This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

A Software Architecture Enabling the Web of Things

L. Mainetti, V. Mighali, and L. Patrono

***Abstract*— The Internet of Things will include billions of smart "things" connected to the Web and characterized by sensing, actuating, and data processing capabilities. In this context, also known as Web of Things, the user should ideally be able to collect information provided by smart things, and to *mash-up* them to obtain value-added services. However, in the current solutions, the access to physical objects is poorly scalable and efficient, the communications are often unidirectional (from the devices to the users), and only tech-savvy people are able to develop mash-up applications. Based on these assumptions, we propose a software architecture to easily mash-up CoAP resources. It is able to discover the available devices and to virtualize them outside the physical network. These virtualizations are then exposed to the upper layers by a RESTful interface, so that the physical devices interact only with their own virtualization. Furthermore, the system provides simplified tools allowing the development of mash-up applications to different-skilled users. Finally, the architecture allows not only to monitor but also to control the devices, so establishing a bidirectional communication channel. To evaluate the proposal, we deeply modify and integrate some existing software components to realize an instance of the architecture.**

***Index Terms*—CoAP, IoT, Mash-up, REST, WoT, WSN.**

I. Introduction

THE ubiquitous and pervasive monitoring systems have always been of great interest to the scientific community, since they represent the meeting point between the physical objects and the digital world. In this perspective, some emerging technologies such as Radio Frequency Identification (RFID) [1] and constrained networks, first of all Wireless Sensor Networks (WSNs), are rapidly emerging as the main kind of distributed pervasive systems [2]. The sensor nodes are connected together to create applications for environmental monitoring, to make cities smarter [3], to easily manage the conditions of our domestic environments [4], etc. To facilitate the communication among these devices, both the industrial and the academic worlds are working on the definition of several low-power protocols, such as Bluetooth, IEEE 802.15.4 [5], and more recently, 6LoWPAN [6] and Constrained Application Protocol (CoAP) [7]. However, although these protocols represent good solutions to make WSN nodes easily reachable through the Internet, leading to the so-called Internet of Things (IoT), embedded devices still form small and separated islands for the application layer.

At a higher level, the Web seems to be the best candidate for a universal integration platform. These considerations have led to the evolution of the IoT concept in the *Web of Things* (WoT) concept [8], according to which all the traditional paradigms of the Web are adapted to the IoT. More in detail, the data produced by smart devices should be directly accessible as normal Web resources, so as providing the so-called *physical mash-up* [9].

However, the widespread dissemination of WSNs is hindered by some drawbacks, mainly related to the power consumption of nodes and to the weaknesses in both programming the network and access to the physical resources. While for the first aspect there are many solutions in the literature focused on minimizing the power consumption at various levels of the protocol stack [10, 11], for the other problems, a dominating solution has not yet emerged: the development of applications based on smart things is currently only accessible to embedded systems experts, and the access to physical resources is often poorly scalable and highly inefficient. Even though many solutions provide high-level interfaces to simplify the development of WoT applications, they are often based on the Hyper Text Transfer Protocol (HTTP), which is too costly for the poor resources of the embedded devices. Instead, more effective solutions based on CoAP suffer from inefficient access to physical resources: some of them require a direct interaction with the embedded devices; other ones expose the resources only through an intermediate database. Obviously, in the first case the computational and storage requirements are too heavy for the embedded devices, whereas in the second case the information provided to the user application could be inconsistent and not up-to-date. Finally, no solution provides really simplified tools to allow the development of WoT applications also to non-expert users.

These considerations have led us to exploit the Web and its emerging technologies as the basis of a complete WoT architecture capable of supporting the development of applications for CoAP-based constrained networks. This architecture meets two main requirements. First of all, it is able to abstract the physical devices, virtualizing them and exposing these virtualizations through a common interface.

Manuscript received October 27, 2014; revised March 31, 2015.

L. Mainetti, V. Mighali, and L. Patrono are with the Department of Innovation Engineering, University of Salento, Lecce 73100, Italy (e-mail: {luca.mainetti, vincenzo.mighali, luigi.patrono}@unisalento.it).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

Then, it allows the development of IoT applications to different-skilled users and by heterogeneous implementation platforms. These requirements represent the core of the proposed approach, as well as the main novelty aspects of the proposal. Indeed, by satisfying the first requirement, the architecture allows to perform a scalable and real-time interaction with embedded devices, since it does not directly communicate with CoAP-based nodes neither exploits an intermediate database. By fulfilling the second requirement, it allows different-skilled users to both develop applications for embedded networks and bi-directionally interact with them. More in detail, the proposal consists of a four-layered architecture, as shown in Fig. 1. At the highest level, there is the composition layer, i.e., the layer closest to the user, which allows the easy composition of the applications. Then, at the next layer, a WebSocket/CoAP proxy server has been designed and implemented in order to enable the communication between the user environment and the running applications. At the next level, the architecture deals with: (i) the virtualization of the physical devices, (ii) the monitoring of their connectivity, and (iii) the execution of the business logic of running applications. At the lowest level, the physical devices have to expose their resources for the WoT applications through a common interface.

The designed architecture is then instantiated by choosing some technologies and software components that, although not fully dedicated to the IoT context and interoperable with each other, have certain characteristics that make them good candidates as components of the designed architecture. Moreover, a functional comparison with another IoT architecture based on CoAP is briefly presented. Finally, the implemented architecture instance has been tested through a simple intrusion-detection case-study application.

The rest of the paper is organized as follows. Section II summarizes the motivation underlying the proposed architecture. The state-of-the-art related to solutions for the implementation of mash-up applications is reported in Section III. Section IV defines the design principles of the proposed WoT architecture. The detailed description of the implemented architecture instance is given in Section V, whereas in Section VI a comparison with another IoT architecture is presented. In Section VII, a case study for evaluating the effectiveness of the architecture instance is described. Conclusions are drawn in Section VIII.

