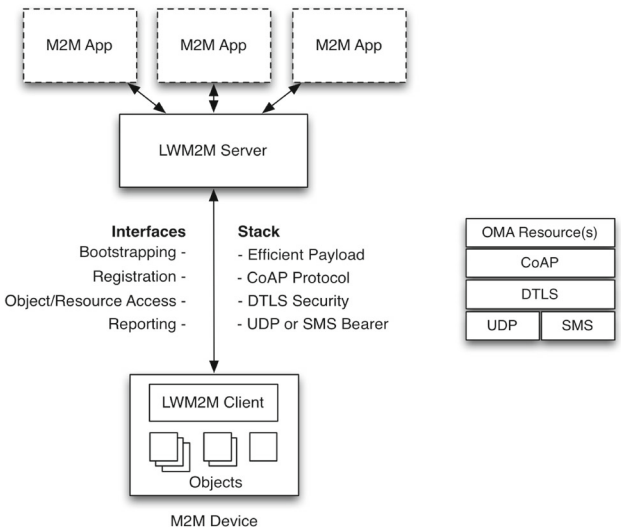

Fig. 5 OMA lightweight M2M architecture [24]

With LWM2M's architecture, there is a small LWM2M Client library in a constrained device which uses the standardized interfaces to manage the built-in Objects created by the Device Manufacture and/or the Application Developer. Managing these known objects between the Client and the multiple M2M Applications, as long as they have authorization to do so, is, therefore, very straightforward, allowing for greater interoperability and abstraction when making use of standard Objects (detailed further). The LWM2M server works as an intermediate between the M2M Applications and the Devices itself—it can command the Device to Read, Write, Execute, Create or Delete Objects and its resources (also detailed further).All of these commands are sent to the Device via four interfaces:

- *Bootstrapping* it is possible to do pre-provisioning out of the SIM card or flash memory, or initiate the configuration mechanism between Client and Server by making use of initial shared keys, server configuration and ACL (Access Control List), for example.
- *Registration* the LWM2M Client can alert the LWM2M Server of its existence and capabilities, by means of Resource Directory (another IETF standard).
- *Management & Service Enablement* used when the Server wants to send operations and commands to the Client. Allows for Device Management and Service Enablement over the previously announced, by the Client, Objects and Resources. The Client responds via this interface as well.
- *Information Reporting* allows for periodically report of resource information by the LWM2M Client in case of triggered events and/or alarms. The notification mechanism is possible due to CoAP's observation feature.

The stack is very small as well, as it uses simple protocols and technologies. CoAP, DTLS and UDP are all lightweight enough, efficient and secure to be used in constrained environments and SMS binding is optional but a must for Cellular use. DTLS is used between LWM2M Server and LWM2M Client to secure all the information exchanged between these two endpoints.

2.4 iPSO Alliance

The IPSO Alliance main objective is to educate, document and support the use of IP for the IoT, while defining the "appropriate protocols, architecture and data definitions for Smart Objects so that engineers and product builders will have access to the necessary tools for "how to build the IoT RIGHT" [25]. It is a much needed Alliance for IoT, giving their efforts on the use of semantics for the whole IoT. These semantics help to represent, in a standard way, common resources like temperature, light control and power control, as examples.

IPSO has published a guideline [26], which should not be recognized as a standard, on how to define Smart Objects using a RESTful design for common M2M Applications such as Home Automation and Building Automation. This defines how a Smart Object can represent its available resources and how to interact with other Smart Objects and back-end services, using the defined set of REST interfaces. The goal here is to show vendors simple guidelines on using IP and Web-based technology to develop and rapidly test interoperability among devices and services. Following these guidelines assures compatibility with the previously mentioned OMA LWM2M. As stated before, third party SDOs can define and register Objects on the OMNA—and that is what IPSO has done. OMNA LWM2M Object & Resource Directory [24] lists the resources already defined by the IPSO Alliance.

2.5 Conclusion

Each standard has its own particularities, and, as it happens with transport layer protocols, each serves a purpose. One standard does not fit all, but a combination might be a good solution. As IoT-A is not really a functional architecture, but an architecting framework, not much can be said at this point; however, this might change in the future as IoT Forum will continue to work its defined ARM so that a a set of re-usable ARM-profiles can be built, to be immediately adopted by IoT Architects. In a much more advanced phase, is ETSI's M2M architecture. It is flexible enough to be widely adopted by either IoT/M2M architects and Application/Services Developers. Its REST API combined with the decoupling of different Services Layer allow for rich applications to be build, while inter-exchanging information and application data with each other, extending their functionality. However, it was not really designed to be supported in small and constrained devices. For a device to be fully compliant with ETSI M2M, its application (DA—Device Application) must support the correct HTTP or CoAP (if built-in bindings or Interworking Proxies are supported) methods to specific URIs with complex XML data.

For real constrained devices, the OMA DM included in ETSI M2M functional architecture might be too complex for such devices. The Remote Entity Management (REM) Service Capability must be supported at both Device/Gateway level and Network level. As the OMA DM standard was initially conceived for not so constrained devices and environments, its natively inclusion in ETSI M2M might be obsolete for real IoT systems. ETSI has laid the foundations for a standard M2M architectural framework; however, as IoT differs from M2M, its standard must support other suitable for IoT protocols.

Contrary to OMA DM, OMA Lightweight M2M's (LWM2M) architecture seems to be more suitable for real IoT systems as some key features are specifically tailored for IoT. It was designed having real constrained devices in mind with the much needed Device Management functions, while adopting open IETF standards designed for typical IoT scenarios (CoAP, DTLS, Resource Directory, UDP and SMS bindings). Although its architecture is not as complex and rich as ETSI M2M, it offers some advantages over the latter. Its interfaces offer much needed functionalities to IoT systems, like: bootstrapping (which allows for a device to automatically connect securely to the correct server), on-the-fly application and devices re-configuration (proper adaptation of the device to the network), security (over UDP and, very important, firmware update to patch security issues). These combined with application data (send and retrieve data using CoAP and UDP/SMS) makes it a very compelling option for an IoT system architect. Also very important is OMA's Naming Authority (OMNA) which allow for simple known semantics to be registered through its extensible model object—like the ones defined by the IPSO Alliance. As it is also very light, most constrained devices need only a small library to support the whole protocol—device manager and application data.

Finally, OneM2M seems to tackle both ETSI M2M's and LWM2M's cons by combining the two. It is a more modern standard that takes ETSI M2M as its basis while offering binding to most commonly used IoT protocols and tools like MQTT and Web Socket (still a work in progress, as evidences found in [22] and [27] suggest so). Its LWM2M integration makes OneM2M the most complete standard ready to target both M2M and IoT solutions supporting all functionalities and services needed by both while, at the same time, providing a concrete way for Application and Services Developers to interact with. This is what is needed in IoT to make way for richer, smarter and interoperable Applications across every domain.

3 Proposed IoT systems

In order to achieve a more generic design that could potentially improve interoperability between other IoT Systems, a proposal is here presented. The goal is to showcase how IoT/M2M Systems should be deployed from now on, as there are new standards and frameworks that will ease the interconnection and interoperability across those systems. This is possible by not simply adopting one architectural standard, but a collection of different ones, which, together, offer the ultimate framework every IoT/M2M designer needs. It was designed with abstraction in mind, hiding much of what is represented with current IoT protocols and standards. Its foundation is based on middleware abstraction layers, ETSI M2M Architecture, LWM2M and iPSO Alliance.

IoT should be open and all things should be able to communicate with each other, even if indirectly—one device, or "thing", should be able to alert other "thing", which is part of a different IoT System, that something occurred. With this, many autonomous and intelligent systems could be born to provide an even better quality of life, since there would be smarter and broader coordination of data and their corresponding action triggers. An example would be the IoT System deployed on an elderly disabled person's smart house: if the system detected a major fall, it could immediately alert that person's family and even trigger an alert on a nursing facility's IoT System, so that they could take immediate action by sending appropriate help to the correct location. These different IoT Systems would be independent and would not be designed with this interactivity in mind—in the future, already deployed IoT Systems will not communicate with other IoT Systems, because they were not designed to do so nor have the needed capabilities. This is why IoT needs a standard System Architecture (not one standard), so that new systems can, in the future, talk with others, creating more intelligent, useful and efficient applications.

With this in mind, Fig. 14 illustrates the layers, at different levels, needed for all of this, and how different systems could interact with each other. The added flexibility comes from the use of ETSI's M2M standard interfaces (mIa, dIa and mId) and Service Capabilities Layers at different domains: device's, gateway's and network's. With ETSI M2M specification as the core of the proposed architecture, different IoT Systems would be ready to interact with each other, even if they do not use the same protocol at the device domain. This way, a system using 802.15.4 to transmit MQTT-SN messages, would also be able to send data to another Gateway from another system, independently of its protocols in use. To put in other words, an MQTT node would talk to a different CoAP node in a different system, while also having both a different radio link (802.15.4 and Bluetooth, for example).

With a proper M2M/IoT architecture, a specific device, which was initially designed to work in a silo manner (Fig. 6), could then interact with a different Network Application (NA) and/or Gateway, providing both a different service to the user and/or an uninterrupted service, if the mobile device connected to a different Gateway, at a different location. This is something ETSI TC M2M has also realized, when analyzing TS 102 689 [28], which refers to M2M Service Requirements. This Technical Specification describes two particular requirements that are much needed for IoT Systems. Although not mandatory for all systems, they were discussed because of the work reported in previous TR (Technical Requirements), where different use cases for different areas were presented. To satisfy much of those use cases, the two particular requirements were: Trusted Applications and Mobility. Trusted Applications means that the M2M Core could handle requests for trusted M2M Applications by pro-

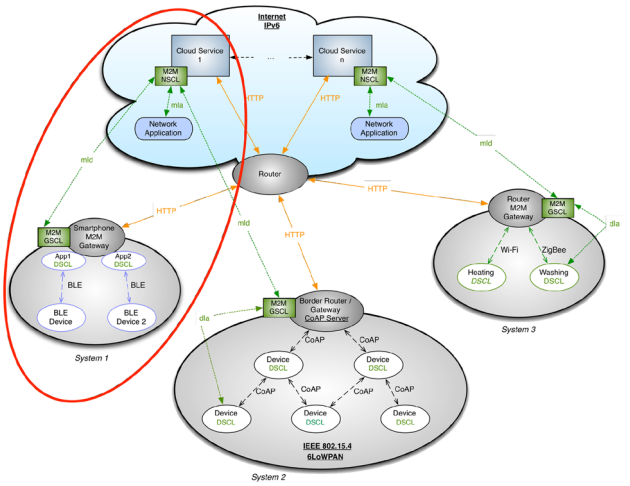

Fig. 6 Current M2M/IoT silos

viding a streamlined authentication procedure—the M2M System would then support Trusted Applications, as they were pre-validated by the M2M Core. The Mobility requirement simply means that if the underlying network supports such mechanism and roaming, the M2M System should be able to make use of such mechanisms. Using Fig. 14 as a reference, it is easier to showcase these two functionalities: Trusted Applications (Fig. 8) and Mobility (Fig. 7). Combining the two of them with LWM2M's pre-provisioning and auto-configuration of devices, would result in a truly ubiquitous IoT System where connectivity, reachability and intelligent coordination would always be possible, which could then make way for smarter Network Applications and new overall services at a personal (end-user) and industry level. With LWM2M's bootstrapping capabilities, any kind of device would be assured of continuous communications if moved from its original domain area or IoT System. In other words, future Bluetooth mobile devices with IP connectivity (Bluetooth Smart [29,30]) would be able to auto-connect to other Gateways of different IoT Systems, and, thus, ensure continuous Internet connectivity. The seamless integration with current Internet services would be possible trough a well documented RESTful API, support for known semantics and various protocol bindings. For instance, having Fig. 14 as a reference, Network Applications from System 1 could discover devices present at System 2, through ETSI M2M discovery mechanism (or even CoAP's, if supported by the node itself), consume the data with a protocol of its choice (some protocols are more suited to specific application/use cases then others) then properly interact with it by making use of known semantics (like IPSO's).

Having this in mind, it is easy to conclude that one M2M/IoT standardized architecture like ETSI M2M might not be enough for all solutions. A combination of different ones with proper bindings to different protocols, use of known semantics and a simple, yet well documented, REST-

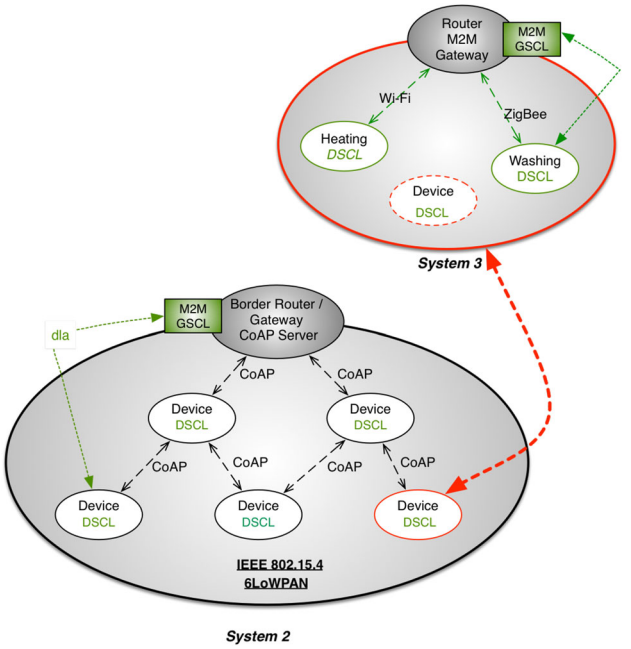

Fig. 7 IoT system architecture proposal—mobility

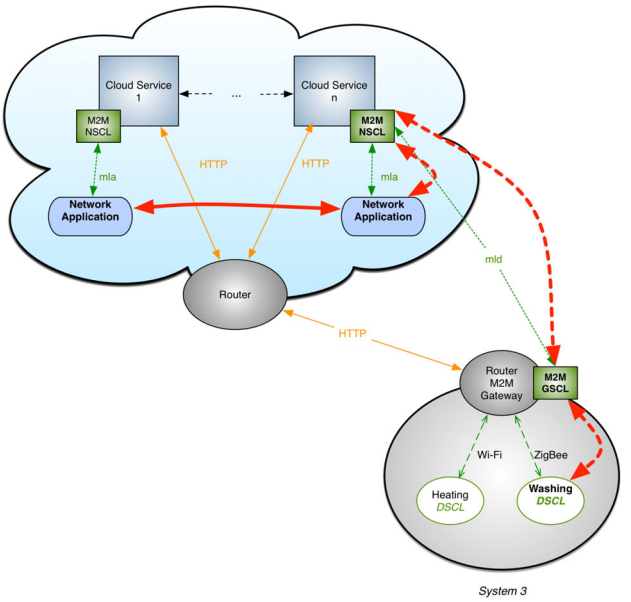

Fig. 8 IoT system architecture proposal—trusted applications

ful API with which one can easily interact with and integrate into the exiting Web, is what is currently needed to achieve the so called internet of things.

Next, a use case is presented and detailed, where an IoT System was developed. It follows the architectural guidelines mentioned before, using as much both free and open source code, tools and frameworks as possible.

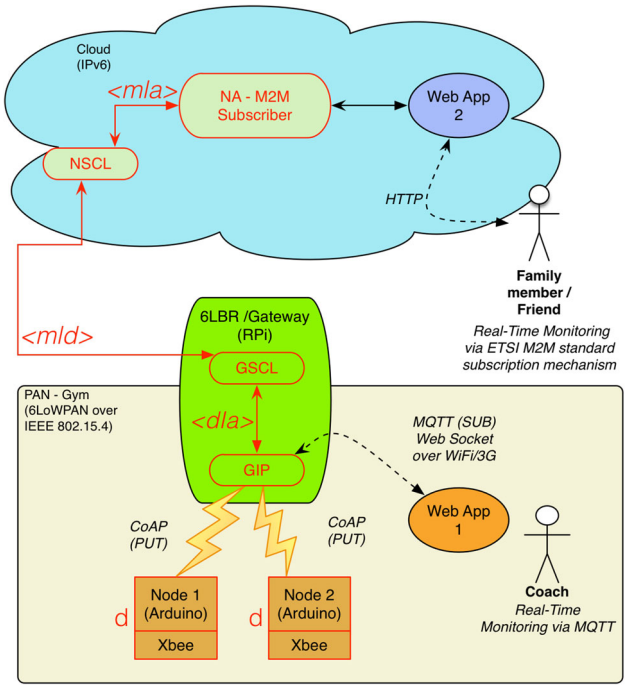

Fig. 9 System's high-level architecture

4 Use case—a real-time web monitoring system

4.1 Overview

The implemented system made use of as much open-source hardware and software as possible, while being low cost at the same time. For this matter, one Arduino Mega [31] and a Raspberry Pi [32] were used for the WPAN and to simulate the cloud service. The overall high-level architecture is depicted in Fig. 9. ETSI M2M architecture was adopted, as it is a finalized specification [3] and because there was already available a developed Java tool compliant to the standard, called OM2M [33]. With this setup, it was possible to develop a full-fledge ETSI M2M-compliant system, with support for legacy and non-compliant M2M devices (Arduino), by extending the OM2M tool, while at the same time supporting real real-time communications.