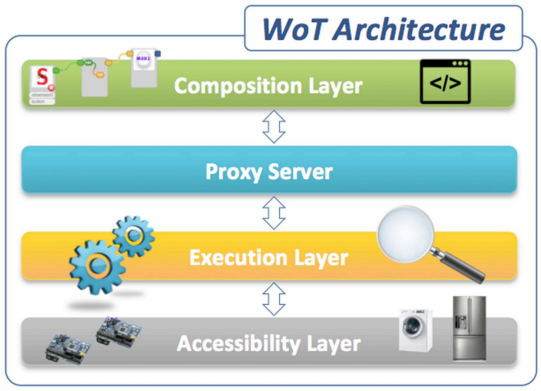

Fig. 1. The four-layered WoT architecture.

II. Motivation

The recent innovations in the ICT field are mainly focused on the development and dissemination of the so-called smart environments. They are characterized by many small and heterogeneous smart objects, which are usually disconnected from the Internet and cannot be controlled without dedicated software and proprietary interfaces. To overcome these limitations, the IoT, or rather the WoT, has mainly focused on establishing Web connectivity and integration among smart things, but there is still no an architecture able to provide a flexible and easy access to their physical resources. To achieve this goal, the proposed approach reuses and adapts patterns commonly used for the Web, focusing on the REpresentational State Transfer (REST) architectural style. However, this scenario involves additional difficulties with respect to the traditional Web, due to the poor computational, memory and energy resources of the embedded devices, which are then not able to manage external interactions in scalable way. In addition, the manufacturers of embedded devices often use different hardware components and follow heterogeneous standards and protocols, so requiring in-depth skills to application developers. For these reasons, it is necessary to enhance the interaction with smart devices by (i) lightening the workload of physical devices, (ii) making the communication with the physical network faster and more robust, and (iii) simplifying the development of WoT applications without worrying about low-level issues. This way, the WoT will be more accessible to all kind of users.

Such a type of approach could be very useful in several fields, ranging from academia to industrial worlds, also including every day life of end-users. From an academic point of view, it would simplify and speed up the implementation and testing of new protocols for constrained networks. In fact, the validation of new protocol solutions is increasingly being done via a test bed approach, that is, by checking "on the field" the behavior of real nodes. This approach requires the execution of time-consuming operations, such as the frequent updating of the code on individual nodes, and heavy storing of data for successive processing. Therefore, a solution for rapid testing and evaluation of new protocols would optimize the research work. From an industrial point of view, companies could exploit this approach to continuously monitor and update their products and networks. For example, consider the post-sales service of a company that sells electronic appliances. Taking advantage of the proposed solution, the company could deploy in the Cloud the programs that manage its appliances. This way, if any malfunction occurs, the company's technicians are immediately notified and can solve the problem quickly, avoiding going to the user's home. Similarly, any software update does not require any action on the product, but simply the replacement of the program running in the Cloud. Last but not least, end-users, more and more involved in the use of embedded devices (e.g., for home automation applications), could exploit a flexible and intuitive

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

software platform to deploy applications for their own needs. They could leverage applications already implemented by other developers or they could autonomously create their own applications.

III. Related Works

Wireless Sensor and Actuator Networks are the core of many recent Internet of Things applications because of their ubiquity. However, the complexity in developing applications for these platforms has been a key issue over the last decade. The advent of IP technologies for WSNs [12] fostered the idea of using the Internet as an integration bus to facilitate applications development: this concept is the core idea of the WoT vision previously described. In this context, one of the first solutions is SensorBase [13], a software tool that considers the sensors network as a set of tables in a relational database, where it is possible to perform queries in SQL-style to process the data coming from sensor nodes. Recent research works take advantage of the development of very small-footprint Web servers implementing full HTTP stacks [14]. So, these approaches are based on the REST architectural paradigm [15], which allows the manipulation of network resources by means of the basic methods of HTTP, that is, GET, POST, PUT, and DELETE. The pioneering idea of using Web protocols and, in particular, RESTful architectures for WSNs was proposed by Drytkiewicz et al. [16] and by Luckenbach et al. [17], who developed two specific protocols, namely pREST and TinyREST, for integrating sensor networks into the Internet through RESTful interfaces. Sensor.Network [18] is another example of the use of the REST paradigm for storing, sharing, searching, and viewing data coming from heterogeneous devices. A more intuitive tool is known as WoTkit [19], a toolkit for the Web of Things. It is a Java Web application that abstracts both sensors and actuators through a single physical model consists of a virtual sensor to which the data coming from physical devices are associated.

In [20], the authors discuss how the REST principles can be applied to embedded devices, by illustrating two concrete implementations: on the Sun SPOT platform and on the Ploggs wireless energy monitors. Then, they show how RESTful interactions can be leveraged to quickly create new prototypes and mash-ups that combine the physical and virtual worlds. In [21], authors describe their idea of Web of Things architecture and the best-practices based on the RESTful principles that have already contributed to the popular success, scalability, and modularity of the traditional Web.

With the convergence of academic and commercial worlds, several Web platforms are emerging with the aim to abstract the heterogeneity of the physical embedded devices by providing access to their resources through HTTP APIs. Famous examples of these Web tools are: Xively [22], ThingSpeak [23], Sen.se [24], and ThingWorx [25]. Using these tools, anyone can access the data made available by smart things localized around the world and see this data through predefined dashboards (charts, tables, graphs, etc.). Moreover, through the available APIs, tech-savvy users can build custom applications that use physical information.

Although the architectures and the platforms described so far are very promising, they implement their RESTful interfaces using the HTTP, which can be quite expensive in terms of computational resources and energy consumption for constrained devices, primarily due to the mechanism for the connection establishment.