The Arduino platform was used as a 6LoWPAN node, by extending its basic functionalities with the so-called shields. Combining a Wireless Proto Shield [34] with a Xbee Series 1 module [35]. With this hardware, the Arduino platform was transformed into a 802.15.4 node. As it needed to have IPv6 capabilities, a dedicated 6LoWPAN library was installed [36] with limited CoAP capabilities (simple non-confirmable PUT messages). Similarly, the Raspberry Pi—the WPAN's Border Router and simulated cloud service -, had its basic functionalities expanded by means of a dedicated 802.15.4 interface (Nooliberry, from Noolitic [37]) which was proven to work with the 6LoWPAN software module called 6lbr [38]. These two different modules were the only hardware used for the whole IoT system. With the Raspberry Pi's ability to run as a 6LoWPAN Border Router, the Arduino was able to perform SAA (Stateless Address Auto-configuration) and thus obtaining its own unique global IPv6.

This architecture allows for, fundamentally, two different things: first, it forms the basic layer to extend the whole system to other technologies, protocols, and devices; and, secondly, it supports real-time web monitoring. The first is possible by extending the OM2M tool by developing different interworking plugins (like Zigbee technology and Phidgets devices which are already built-in [33] and by defining special protocol bindings which works best for each scenario. For this particular solution, an interworking proxy unit and a MQTT protocol binding were developed. These two were implemented under the GIP module (Gateway Interworking Proxy) allowing both the adaptation of the non-compliant Arduino node to interact with the GSCL (Gateway Service Capability Layer), and the protocol binding from CoAP to MQTT and Web Socket, for the real-time monitoring requirement. This is detailed further.

4.2 Raspberry Pi

A Raspberry Pi was used to work both as a 6LoWPAN Border Router and as a Gateway of the M2M device domain area. It was able to fulfill all the necessary needs, while being a small and low cost linux computer. With it, it was possible to:

1. Establish an IPv6 tunnel connection, via Gogo6 [39] tunnel broker and the gogoc [40] software model. An account was created at Gogo6, so the tunnel can assign a /56 prefix and a static address. The prefix is transmitted over the RA (Router Advertisement), as seen next.
2. Form and maintain the 6LoWPAN, via the 6lbr [38] software module and radvd [41]. radvd is responsible to send the RA messages via 802.15.4 to the Arduino nodes, so they can proceed with SAA (Stateless Address Autoconfiguration)
3. Initialize the three ETSI M2M modules: GIP, GSCL and NSCL. These modules are detailed further at Sect. 4.4.
4. Initialize the web server, which serves the real-time web monitoring page to the clients.

To summarize what software components are part of the Rasberry Pi, Fig. 10 showcase their stacking and basic functionalities.

4.3 Arduino

The Arduino device, as a typical 6LoWPAN node, has one function only: to sense and send data (CoAP PUT) back to the gateway. However, before looping into that state, it must configure itself to work as a 6LoWPAN node. To

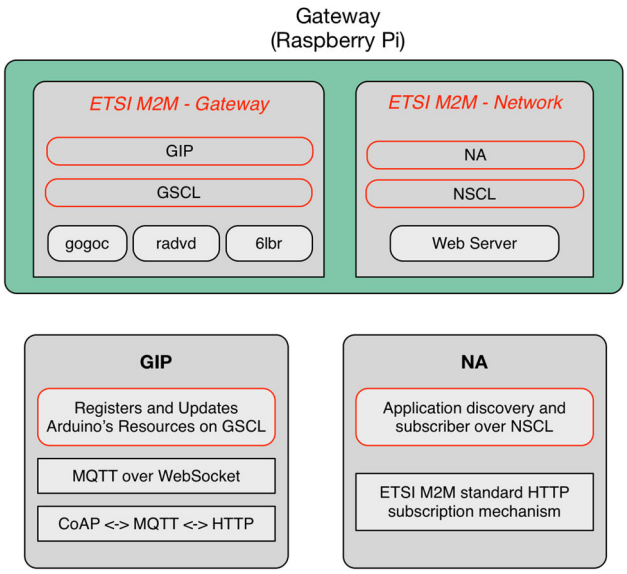

Fig. 10 Raspberry Pi's software modules overview

achieve such thing, the μIPv6Stack library [36] and part of its counterpart, ρIPv6Stack, [42] were used. ρIPv6Stack differs from μIPv6Stack in that it possess a simplified version of RPL and, as the authors say but which could not be confirmed, reduced CoAP capabilities. With the first one, μIPv6Stack, the authors claimed that there was CoAP support, but there was not a single file under its libraries' folder that indicated so. Because of that, the CoAP Engine found on the ρIPv6Stack library was adopted and modified to work under ρIPv6Stack. However, this called CoAP Engine library serves only one purpose: to send CoAP Non-Confirmable PUT messages. As this solution does not require further functionality besides this sensing and reporting, the library was enough. Few modifications to the source code was needed. As the called CoAP Engine was designed to work in combination with ρIPv6Stack, that link had to be broken. This library was then re-configured to work with the μIPv6Stack library.

Once the device boots, it starts the Xbee module, IP stack and UDP. If everything is ok, it starts broadcasting RS (Router Solicitation). Once it is has a properly configured global unique IPv6 (detailed further, under Chapter 6) it will report this address in its first registration. After the registration, it loops and sends periodically the sensed data according to its configuration. All different CoAP messages (the unique registration one and the forthcoming periodically sensed data) is reported solely to the Raspberry Pi's GIP module, which is detailed further.

4.4 ETSI M2M compliance

The main software components of this system's architecture are the GSCL, NSCL, GIP and NA. These nomenclatures follow ETSI's M2M specification as they are compliant to the standard. By making use of such GSCL and NSCL entities, via OM2M, there was built-in support to interact with the different SCLs which then allowed the creation of applications. The applications created were then the NA (Network Application) and the GIP, as mentioned before. Adopting this standard drastically strengthens the solution's interoperability and ability to interwork with other solutions and different technologies. This horizontal layer of services is independent of the underlying network and with its RESTful API, it is easy to parse data and interact with other solutions. Discovery, reachability, addressing, generic communication and application enablement are all features part of the SCL. With the RESTful API, applications and/or services can, upon successful authentication, register new applications, retrieve data and manage access rights, as examples. The NA and the GIP are prime examples of what can be built upon the standard.

GIP

Because the Arduino is not compliant with ETSI M2M, a special Interworking Proxy was developed, at the gateway. One of its goals is to translate specific CoAP messages sent by the node to a special set of HTTP POST messages with a special body and to a special URI. For instance, a simple registration of a device/application is composed of five different HTTP POST messages; this means that, for the Arduino application to be created at the GSCL, there is a need to sequentially issue five different HTTP POST messages to different URI's and with different bodies, after decoding one single CoAP message. This CoAP message is sent by the Arduino only once, after successful initialization and configuration of the IP and UDP stack. Once registered, the GIP translates the upcoming periodic CoAP messages reporting the new sensed data.

As there was a need to deliver the sensed data in real-time and to render this data in a browser, the same GIP was adjusted to support just that. It was designed this way because the Arduino was already sending data to this location and because there was already built-in support to bridge from and to CoAP, HTTP and MQTT (through ponte [43]). Because the whole GIP was developed in Node.js, support for fast and scalable real-time web applications was guaranteed. With all these abilities, all that was needed was to use the right modules and combine them together. Besides the ponte module, its dependent mosca [44] (MQTT Broker) and mows [45] together formed the combination for MQTT over WebSockets support right bellow the MQTT broker.

NA

The NA simulates a typical application that could be built over ETSI M2M standard (NSCL and GSCL, specifically).

Once the sensing node (Arduino, in this case) is reporting data to the GSCL, higher-level applications at the network-domain can properly interact with the appropriate NSCL to retrieve this data. This is what the NA does. It first interacts with the GSCL, via the NSCL, to discover what kind of applications are registered (Arduino nodes) and then creates a subscription to that particular resource, using ETSI M2M standard procedure. The supporting method is an asynchronous subscription, meaning that the NA must be a server application listening on an ip and port for incoming data. Once the Arduino reports new data, the GSCL will automatically send an HTTP POST with a XML encoded body with the new reported value to the listening NA.

ETSI M2M's subscription mechanism might be good enough for some applications, but not for the needed real-time web monitoring solution. It is not lightweight enough and would not scale as well as a proper PUB/SUB type of messaging pattern protocol would, like MQTT. Furthermore, OM2M implements, by default, non-persistent connectivity, resulting in aggravated overload. An experiment was set to properly analyze the differences introduced when using both mechanisms (ETSI M2M's and MQTT's). This is detailed at Sect. 4.6.

4.5 Node.js & real-time web monitoring support

Having the system in compliance to ETSI M2M standard is a good starting point, however, that standard, as it is, lacks some functionalities required for this particular solution. Specifically, there is no real-time monitoring support. ETSI M2M does, in fact, support synchronous (long polling) and asynchronous subscriptions to new events, but those mechanisms rely solely on HTTP POST messages for every new update. The synchronous type of subscription is not a real push notification mechanism, as the client needs to keep polling the server for new data, which is not optimal for real-time monitoring. On the other hand, the asynchronous mechanism does support instant notifications to a specific listening application. This last mechanism was adopted for the developed NA, as detailed before. As there are specific protocols designed that work best for this type of messaging pattern (publish/subscribe, or PUB/SUB), like MQTT, this protocol was used to enable real-time web monitoring, when coupled with Web Sockets technology.

To support this real-time web monitoring, Node.js modules were adopted. Node.js is a "platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications." [46] Its event-driven non-blocking I/O model make it ideal for this kind of IO-bound applications, as stated by the authors: "Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices." [46] In a simplified manner,

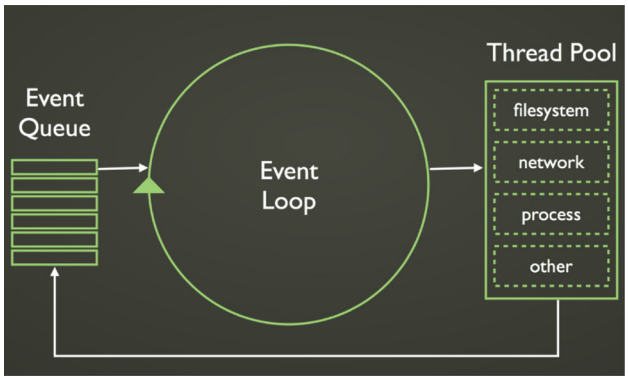

Fig. 11 Node.js's event loop [49]

Node.js manages to do just that by taking away the normal waiting time for I/O tasks to complete and replacing it by small CPU activity, by means of an event loop and a thread pool. The Node.js event loop is a single-thread application, however, the I/O is delegated to a thread pool which is maintained by the OS. The event loop keeps grabbing code from the event queue and execute it, while there is no callbacks from previously stacked IO tasks. Once the previously I/O task is completed, the callback will be picked up by the event loop and process it right away. This mechanism is represented in Fig. 11. It is a little bit different from what is typically found in other language programming models, but it serves its purpose (fast and scalable IO-bound applications) very well, as it abstracts the low level complex event loop and all the necessary OS's callbacks in a very simplified way, as a Javascript programming language. It is possible to do so, due to its bindings to the OS and the fundamental dependencies on libraries supporting thread pool (libeio [47]) and event loop (libev [48]). These low level bindings and dependencies are written in C++ and C code while the upper node standard library allows for developers to leverage this non-blocking I/O programming model by means of Javascript's callbacks and anonymous functions. For this matter, the whole GIP and its built-in real-time web monitoring support was developed as single-thread Node.js application. By making use of the appropriate modules, it was easy to provide support for MQTT and Web Socket communication, under the GIP.

Extending ETSI M2M to support bindings for this type of communication protocols greatly enhances its overall ability to support different IoT applications, as each of them have its own pre-requisites. The OneM2M partnership is already working on this, as they have too realized the importance of proper built-in protocol bindings [22]. This mechanism could also be incorporated onto the OM2M platform, as it supports the development of external plugins; however, due to the real-time nature of the application at hand, an external application developed entirely on Node.js supporting the Interworking Proxy (needed for the Arduino and its CoAP

to work with ETSI M2M, as explained before) and MQTT plus WebSocket support was thought to be the best solution. These separate applications (the NA making use of ETSI M2M standardized subscription method and the Web application making use of MQTT and Web Sockets) allowed for setting up an experiment where different metrics were measured, comparing both latency the traffic load differences. The results can be found in Sect. 4.6 and they fundament the need for IoT/M2M standards to built-in different protocol bindings.

Real-time web monitoring

Currently, the ETSI M2M standard specifies that, in order to subscribe for new data, an IP and listening Port must be passed on to the GSCL which will, in turn, send an HTTP POST to the specified location as soon as the application reported new data (like the previously mentioned NA). At first sight, this does not appear to be a practical solution, since there are proper protocols for this kind of messaging pattern. One should expect that, over time, once there are thousands of Device Applications (DA) and subscriptions, it would be difficult to handle and scale such a large number of notifications. Furthermore, in the future, there will be a need to monitor sensors and/or devices in real-time, so a better mechanism should be adopted. This asynchronous HTTP reporting mechanism is not ideal for this solution and it would not scale as well as a proper Pub/Sub protocol would. For this matter, a combination of MQTT-binding and Web Sockets was developed under the GIP. Besides taking away the complexity of the HTTP notification mechanism, the Web Application communicates directly with the GIP, taking away the processing time between this one and the GSCL.

On the server end, the GIP is leveraging the ponte module (detailed before) and its ability to seamlessly bridge from CoAP to MQTT. As this module is embedded in the GIP, it was also configured to make use of its capabilities to work as a MQTT broker using the dependent mosca [44] and its built-in Web Socket mechanism through mows [45]. All in all, ponte is a powerful tool that wraps a plethora of other modules to provide a fast, event-based mechanism to bridge HTTP, CoAP and MQTT services and messages. Once properly configured, the GIP was ready to both automatically bridge from CoAP to MQTT as it was possible to connect and subscribe to its MQTT broker via Web Socket. The client end needed only to use the appropriate Javascript library which allows the use of WebSockets to connect to the broker, which was part of the Eclipse's Paho open-source project [50,51].

This Web page was designed to solely demonstrate how real-time monitoring web-based monitoring solutions can be built, using standard and appropriate IoT protocols and technology. By leveraging Node.js capabilities to scale and offer fast real-time I/O bound applications, CoAP, Web Sockets and MQTT, a future-proof solution was deployed using only already available open source libraries and tools. The Web page was designed to work for this particular scenario only. There is no way to choose which broker we want to connect and which topics we wish to subscribe to—all of this is hard-coded for the particular solution. Figure 12 showcases the Web page developed. The graph is updated as soon as new data comes in through the Web Socket (in form of a MQTT Publish message).