An interesting approach to address the HTTP issues is represented by the use of CoAP. In addition to the HTTP features, it offers a built-in mechanism for the resource discovery, it supports the IP multicast, and it natively provides a server-push model and an asynchronous exchange of messages. A recent and promising solution regarding the development of IoT applications based on CoAP is reported in [26, 27]. It exploits the features of this protocol to achieve full interoperability at the application level and to adapt the REST communication paradigm, based on User Datagram Protocol (UDP), to the "things". A key aspect of the proposed framework is the Thin Server [28], a light CoAP server installed on the physical devices that exposes hardware functionalities through a RESTful interface. Another fundamental building block is the application server Actinium that allows the execution of WSN-based applications implemented by using JavaScript. Another solution based on CoAP is presented in [29], where authors propose the so-called uID-CoAP architecture. It combines CoAP with the ubiquitous ID (uID) architecture [30] and aims to host IoT services on common embedded systems, like usual consumer appliances.

In the light of this state-of-the-art, what is still missing is a lightweight and efficient IoT architecture able to provide any user with the ability to monitor and control embedded networks by developing custom applications in an intuitive and easy way.

IV. WoT Architecture Priciples and Design

The development of IoT applications on top of embedded operating systems, protocols and libraries requires specific skills and it is only accessible to experts in embedded systems. For this reason, the overall goal of a WoT architecture should be to facilitate the development of these applications, regardless of the target physical technology. For this reason, the designed architecture follows these general principles:

- Lowering of the entry barrier for developers and foster rapid prototyping. This would allow a wider range of developers, tech-savvy users, researchers and common end-users to develop on top of smart things and would contribute to fostering third party innovation using smart things.
- High usability of the architecture in order to guarantee direct and ubiquitous access for users. They should be able to access and use smart things data and services from everywhere by using different kinds of platforms and systems.
- Lightweight access to smart things data. This allows creating applications in which real-world data are directly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

consumed by resource-constrained devices, such as mobile phones, without requiring high computational and storing resources.

- Remote control, where possible, of the embedded devices, changing their state in accordance with the user requirements. The architecture should provide a bidirectional communication channel through which it allows not only to receive and use information coming from smart things, but also to actuate them.
- Low computational load for embedded devices that are often characterized by both low computational and memory resources.

The fulfillment of these principles has led to several design challenges. The most important ones are the following:

- The embedded devices must be free of any application business logic and only have to expose their resources through a common interface that abstracts their heterogeneous hardware features. This approach is fully in line with the needs of embedded devices, which can act as a small server, exposing their resources as normal Web resources. Therefore, the devices should be agnostic about the business logic, so preserving their computing and memory resources. Of course, this requirement does not exclude that a resource could be the result of processing algorithm inside the node. However, the internal logic should be quite general and decoupled from any specific application. In this way, the same WSN could be exploited by multiple simultaneous applications.
- The central part of the architecture must be able to discover and to index the available resources. In addition, it should be responsible of the business logic execution. Indeed, in order to keep physical devices free from computational tasks, the execution of mash-up applications should run out from the physical layer and has to take care of both the data processing and sending the results to client applications.
- The upper part of the architecture should provide simple APIs allowing developers to deploy their applications quickly. They should be able to collect information from embedded networks through high-level methods and, if allowed, to control physical devices by changing their state. Exploiting these APIs, tech-savvy users can develop their own applications using the preferred programming language, whereas "standard" users might leverage simple software tools, as graphical editors, to visually implement their applications.

Finally, it is important to emphasize that the architecture should not be a "black box", but rather, each layer has to provide APIs for exposing its capabilities. In this way, applications can be built on top of any layer, depending on the requirements of a particular use-case.

V. The Proposed Architecture Instance

The effectiveness of the designed architecture is evaluated by exploiting some technologies and software components suitable for the implementation of an architecture instance. The starting point of this implementation is represented by three main components that are not currently included in the same architecture, but whose features, properly combined, fit the principles described in the previous section. The first component is the CoAP server Erbium [31], an implementation of CoAP for Contiki [32]. The second component is the application server Actinium, a runtime CoAP container that deals with the execution of JavaScript applications interacting with CoAP devices. The last component is the ClickScript editor, a graphic tool for implementing applications running in a browser. It has nothing to do with IoT context, but it is a good starting point for a flexible WoT editor.

At the lowest layer, the Erbium Thin Server was adapted to the available hardware, so that it can provide a RESTful interface for each embedded resource. At the next layer, the attention was focused on both the traceability of IoT devices and the optimization of the communication with the physical network. In particular, the application server Actinium was enhanced and extended to provide key features for the WoT context. It now implements appropriate mechanisms able to continually discover the available nodes and their resources: the fall of the nodes, their reconnection to the network, and the joining of new nodes are fully supported. Moreover, each physical device is virtualized by a software clone to avoid an excessive burdening of its resources. Finally, the management of the server-push mode was integrated in Actium in order to allow smart devices to send notifications upon the occurrence of certain events. Indeed, this operation mode, natively provided by CoAP, is a key aspect in many IoT scenarios, but it is not yet managed by Actinium. Then, in order to allow the communication between the composition layer and the application server, an ad hoc layer was defined and integrated into the architecture. This layer consists of a WebSocket/CoAP proxy server that, on the CoAP side, communicates with Actinium, whereas, on the WebSocket side, provides the real-time notifications to the user interfaces. Finally, the structure and the capabilities of the ClickScript editor were deeply modified and extended to obtain an editor for the WoT. The new GUI provides the user with several graphical elements that model typical IoT components, such as sensors and actuators. Moreover, a mapping algorithm was defined to transform the code listing of a ClickScript mash-up application in a JavaScript file compliant with Actinium requirements. This composition layer, represented by the new editor, allows users to implement their own applications without knowing any specific programming language. The overall description of the architecture is summarized in Fig. 2.

In accordance with the design principles described in Section IV, the user can develop and deploy her/his applications through different approaches: by exploiting the graphical editor, by connecting with the WebSocket/CoAP proxy, or by installing the applications on Actinium. Of course, s/he can also directly "talk" with the embedded devices using the RESTful interface provided by Erbium.