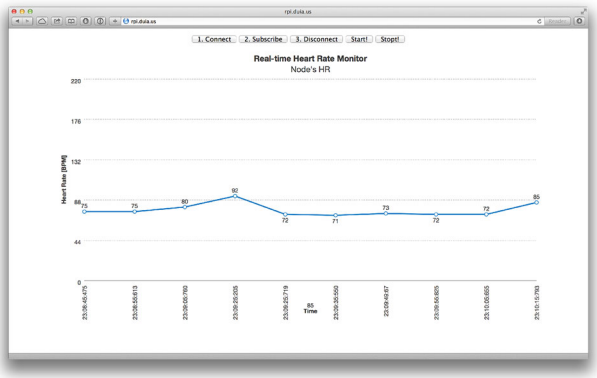

Fig. 12 Real-time monitoring via web sockets

Table 1 Traffic comparison when using ETSI M2M standard subscription method and MQTT

	HTTP (bytes)	MQTT (bytes)
BR → NA	715	15
NA → BR	395	0
Total per message	1110	15
Total session	33300	511
Diff. Factor	65,17×	1×

4.6 MQTT versus ETSI M2M standard subscription mechanism (HTTP)

This section show the practical difference when adopting current ETSI M2M standard subscription mechanism and MQTT's. For this, it was tested both the traffic and latency found in each case. As a normal NSCL would be part of an IoT/M2M Service Provider, located at the network domain, changes in the previously mentioned solution (where both GSCL and NSCL were functioning at the domain/gateway domain) had to be done. To approximate the experiment with a real life properly-deployed IoT solution, both the NSCL and NA were installed on a VPS (Virtual Private Server), located in the USA, and the MQTT client on a Macbook Air, located in a domestic Spanish network. Although the NA was in the same computer as the NSCL, the transmitted data was coming directly from the Border Router's GSCL which was located in a domestic Portuguese network.

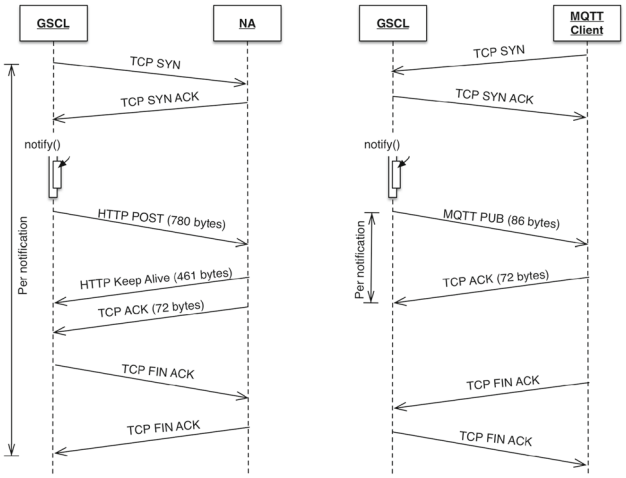

Fig. 13 ETSI M2M standard subscription method versus MQTT

With only one subscription in the whole system (the one used for the test), it was already possible to notice a considerable difference of 788ms average. This was expected as the standard ETSI M2M subscription mechanism depends heavily on new TCP sessions for every new notification (non-persistent connectivity of OM2M's) with a complex XML payload. This experiment, detailed in Fig. 15, lasted for 37 min with the Arduino node reporting new data every 10 s. For best results, the different machines' clocks were synchronized with the same server, using (Network Time Protocol). As one of the two machines used was a Macbook Air, the chosen NTP server for both machines was "time.apple.com". It is important to notice that during this experiment, there was only one active subscription registered at the GSCL. In the future, it is expected for an IoT system to be composed of thousands of devices, applications and subscriptions. The graph shows the delay noticed on each message and the accumulated for the whole experience. In the end, the accumulated delay was of 2 m 52 s and 700 ms. Also important to notice is the 0 m 7 s 647 ms delay occurred at the 66th message

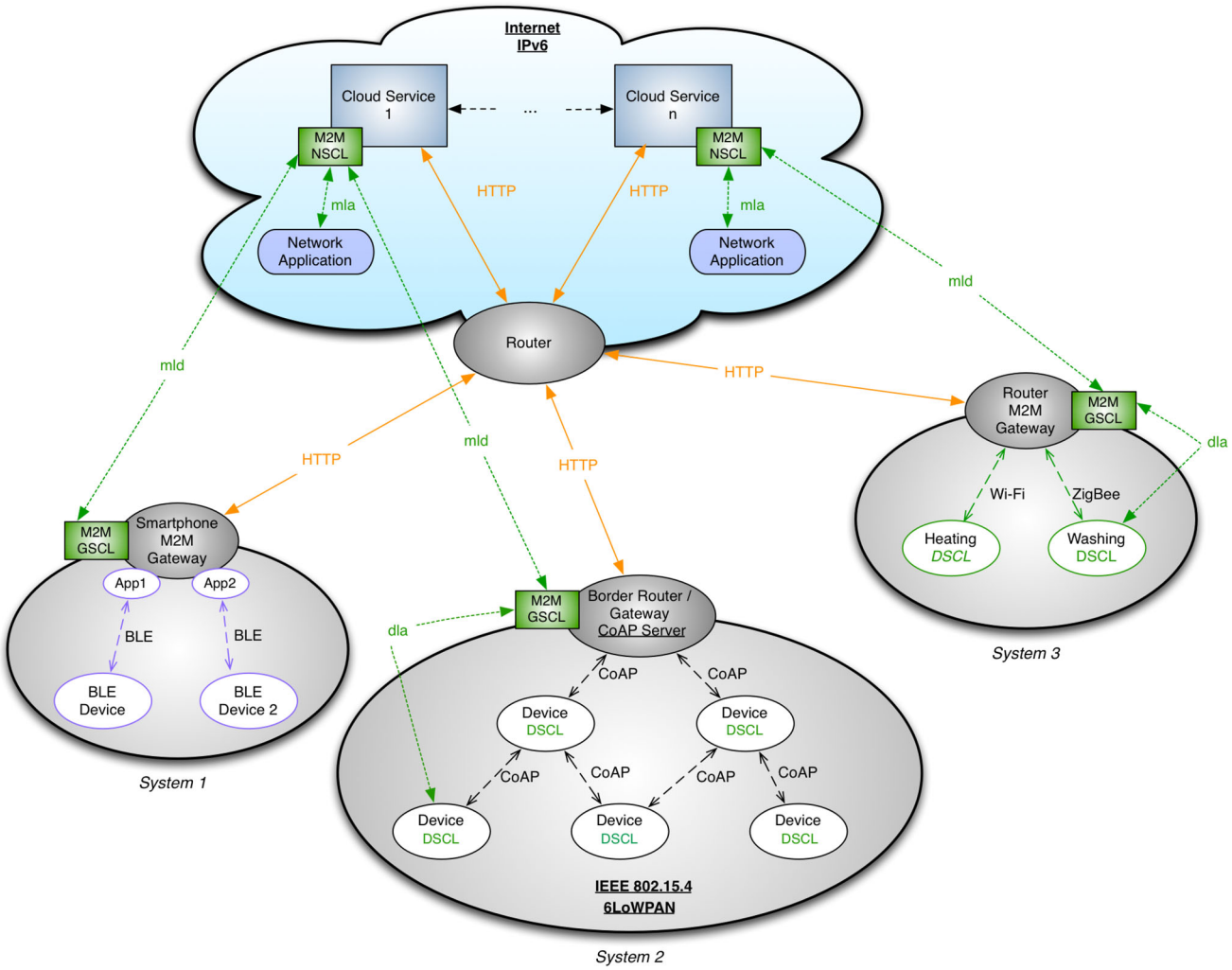

Fig. 14 Internet of things standardized system architecture

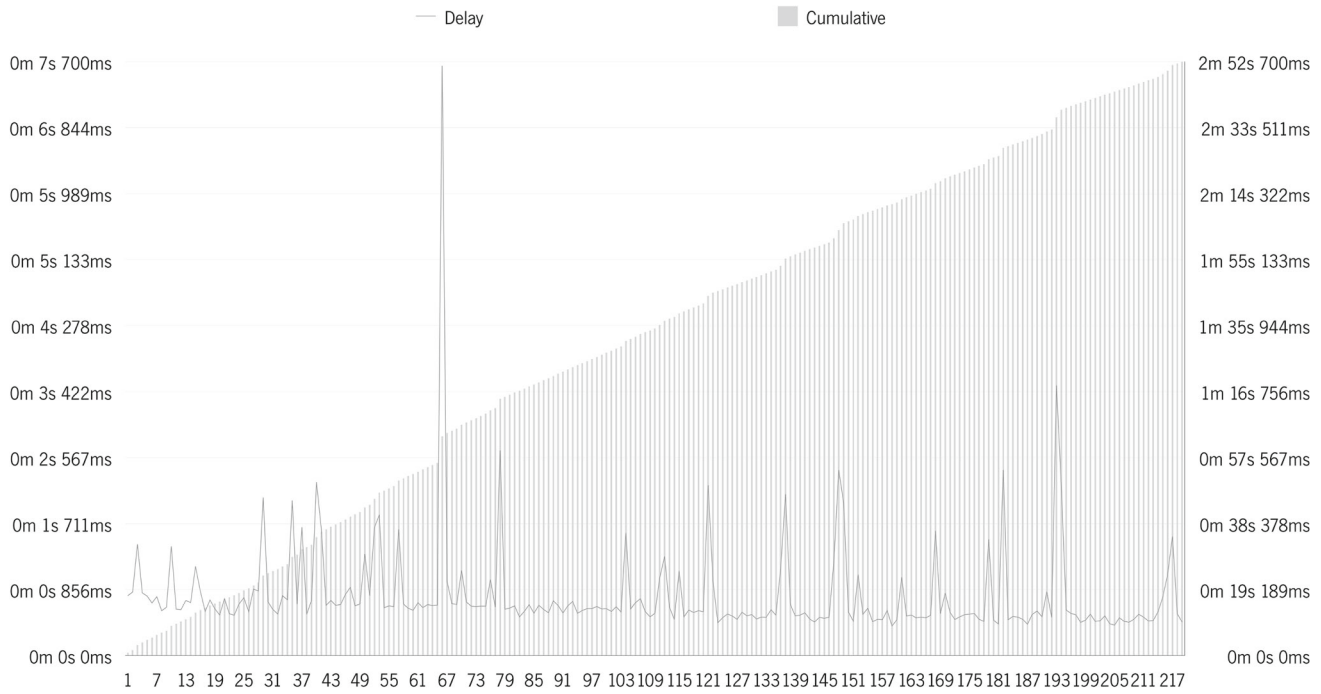

Fig. 15 Delay difference between MQTT and standard ETSI M2M subscription mechanism

and various others in the range of almost 2 s. It clearly demonstrates how inadequate it currently is for real-time solutions.

One could argue that this difference could be debatable given the nature of Node.js event-driven, non-blocking IO model. There is no certainty that it first bridges to MQTT rather than HTTP. However, it is expected for the first to happen because the MQTT client (Web page) is automatically subscribed to the same CoAP's resource, whereas in the case of HTTP bridging, the application needs first to decode the incoming CoAP URI to then forward to an external HTTP service. This is why the experiment lasted for 37 min, with a total of 219 updates. Independently of the result—which bridging takes place first -, the average difference was still 788ms, being the MQTT solution faster. This could be analyzed with further deeper debugging.

The next experiment tested how much data was actually transmitted during the whole session and for each new notification. Making use of Wireshark, it was possible to conclude that there is a big difference between the standardized ETSI M2M solution and that of MQTT's. For starters, the standard subscription initializes a new TCP connection every time it wants to report data—in real-time monitoring, it is expected that new data comes in very frequently and, thus, the session should be open all the time, for efficiency reasons. Also, as there is no way to tweak the QoS, the NA automatically replies with another HTTP POST for every notification. MQTT takes all of this complexity and inefficiency away. First, it keeps a single TCP connection open during the entire subscription and, secondly, it is possible not to wait for ACK messages, by setting the level of QoS to 0 when subscribing to the specific topic. Again, for real-time monitoring, there is no need to acknowledge the reception of every single notification. If, however, one feels the necessity of such, a simple adjustment of the QoS parameter would assure so. Table 1 summarizes the findings, when using Wireshark. As detailed in the table, the difference is quite big: 65 times bigger when compared to MQTT protocol with QoS parameter set to 0. The reason for this is that, as per RFC 2616 [52], for every POST message received, a response code of 200 (OK) or 204 (No Content) should be sent whenever the resource can not be identified by the URI. As the NA was developed merely to showcase how one can interact with ETSI M2M architecture, the 204 reply code is sent automatically by ponte, with a total of 395 bytes. Had this reply been completely ignored, the overall generated traffic would still be roughly 42 times greater than that of MQTT's. Figure 13 illustrates what is actually happening during each notification, when making use of ETSI M2M's standard subscription mechanism and MQTT's, respectively. The NA subscription model could be more efficient if OM2M's mechanism had support for persistent connection. As Wireshark has demonstrated there a new TCP connection is established whenever there is new data to be reported.

This experiment shows how important it is for IoT standard architectures to have this kind of built-in protocol

binding and why MQTT is such a good protocol for IoT. As Cisco estimates that 50 billion devices and objects will be connected to the Internet by 2020 [53] this traffic efficiency is much needed. MQTT offers even more flexibility and scalability capacity, has documented before in Chapter 2. All of these factors are crucial to provide real-time monitoring.

5 Conclusion

This work has focused on the most promising M2M/IoT architectural standards and protocols, demonstrating how they can solve the current IoT issues and, thus, achieving the ultimate envisioned IoT. As it will have a major impact on the health sector, a testbed was set to demonstrate how real IoT systems can currently be deployed using open standards and open-source tools. With this testbed implementation and validation, one can conclude that an IoT architecture should be as generic as possible, meaning that it should be designed to cover the most different use cases typically found in IoT and M2M. The real-time monitoring web-based solution, found in this testbed, is a prime example of how such flexibility is needed. If this solution had not been well adapted to meet the real-time requirement, the IoT architecture adopted (ETSI M2M) would not be able to deliver such a service. This is something of utmost importance and that is something the next generation of IoT standards (oneM2M, specifically) will tackle, by combining even more technologies into the standard (like device management via LWM2M, for example) and protocol bindings like MQTT. Having this standard architecture with different horizontal layers offering fundamental functionalities, while being network agnostic, is a major breakthrough and will allow the IoT to fully grow up to its potential. Interacting with the services via a standardized RESTful API will make way for new Web and mobile based applications to grow and directly interact with the already well established Internet. In the end, this will, ultimately, result in smarter, fully aware IoT applications which combine different data sources and mechanisms to trigger events in other IoT systems/applications, offering the end-user a better quality of life with simplified, smart and autonomous real-life actions.

The system deployed fulfilled the pre-determined requirements using state-of-the-art open source tools compliant with the most complete IoT standard—ETSI M2M. This allowed for a development of an open solution completely future proof with the next iteration of the standard: oneM2M . This proof of concept demonstrated that it is possible to use low cost embedded devices that are not compliant to the standard itself, by developing internal Interworking Proxies which are, in fact, praised by the standard. Besides, it was possible to notice something really important for IoT: a combination of a real-time web monitoring solution and storage of this aggregated data. The former by combining a MQTT binding at the Interworking Proxy and the latter by interacting with the standard using the documented RESTful API—Network Application (NA), specifically, which discovers new applications and subscribes to its updates. The results have shown a difference of 788ms and a traffic increase of, at least, 42 times fold, which fundaments the need to use proper protocols and messaging patterns.

As a final note, one should denote that the internet of things, due to its own broad nature, is applicable at very different areas, each of them having their own prerequisites and singularities. This means that while there is a need to make all of these different systems cross-compatible and interoperable, there is also a demand to define specific mechanisms necessary to solve specific solutions. With this in mind, one can conclude that the most sophisticated standard IoT architecture is the one that presents greater flexibility to adapt to other technologies (and standards and protocols, for that matter), while abstracting this interworking and binding mechanisms. ETSI M2M laid the foundation for this horizontal abstraction layer, but it must be completed with the much needed proper protocol bindings and different various interworking processes. OneM2M is already working on this and it might be the much needed common framework for the IoT. This cross-compatibility with different standards, where each serves its own purpose, is part of the needed capabilities to achieve the desired IoT, as demonstrated over at Sect. 3. The developed system takes all of this into account and showcases how future-proof it is and how it would be ready to interact with other IoT systems, inter-exchanging relevant data. At another level, accompanying the evolution of the IoT, this data would then become relevant knowledge, which would trigger appropriate actions on other different IoT Systems. This is what is possible to achieve with the so called internet of things, when its issues are sorted out and there is a consensus across the whole industry and literature.