Finally, this architecture instance provides a bidirectional communication channel with the smart network. In one

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

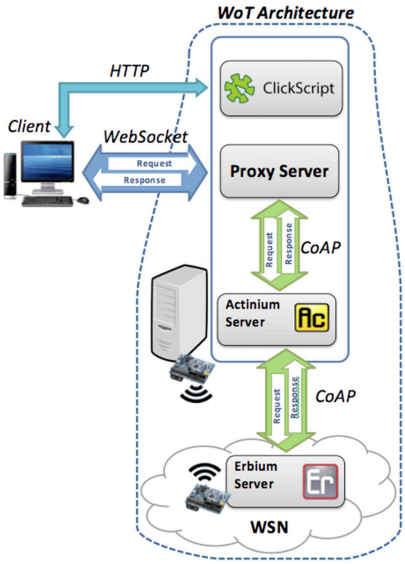

Fig. 2. The overall structure and the components of the implemented architecture instance.

direction, the user receives information and notifications from the network, in the other direction, s/he can control devices (e.g., actuators) in accordance with her/his own needs.

A. Adapting the Erbium Thin Server

The WSN nodes used to validate the proposed architecture were two evaluation boards developed by ST Microelectronics, namely MB851 and MB954 boards. The MB851 boards were used as *smart things*, whereas the MB954 board was used as WSN border router.

These boards are equipped with a temperature sensor, an accelerometer, two LEDs, and a physical button. According to the thin-server paradigm, Erbium should permit to *get* the current state of sensors and actuators, as well as to change the state of the latter. The application logic is instead hosted on the application server Actinium.

More in detail, in order to read the value of the sensors, each of them was registered in Erbium as resources and a handler was defined for each sensor, in the format *[resource_name]_handler*. Upon receipt of a GET request, the handler polls the sensor and builds the response message using the sensor state as payload. The same approach was used for the actuators, but, in this case, the defined handlers are able to respond to GET and POST requests. Then, in order to define an observable sensor on Erbium (i.e., a sensor that an application can subscribe to), it was necessary to implement an *EVENT_RESOURCE* that, at each occurrence of a state change, generates a notification with the new resource state to notify all the observers.

B. Extending the Actinium Server

An application server dedicated to WoT context should provide the user with the capability to efficiently explore physical resources. For this reason, a resource discovery mechanism is needed. In the proposed architecture, this discovery procedure is performed by a JavaScript application, called *discover-motes*, which runs at the Actinium startup and stores the available resources in a resource directory. In particular, it retrieves, from the border-router, the list of the IP addresses of the available WSN devices and, for each IP, it sends a GET request to a specific resource, i.e. *coap://board_ip:coap_port/.well-known/core*. This resource is always available on an embedded device running Erbium Thin Server and it responds by returning a specific string describing the node resources (in accordance with the CoRE Link Format [33]). For each response, the *discover-motes* application installs on Actinium a clone-application that acts as a "proxy" for the corresponding embedded device. More in detail, each proxy application has the same resources of the corresponding node (sensors, actuators, etc.), so as to realize a mapping of physical resources on the Actinium resources. This way, client applications can exploit a more efficient “point of access” to physical resources.

The advantage of using the proxy applications is clear in a subscription/notification scenario. In the discovery phase, Actinium detects the observable resources of the physical nodes, so that the proxy applications can subscribe to them. Therefore, each client application (created by the user) can subscribe to the observable resources of the proxy application, rather than to those of the physical device. In this way, the embedded devices must send notifications only to their proxy applications, which in turn forward all data to the observers (Fig. 3). Without proxy applications, each client application would send its subscription requests to the Erbium server, which would store a large number of observer relationships. Such a situation not only introduces a high computational cost and a critical waste of RAM, but it also leads to a significant data traffic from the physical device to Actinium, thus affecting the energy resource of the device.

As shown in Fig. 3, *discover-motes* links each found IP address with a unique name, which is used to identify the corresponding proxy application on Actinium. This choice enables the identification of physical resources through simple-to-remember names. For example, to refer to a temperature sensor, the user has to write a path such as */apps/running/board3/sensors/temp*, rather than a long and poorly significant IPv6 address. When the user wants to run an application, each ClickScript component (sensor, actuator, etc.) must be associated with the path of the respective physical resource. The pairs *<IPv6 address, assigned name>*,

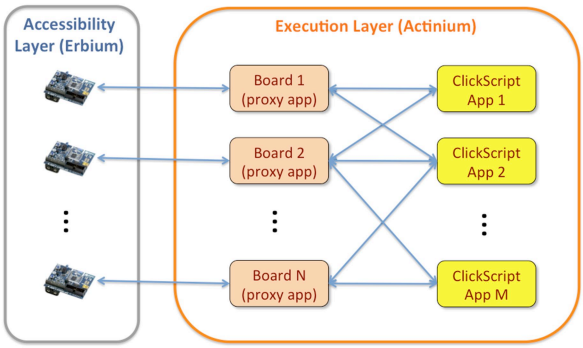

Fig. 3. Proxy applications in a subscription/notification scenario.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

defined in the discovery phase, are obviously stored, so that any changes in the set of physical devices do not generate inconsistency problems if Actinium is restarted.

Moreover, to both discover new nodes in the network and detect disconnection (or re-connection) of current nodes, a specific thread in the *discover-motes* application is integrated. This thread periodically polls, as in the startup phase, the border router to discover new node in the routing table. Then, it checks the known IP by querying the */.well- known/core* resource. If no answer is received, the corresponding proxy application is uninstalled.

C. The WebSocket/CoAP proxy

The WebSocket/CoAP proxy server is the middle component of the architecture, since it enables communications between the ClickScript application and the Actinium server. It has the burden of translating the requests coming from the user interface in CoAP messages directed to the applications running on Actinium and vice versa. The Fig. 4 summarizes the proxy features.

To implement the CoAP side of the proxy server, the JavaScript language has been used, exploiting the capabilities of the Node.js framework [34]. It allows using JavaScript, a typical *client-side* language, for building *server-side* applications. Particular attention was paid to the so-called *blockwise* messages, i.e. messages whose payload is fragmented into several successive packets. Indeed, before sending a response packet to the WebSocket client, the packet itself must be fully and properly received. Therefore, a specific process has been integrated to aggregate fragmented packets. Finally, an ad-hoc mechanism has been implemented, in order to allow the simultaneous observation of multiple observable resources.