6 Future work

There are many different areas where the developed solution could be improved. For instance, there is no guaranteed privacy at the device domain area. This is an issue for all constrained devices and the Arduino is not an exception. Although the model used had 8KB of SRAM, the minimal code size needed for a modified version of the already optimized DTLS library (tinyDTLS [54]) is 9812 bytes [55]. This makes it hard for constrained devices to use both CoAP and DTLS/TLS, and that is something the IETF Working Group, called LWIG (Light-Weight Implementation Guidance [56]), is actually working on. However, this could be temporarily mitigated by implementing relic-toolkit [57] which is fast and its code footprint is small enough to fit in the used Arduino

Mega as per [58] findings—the final Arduino code developed for this solution could still hold 3342 bytes of code which would be enough to hold the needed 2804 bytes, if the NIST K163 algorithm was chosen [58]. Besides this encryption at the PAN level, the OM2M tool should also be able to encrypt the exchanged data. Currently, it supports only a basic "username/password" type of authentication mechanisms encoded to base64. As both GSCL and NSCL work at the network layer (and, thus, not a constrained environment), they already possess the ability to provide security over the Internet, if cryptographic protocols like TLS (Transport Layer Security) were adopted.

Although the chosen prototyping board (Arduino) was capable of doing the necessary minimum for the whole solution to work properly, it has demonstrated not to be fully capable of adopting typical IoT technology. It was possible to configure real IPv6 routing over a 802.15.4 radio link, however, it was not possible to communicate directly with it from any other IPv6 endpoint. Tests were made by simply sending IPv6 UDP packets, but the used library could not recognize the UDP datagram, although it did recognized the incoming IP packet. Further research on the adopted library would have to be done to understand what was actually preventing this. Had this been possible, the GSCL could have been re-configured to support a mechanism called Application Point of Contact (aPoC). This aPoC is an attribute of the registered application in the SCL and once the "aPoCPaths" were configured, it would then be possible to re-target incoming HTTP POST messages aimed at the SCL directly to the Arduino using the CoAP protocol. If such real IPv6 end-to-end bi-directional communication was possible, this GSCL's aPoC feature would allow for real direct orders to reach the nodes (like a start/stop button on a web page to start/stop the reporting mechanism on the node). The CoAP library was also very limited. In essence, it only supported the transmission of non-confirmable PUT messages. However, that feature was indeed the only needed one for the whole IoT solution to work as intended.

In order to achieve the ultimate optimal solution, there would have to be changes at both hardware and software level. As the Arduino has showed limited capabilities to work as a full fledge 6LoWPAN node, new more capable boards would have to be used. As there are many platforms proven to work properly as a 6LoWPAN node, by supporting Contiki OS [59], those could be a good upgrade. Also, by natively supporting Contiki OS, there would also be support to a complete CoAP implementation, as it is part of the OS. After having this basic layer, the whole solution could then be improved by making use of the much praised LWM2M standard. As mentioned before, this standard offers some great functionalities to IoT Systems. As it works with CoAP, it would be possible to also bring this standard to both clients (new boards) and the Raspberry Pi, by porting the recent project called Wakaama (formerly known as liblwm2m) [60]. Also important, these boards would then be much more efficient and secure by supporting sleeping mode and privacy through DTLS.

With all these additions, the solution would be compliant to the next upcoming universal M2M/IoT standard that is oneM2M. Heading towards this standard, which will provide bindings to MQTT and LWM2M, would allow for further development of the ultimate features mentioned in Sect. 3: trusted applications and mobility. At this point, a new framework supporting oneM2M and its needed bindings would be developed, or further improvements to the oM2M would be made. As this tool runs on top of an OSGi Equinox runtime, it would be possible to extend its functionalities by developing new plugins. Alternatively, the new oneM2M framework could be developed completely in Node.js to leverage its great capabilities to deal with I/O-bound applications and systems. Either way, security using TLS could also be developed to secure the whole communications at the network level (Figs. 14, 15).

Acknowledgements This project was funded by Fundo Europeu de Desenvolvimento Regional (FEDER), by Programa Operacional Factores de Competitividade (POFC) - COMPETE and by Fundação para a Ciência e Tecnologia, on the Scope of projects: PEstC/EEI/UI0319/2015 and PEstC/EEI/UI0027/2015. This paper is a result of the project "SmartEGOV: Harnessing EGOV for Smart Governance (Foundations, methods, Tools) / NORTE-01-0145-FEDER-000037", supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (EFDR).

References

1. Gartner: Gartner Press Release, STAMFORD, Conn. http://www.gartner.com/newsroom/id/2636073 (2013)
2. Bradley, J., Barbier, J., Handler, D.: Embracing the internet of everything to capture your share of $ 14 . 4 Trillion, http://www.cisco.com/web/about/ac79/docs/innov/IoE_Economy.pdf (2013)
3. ETSI TC M2M: ETSI TS 103 093 V2.1.1. http://www.etsi.org/deliver/etsi_ts/103000_103099/103093/02.01.01_60/ts_103093v020101p.pdf (2013b)
4. 6lowpanWG, Internet Engineering Task Force (IETF): IPv6 over Low power WPAN (6LoWPAN). http://datatracker.ietf.org/wg/6lowpan/ (2012)
5. Shelby, Z., Hartke, K., Bormann, C. RFC 7252—the constrained application protocol (CoAP). Tech. rep., http://www.rfc-editor.org/rfc/pdfrfc/rfc7252.txt.pdf (2014)
6. International Business Machines Corporation (IBM), Eurotech: MQTT V3.1 Protocol Specification, pp 1–42, http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf (2010)
7. ETSI: ETSI TS 102 690 - Machine-to-Machine communications (M2M): Functional architecture. RTS/M2M-00002ed211, http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/02.01.01_60/ts_102690v020101p.pdf (2013)