Regarding the WebSocket side, it was realized starting from an open-source implementation of the WebSocket protocol for the Node.js framework. The translation between the two protocols managed by the proxy server is carried out by exploiting specific numerical codes included in the WebSocket request, each of which corresponds to a specific command of the CoAP interface of the proxy. More in detail, once the WebSocket request is received, the proper CoAP method is invoked and the CoAP message is created basing on the other parameters of the WebSocket request. Then, the message is sent to the CoAP server Actinium through an UDP socket. In the opposite direction, once the CoAP side of the proxy server receives the response coming from Actinium, it uses this message to create the payload of the response for the WebSocket client, eventually after reconstructing the blockwise packet. This way, the CoAP communication is completely transparent to the ClickScript client.

The proxy also provides storing capabilities, allowing to store ClickScript files and graphical user interfaces on the file system of the gateway.

D. The new ClickScript Architecture

The original architecture of the ClickScript application has been widely extended in order to adapt it to the WoT context. First of all, the applications built through ClickScript are no longer executed locally in the browser, but their execution has been transferred on Actinium and it is controlled by means of graphical dashboards. All interactions between ClickScript and proxy server are done through a WebSocket channel.

The new architecture, shown in Fig. 5, reflects the different stages in which ClickScript can be used. The first three layers are used in the Programming phase. The Library Layer defines the graphical and functional structure of each visual component of ClickScript, in terms of number of inputs, outputs and configuration fields. The Data Model Layer handles the creation of the application control flow, concerning the data and control dependencies among involved components. Then, the Programming Layer allows user to visually define the structure of the application and its dashboard. The Mapping Layer is in charge of the implementation of the mapping algorithm, which translates the visual script into a single JavaScript file. The GUI Layer handles the user interaction with the graphical interface during the Execution phase. It intercepts user actions and translates them into messages for the proxy server. In the opposite direction, it displays in the dashboard the updating messages coming from the server, such as the push notifications. Finally, the Communication Layer implements a WebSocket client for the interaction with the proxy server.

The new architecture structure has implied changes to the ClickScript user interface. The Programming View now includes a dedicated area for the dashboard associated to the application, and a sidebar useful to install the script on Actinium and to interact with all resources defined on it. Two new tabs have been added to run an application already installed (Instantiating View) and to graphically control one or more applications through their dashboards (Execution View). In order to implement applications like the use case discussed in Section VII, new ClickScript components have been defined. Sensor, Observable Sensor and Actuator are used to model the physical resources and to define their graphical user interface for runtime interaction. In particular, the Observable

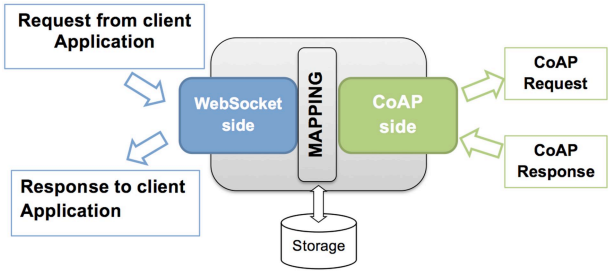

Fig. 4. The WebSocket/CoAP proxy server structure.

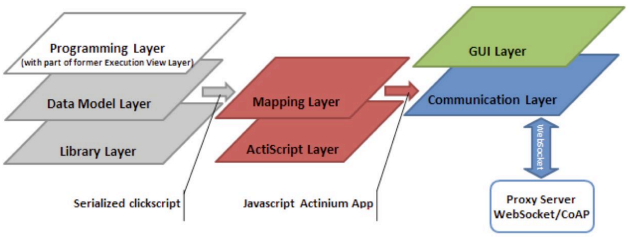

Fig. 5. The new structure of the ClickScript Editor.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

Sensor, called *ObSensor*, is useful to model sensors that send push notification towards the applications.

The mapping algorithm is a key component of this new version of ClickScript. It translates the data-flow of the visual application defined within the Programming View into a sequential JavaScript file structured as an Actinium application. The algorithm consists of three phases: (i) parsing, (ii) component mapping, and (iii) merging. In the parsing phase, it takes the serialized script created in ClickScript and parses it in order to obtain a number of JavaScript objects that wrap all the structural dependencies of application components. During the component mapping phase, for each object, the related ActiScript entity is retrieved. Finally, in the merging phase, all code snippets generated in the previous step are merged together to obtain the Actinium App final listing.

VI. An Architectural Comparison

A brief comparison between the proposed architecture and another solution already described in the Related Works Section [29] is reported. In particular, in [29], the authors propose the so-called uID-CoAP architecture, which combines CoAP with the ubiquitous ID (uID) architecture. The building blocks of this solution are shown in Fig. 6.

First of all, it is important to highlight that a thorough and detailed comparison between architectures for the Internet of Things is anything but simple. Indeed, although all solutions are aimed at the integration of the emerging technologies in the next Internet, the specific goals of each architecture change also considerably. Moreover, the applicative contexts often change, so making the design requirements very heterogeneous. For these reasons, the purpose of this section is not to make a deep comparative analysis between the two architectures, but to carry out a comparison in order to bring out the uniqueness of our solution.

At the lowest level (the physical one), both solutions adopt the REST paradigm provided by CoAP as the interface exposing the resources of the physical devices. However, while the uID-CoAP architecture leverages T-Kernel [35], the proposed solution is based, as mentioned, on Erbium, a CoAP implementation for Contiki. The choice of Erbium is due to the fact that Contiki is more widespread and it seems to have a more active development community.

As regards the remaining layers, the comparison between the two architectures highlights clear differences. The uID-CoAP architecture requires the presence of a database, i.e. the uID-based Database System, which fetches the data from the physical devices using the REST interface. These data are then stored and made available to client applications, possibly together with additional information about the device. Instead, in the proposed solution, each physical device is cloned by a software proxy application that takes care of the interaction with client applications. These different approaches involve substantial differences in the performance of the two systems. In the uID-CoAP architecture, the semantic database periodically sends GET commands to physical nodes in order to collect their data, which are then stored asynchronously with respect to the requests of the user applications. This behavior leads to two main problems: unrequested messages and significant overhead on the CoAP network, and possible inconsistency of fetched data. In particular, if the update requests of the database are frequent, the physical network is overloaded with potentially unuseful data, i.e., data not actually required by the user applications. On the other hand, if the database interacts with the physical devices less frequently, the data stored in the database could be not up-to-date and therefore they could be inconsistent with the current network state. Obviously, this negative effect increases with the devices in the network. To prove these considerations, we have emulated the uID-CoAP architecture by storing the physical data in a database, without exploiting the proxy applications. As result, Fig. 7 shows the probability for successful (i.e., up-to-date) fetching for different polling periods (10, 20, 30, 40, 50, and 60 seconds) and different number of appliances (10, 30, 50). To give statistical significance to the results, each user application has randomly queried the database. As shown, the probability decreases with the increasing of the polling period. On the other hand, if the polling period is short, the successful probability is high, but also the overhead increases. In the proposed solution, these problems are totally absent. In fact, user applications interact with the software clones of embedded devices, which in turn interact with the physical network. This approach provides always up-to-date data and avoids unnecessary transmissions over the network. The only potential weakness of the proposed

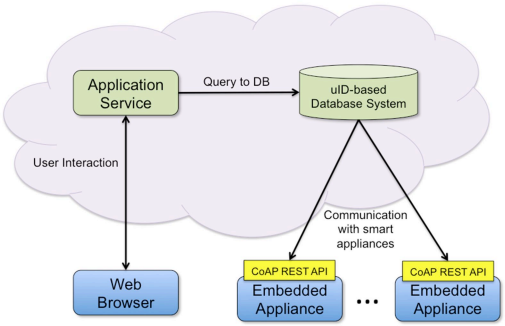

Fig. 6. uID-CoAP architecture structure.

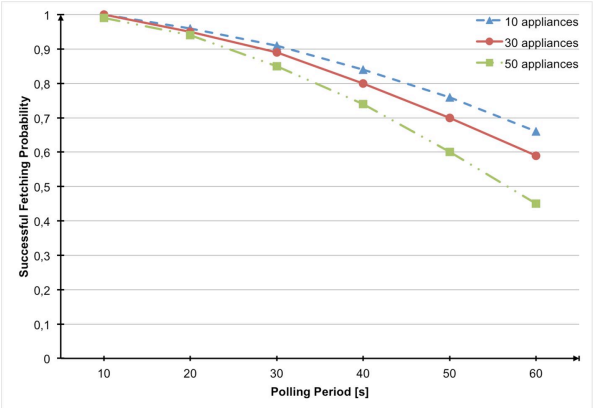

Fig. 7. Successful fetching probability of uID-CoAP architecture for different polling period and different number of appliances.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

solution could be the response time of the network. However, since the communication between the proxy applications and the related real devices takes place locally, the response delay is largely negligible. Moreover, this kind of interaction makes extremely fast and scalable the subscription/notification process, which provides real-time push notifications (not provided by uID-CoAP architecture at all). In addition, uID-CoAP architecture does not provide specific mechanisms to monitor and manage the physical connectivity of the devices. On the contrary, the proposed architecture is equipped with appropriate procedures to keep the nodes availability always up-to-date, so as to guarantee the consistency and robustness of the applications provided to the users.

Another important difference between the two architectures is represented by the "application layer". It is worth noting that, in this analysis, the “application layer” represents both the layer exploited by the user for interacting with the architecture and the mean to build the mash-up applications for CoAP-based WSNs. That said, the comparison does not concern the simplicity in use the interface for communicating with the embedded network, but the user-friendliness in leveraging it to autonomously develop new applications. Regarding this issue, in [29], client applications, i.e. applications that query the semantic database, must be implemented by tech-savvy users, exploiting the APIs provided by the architecture. Instead, our solution, even allowing the independent development of client applications, pushes further the boundaries of the WoT. It is not only moving closer to developers, but also to end-users, enabling them to create simple composite applications on top of smart things and to control them through user-friendly dashboards. Indeed, the physical mash-up Editor, which is part of the Composition Layer, allows the visual implementation of applications and their quick installation on the application server. If a user cannot develop an application on his own, s/he might exploit the applications already developed by other users, by instantiating them on the desired physical devices.

Table I summarizes the differences described so far. As can be deduced from the table, both solutions are perfectly inherent to the new IoT vision and provide useful features to facilitate the integration and interoperability of heterogeneous physical devices. However, the proposed solution is more user-oriented, so it better serves the purpose of making the WoT more accessible.

TABLE I
FUNCTIONAL COMPARISON

	Proposed WoT architecture	uID-CoAP architecture
CoAP RESTful Interface	✓	✓
Connectivity Management	✓	
Lightweight data communication	✓	
Full real-time support	✓	
Semantic metadata		✓
Subscription/notification paradigm	✓	
Simple Application-level APIs	✓	✓
User-friendly composition layer	✓	

VII. A PROOF-OF-CONCEPT: ACCESS CONTROL

A sample access control application has been implemented to validate the previously described architecture instance from a functional point of view,. The scenario consists of a Wireless Sensor Network with a contact sensor node, which monitors the state of a window, and an ON/OFF actuator node, which directly drives an alarm system (Fig. 8). In order to guarantee a reliable access control system, the contact sensor (simulated through a button) is modeled as an observable sensor, i.e., a resource that generates a notification whenever its state changes. In this way, if the window is opened, the alarm (simulated through a led) is fired and a push notification is sent to the user. The application runs in the Actinium server installed on the gateway and consists of an infinite cycle that waits for a notification from the observable sensor. The pseudo-code of the application is the following:

```
period = 0.5 (seconds)
while (true) {
  if (sensor_notification && open) {
    if (actuator = OFF) actuator = ON
  } else if (sensor_notification && closed) {
    if (actuator = ON) actuator = OFF
  }
  sleep(period)
}
```

Each time the button is clicked, the Erbium Thin Server increments a click counter, which is then converted to a *boolean* value that indicates the window state (open/closed). If the window is opened the led is turned on and an update is sent to all the observers.