8. Open Mobile Alliance: Lightweight machine to machine technical specification candidate Ver 1.0. http://technical.openmobilealliance.org/Technical/technical-information/release-program/release-program-copyright-notice?rp=154&r_type=technical&fp=Technical%2FRelease_Program%2Fdocs%2FLightweightM2M%2FV1_0-20131210-C%2FOMA-TS-LightweightM2M-V1_0-20131210-C.pdf (2013)
9. Pareglio, B.: Overview of ETSI M2M architecture (October), http://docbox.etsi.org/workshop/2011/201110_m2mworkshop/02_m2m_standard/m2mwg2_architecture_pareglio.pdf (2011)
10. Boswarthick DTOTM: Status of machine to machine standards work in TC M2M and oneM2M. http://www.etsi.org/plugtests/COAP2/Presentations/03_ETSI_M2M_oneM2M.pdf (2012)
11. Lu, G.: Overview of ETSI M2M Release 1 Stage 3—API and resource usage. http://docbox.etsi.org/workshop/2011/201110_m2mworkshop/02_m2m_standard/m2mwg3_api_andresource_usage_lu.pdf (2011)
12. The Broadband Forum: TR-181 Device Data Model for TR-069. http://www.broadband-forum.org/technical/download/TR-181_Issue-2_Amendment-7.pdf (2013)
13. OMA: OMA device management V1.2. http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/dm-v1-2 (2008)
14. ETSI : Major Standards Development Organizations Agree on a Global Initiative for M2M Standardization. http://www.etsi.org/news-events/news/381-news-release-17-january-2012 (2012)
15. Koss, J.: oneM2M—a global initiative for M2M standardization. http://goo.gl/RBcjkR (2012)
16. OneM2M: oneM2M candidate release August 2014. http://www.onem2m.org/technical/candidate-release-august-2014 (2014f)
17. OneM2M: CoAP protocol binding technical specification. http://www.onem2m.org/candidate_release/TS-0008-CoAP_Protocol_Binding-V-2014-08.pdf (2014a)
18. OneM2M: HTTP protocol binding technical specification. http://www.onem2m.org/candidate_release/TS-0009-HTTP_Protocol_Binding_V-2014-08.pdf (2014b)
19. OneM2M: Management enablement (BBF). http://www.onem2m.org/candidate_release/TS-0006-Management_Enablement_(BBF)-V-2014-08.pdf (2014c)
20. OneM2M: Management enablement (OMA). http://www.onem2m.org/candidate_release/TS-0005-Management_Enablement(OMA)-V-2014-08.pdf (2014d)
21. OneM2M: oneM2M functional architecture baseline draft. http://www.onem2m.org/candidate_release/TS-0001-oneM2M-Functional-Architecture-V-2014-08.pdf (2014g)
22. OneM2M: oneM2M—developing MQTT Protocol Binding ftp://ftp.onem2m.org/Meetings/TP/2013meetings/20131209_TP8_Miyazaki/oneM2M-TP-2013-0388-WI_for_MQTT_binding.DOC (2013)
23. OneM2M: RFC 5139—oneM2M service layer protocol core specification. http://www.onem2m.org/candidate_release/TS-0004-CoreProtocol-V-2014-08.pdf (2014h)
24. Open Mobile Alliance: OMNA lightweight M2M (LWM2M) object & resource registry. http://technical.openmobilealliance.org/Technical/technical-information/omna/lightweight-m2m-lwm2m-object-registry (2014)
25. IPSO Alliance: About IPSO—vision and mission. http://www.ipso-alliance.org/about/mission (2014)
26. IPSO Alliance, Shelby, Z.: The IPSO application framework (draft-ipso-app-framework-04), pp. 1–19, http://www.ipso-alliance.org/technical-information/ipso-guidelines (2012)
27. OneM2M: OneM2M—WebSocket based Notification ftp://ftp.onem2m.org/Meetings/PRO/20140407_PRO10.0_Berlin/PRO-2014-0160R01-WebSocket_based_Notification.DOC (2014e)
28. ETSI TC M2M: ETSI TS 102 689 - Machine-to-Machine communications (M2M); M2M service requirements. http://www.etsi.org/deliver/etsi_ts/102600_102699/102689/02.01.01_60/ts_102689v020101p.pdf (2013a)
29. Bluetooth SIG Inc: Updated Bluetooth® 4.1 Extends the Foundation of Bluetooth Technology for the Internet of Things. http://www.bluetooth.com/Pages/Press-Releases-Detail.aspx?ItemID=197 (2013)
30. Siekkinen, M., Hiienkari, M., Nurminen, J.K., Nieminen, J. How low energy is bluetooth low energy? Comparative measurements with ZigBee/802.15.4, pp. 232–237 (2012)
31. Arduino (2014b) Arduino Mega 2560. http://arduino.cc/en/Main/arduinoBoardMega2560 (2014)
32. Raspberry Pi Foundation: Raspberry Pi Model B. http://www.raspberrypi.org/products/model-b/ (2012)
33. Alaya, M.B., Banouar, Y., Monteil, T., Chassot, C., Drira, K.: OM2M: extensible ETSI-compliant M2M service platform with self-configuration capability. Procedia Comput. Sci. **32**, 1079–1086 (2014). doi:10.1016/j.procs.2014.05.536. http://www.sciencedirect.com/science/article/pii/S1877050914007364
34. Arduino (2014a) Arduino—Wireless Proto Shield. http://arduino.cc/en/Main/ArduinoWirelessProtoShield (2014)
35. Digi International Inc: XBee® 802.15.4. http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-series1-module (2014)
36. Télecom Bretagne: Arduino-IPv6Stack. https://github.com/telecombretagne/Arduino-IPv6Stack/ (2012)
37. Noolitic: Noolitic's Nooliberry. https://github.com/Noolitic/Nooliberry/wiki (2013)
38. CETIC: 6lbr—a deployment-ready 6LoWPAN Border Router solution based on Contiki. http://cetic.github.io/6lbr/ (2014)
39. Gogo6: Freenet6 Tunnel Broker. http://www.gogo6.com/freenet6/tunnelbroker (2014a)
40. Gogo6: gogoCLIENT Download Page. http://www.gogo6.com/profiles/profile/show?id=gogoCLIENT (2014b)
41. Litechorg: Linux IPv6 router advertisement daemon (radvd). http://www.litech.org/radvd/ (2014)
42. Télecom Bretagne: Arduino-pIPv6Stack. https://github.com/telecombretagne/Arduino-pIPv6Stack (2013)
43. Collina, M.: ponte—the Internet of Things Bridge for REST developers. https://www.npmjs.org/package/ponte (2014)
44. Collina, M.: mosca—MQTT broker as a module. https://www.npmjs.org/package/mosca (2013a)
45. Collina, M.: Use MQTT from the browser, based on MQTT.js and websocket-stream. https://www.npmjs.org/package/mows (2013b)
46. Joyent, I.: Node.js official website. http://nodejs.org (2014)
47. Lehmann, M.: Libeio's offical web page. http://software.schmorp.de/pkg/libeio.html (2014a)
48. Lehmann, M.: Libev's official web page. http://software.schmorp.de/pkg/libev.html (2014b)
49. Kunkle, J.: Node.js presentation. http://kunkle.org/nodejs-explained-pres/#/event-loop (2014)
50. Eclipse: Paho—open source messaging for M2M. http://www.eclipse.org/paho/ (2013)
51. Eclipse: Paho—JavaScript client. http://www.eclipse.org/paho/clients/js/ (2014)
52. Fielding, R., Gettys, J., JMHFLMPLTBL: RFC 2616—hypertext transfer protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5 (1999)
53. Cisco: the IoT opportunity. http://www.cisco.com/web/solutions/trends/iot/indepth.html (2014)
54. Bergmann, O.: tinydtls web page. http://tinydtls.sourceforge.net/ (2013)
55. Bergmann, O., Gerdes, S., Bormann, C.: Simple keys for simple smart objects. Work smart object secur. http://www.lix.poly

technique.fr/hipercom/SmartObjectSecurity/papers/OlafBergmann.pdf (2012)
56. Cao, Z., Cragie, R., Haberman, B.: Light-weight implementation guidance (lwig) working group. https://datatracker.ietf.org/doc/charter-ietf-lwig/ (2011)
57. Aranha, D.F., Gouvêa, C.P.L.: Relic is an efficient library for cryptography. https://code.google.com/p/relic-toolkit/ (2013)
58. Sethi, M., Arkko, J., Keranen, A., Rissanen, H.: Practical considerations and implementation experiences in securing smart object networks. http://www.arkko.com/publications/draft-aks-crypto-sensors.txt (2012)
59. Thingsquare: Contiki: the open source OS for the internet of things. http://www.contiki-os.org/ (2014)
60. Navarro, M.S.D., Vermillard, J.: Wakaama (LWM2M library). http://eclipse.org/proposals/technology.liblwm2m/ (2013)

Pedro Diogo is a young Software and Networks Engineer with a passion for Entrepreneurship. He founded a startup right after finishing his MSc thesis on Internet of Things and published a paper with IEEE connected with CISTI (Iberian Conference on Information Systems and Technologies). He is currently working at a Portuguese IoT-related company, developing products and services for Smart Cities, while also being actively involved in different European R&D projects, such as EMBERS.

Nuno Vasco Lopes is a researcher of UNU-EGOV (United Nations University Operating Unit on Policy Driven Electronic Governance). He holds two Postdoctoral positions, one in Internet of Things, Computer Science, at the University of Coimbra and another in Electronic Governance at United Nations University. Currently, he is being working at the Universidade do Minho and at the United Nations University. He began his career as a professor in 1998, since than he has been teaching at several public and private universities. During is working life he has been involved in several National, European and International projects such as Electronic Governance for Context-Specific Public Service Delivery, Knowledge Society Policy Handbook, Policy Monitoring on Digital Technology for Inclusive Education, Intelligent Computing for Internet and Services, Internet of Things for Disable People, Smart Defense and Smart Cities for Sustainable Development. He also gives in a regular basis professional courses, seminars and workshops on ICT, elearning, computer networks, cybersecurity, smart cities, among others. His current research interests are Smart Cities, e-Governance, Public Service Delivery, Mobile Networks, Cybersecurity, Quality of Service, Real-Time Services, Vehicular Networks, Nano-Communication and Internet of Things.

Luis Paulo Reis is an Associate Professor at the University of Minho in Portugal and Director of LIACC–Artificial Intelligence and Computer Science Laboratory where he also coordinates the Human-Machine Intelligent Cooperation Research Group. He is a IEEE Senior Member and vice-president of both the Portuguese Society for Robotics and the Portuguese Association for Artificial Intelligence. During the last 25 years he has lectured courses, at the University, on Artificial Intelligence, Intelligent Robotics, Multi-Agent Systems, Simulation and Modelling, Educational/Serious Games and Computer Programming. He was principal investigator of more than 10 research projects in those areas. He won more than 50 scientific awards including wining more than 15 RoboCup international competitions and best papers at conferences such as ICEIS, Robotica, IEEEICARSC and ICAART. He supervised 17 PhD and 100 MSc theses to completion. He organized more than 50 scientific events and belonged to the Program Committee of more than 250 scientific events. He is the author of more than 300 publications in international conferences and journals (indexed at SCOPUS or ISI Web of Knowledge).