A remote user can visually create and control this application by connecting, locally or remotely, with the public gateway and downloading the ClickScript interface on his/her browser. If the connection occurs over the public Internet, a Virtual Private Network (VPN) provides the needed security level. This tunnel is based on IP Security (IPSec) for desktop connections and on Point to Point Tunneling Protocol (PPTP) for mobile connections. Then, s/he can create the control flow of the algorithm by selecting the proper components from the toolbar in the Programming View and connecting them according to their data dependencies (Fig. 9). As shown in the

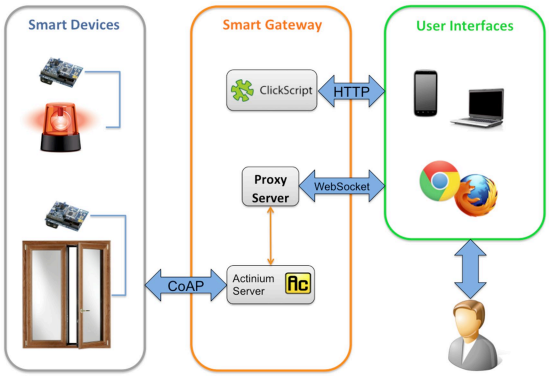

Fig. 8. The scenario for the proof-of-concept.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

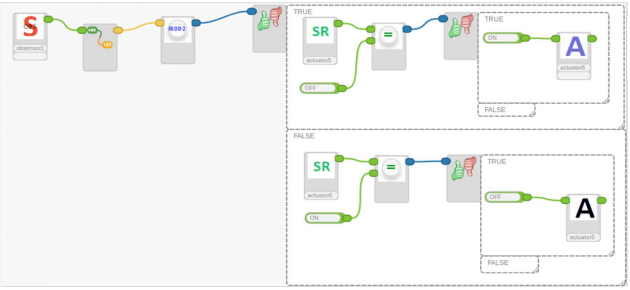

Fig. 9. Data-flow logic of the proof-of-concept application.

Fig. 9, the first component is represented by an observable sensor, whose output is first converted from *String* to *Int* and then processed by the *mod2* operator. Checking the result of this last operation, the application decides whether the alarm has to be started or not. While the script components are added to the programming area, the dashboard for controlling the application is automatically created in the lower section of the Programming View. Once the application has been completely defined, it can be installed on the application server Actinium. At the same time, a template of the application Configuration File is also created and stored on the file-system of the proxy server, as well as the dashboard template. After these steps, only the application logic and the graphical dashboard are defined and stored, but no reference to specific physical resources is defined. In order to execute the application, an instance must be configured and started in the Instantiating View, by selecting the desired application and filling in its configuration form. The resource URIs can be selected from the Running Resources listed in the right menu. To control the execution of an application instance, the user can simply select its name in the Available Applications list and click the *Load UI* button. This way, the application dashboard is loaded in the Execution View, so that the user can control the remote devices and automatically display all updating messages. It is clear that the use of the proposed architecture entails benefits even for this simple use case. By exploiting the intuitive graphical blocks provided by the system, the access control application can be easily developed by users without specific programming skills. In addition, the software virtualization of the physical devices allows a more efficient communication between the users and the embedded network. In particular, the devices can timely update multiple users (e.g., the residents and any other authorized person), without burdening the limited resources of the physical devices. Finally, the separation between the business logic and the configuration of the applications allows to manage the system in a very flexible way: the real devices can be replaced without affecting the business logic, the system can be extended by simply creating new instances of the application, and so on.

In order to prove that the proposed architecture allows tech-savvy users to autonomously develop their own applications, a simple testing *App* was developed on an Android mobile device to directly interact with Actinium. This application exploits the Californium library to build CoAP message by itself, so bypassing the WebSocket/CoAP proxy server. At the startup, the application sends a CoAP DISCOVERY message to Actiunium in order to know what resources are available in the network. For each resource, a Java object is instantiated. In this case, the application can discover only one or more observable contact sensors and one alarm actuator. Of course, an OBSERVE message is sent to Actinium for each contact sensor, which is then associated to a window according to the name of the resource. As shown in Fig. 10, every time a window is opened, a notification arrives to the smartphone and a red dot appears on the interested window. By clicking on the red dot, the user can simply verify the state of both the window and the alarm (the application sends a GET to the corresponding resource).

VIII. CONCLUSIONS

In this work, a WoT architecture has been defined and implemented to facilitate the communication with CoAP-based networks. The architecture supports the interaction with WSN nodes at various levels: from a direct communication with the physical devices up to a composition layer that simplify the developers' effort. This layer provides simple APIs that can be exploited in several ways. Tech-savvy users could use them to develop their own mash-up applications by means of any programming languages, whereas non-expert user could exploit a simple graphical Editor. The core component of the architecture is a middleware that executes all the business logic of the system. It also virtualizes the physical devices in order to improve the efficiency of the communications between them and the user applications.

The designed solution was evaluated by choosing some technologies and software components that, although not fully dedicated to the IoT and interoperable with each other, have been deeply modified and combined for the implementation of an architecture instance.

To improve and extend present work, it is already under definition and development an indexing and discovery service, in order to allow the use of multiple remote networks for the implementation of more complex and smart applications.

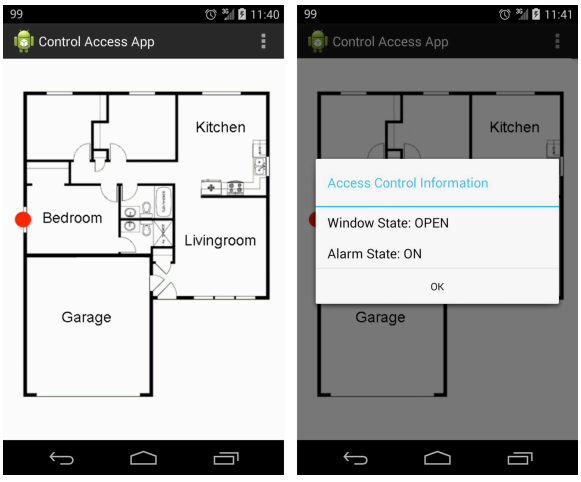

Fig. 10. Android test application to discover and use CoAP resources.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2015.2477467, IEEE Internet of Things Journal

References

[1] L. Catarinucci, R. Colella, L. Mainetti, V. Mighali, L. Patrono, I. Sergi, and L. Tarricone, "Near field UHF RFID antenna system enabling the tracking of small laboratory animals," *International Journal of Antennas and Propagation*, vol. 2013, no. 713943, 10 pages, 2013.

[2] L. Mainetti, L. Patrono, and A. Vilei, "Evolution of wireless sensor networks towards the Internet of Things: a survey," in *Proc. 2011 International Conference on Software, Telecommunications and Computer Networks*, *SoftCOM 2011*, 2011, pp. 16-21.

[3] A. Zanella, N. Bui, A. Castellani, and L. Vangelista, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. PP, no. 99, to be published.

[4] G. De Luca, P. Lillo, L. Mainetti, V. Mighali, L. Patrono, and I. Sergi, "The use of NFC and Android technologies to enable a KNX-based smart home," in *Proc. 2013 International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2013*, 2013, Article number 6671904, pp.1-7.

[5] IEEE Standard for Local and metropolitan area networks - Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC Sublayer, 802.15.4e, 2012.

[6] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC4944, 2007.

[7] The Constrained Application Protocol (CoAP), RFC 7252, June 2014.

[8] D. Guinard, V. Trifa, and E. Wilde, "A Resource Oriented Architecture for the Web of Things," in *Proc. IoT*, Tokyo, Japan, 2010, pp. 1 – 8.

[9] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proc. DAC*, Anaheim, California, 2010, pp. 731–736.

[10] L. Anchora, A. Capone, V. Mighali, L. Patrono, and F. Simone, "A novel MAC scheduler to minimize the energy consumption in a Wireless Sensor Network," *Ad Hoc Networks*, vol. 16, pp. 88–104, 2014.

[11] L. Catarinucci, S. Guglielmi, L. Mainetti, V. Mighali, L. Patrono, M.L. Stefanizzi, and L. Tarricone, "An Energy-Efficient MAC Scheduler based on a Switched-Beam Antenna for Wireless Sensor Networks," *Journal of Communication Software and Systems*, vol. 9, no. 2, pp. 117–127, June, 2013.

[12] J.W. Hui and D.E. Culler, "IP is dead, long live IP for wireless sensor networks," in *Proc. SenSys 2008*, Raleigh, NC, USA, 2008, pages 15–28.

[13] M. H. Gong Chen, N. Yau, and D. Estrin, "Sharing sensor network data," CENS, Los Angeles, CA, Tech. Rep. 71, March 2007.

[14] S. Duquennoy, G. Grimaud, and J.J. Vandewalle, "The Web of Things: interconnecting devices with high usability and performance," in *Proc. ICESS 2009*, HangZhou, Zhejiang, China, 2009, pp. 323–330.

[15] L. Richardson and S. Ruby, "RESTful web services," O'Reilly Media, May 2007.

[16] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, "pREST: a REST-based protocol for pervasive systems," in *Proc. MAHSS*, 2004, pp. 340–348.

[17] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST: A protocol for integrating sensor networks into the internet," in *Proc. REALWSN*, Stockholm, Sweden, 2005.

[18] V. Gupta, A. Poursohi, and P. Udupi, "Sensor.Network: An open data exchange for the web of things," in *PERCOM Workshops*, Mannheim, 2010, pp. 753–755.

[19] M. Blackstock and R.Lea, "WoTKit: a lightweight toolkit for the web of things," in *Proc. WoT 2012*, New York, NY, USA, 2012.

[20] D. Guinard and V. Trifa, "Towards the Web of Things: Web Mashups for Embedded Devices," in *Proc. WWW 2009*, Madrid, Spain, 2009.

[21] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the Web of Things," in *Proc. IOT 2010*, Tokyo, 2010, pp. 1–8.

[22] "Public Cloud for the Internet of Things". Internet: https://xively.com [May. 20, 2014].

[23] "Internet of things – ThingSpeak". Internet: https://www.thingspeak.com/ [May. 16, 2014].

[24] "Feel, act, make sense - sen.se". Internet: http://open.sen.se/ [May. 2, 2014].

[25] "ThingWorx the 1st application platform for the connected world". Internet: http://www. thingworx.com/ [April. 12, 2014].

[26] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications," in *Proc. IoT 2012*, Wuxi, China, 2012, pp. 135–142.

[27] L. Mainetti, V. Mighali, L. Patrono, P. Rametta, and S.L. Oliva, "A novel architecture enabling the visual implementation of web of Things applications," in *Proc. 2013 International Conference on Software, Telecommunications and Computer Networks*, *SoftCOM 2013*, 2013, pp. 1-7.

[28] M. Kovatsch, S. Mayer, and B. Ostermaier, "Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things," in *Proc. IMIS 2012*, Palermo, Italy, 2012, pp. 751–756.

[29] T. Yashiro, S. Kobayashi, N. Koshizuka, and K. Sakamura, "An Internet of Things (IoT) architecture for embedded appliances," in Proc. 2013 IEEE Region 10 Humanitarian Technology Conference (R10-HTC), Sendai, 2013, 314-319.

[30] N. Koshizuka and K. Sakamura, "Ubiquitous ID: Standards for Ubiquitous Computing and the Internet of Things," *IEEE Pervasive Computing*, vol. 9, no. 4, pp. 98-101, 2010.

[31] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A Low-Power CoAP for Contiki," in *Proc. MASS 2011*, Valencia, Spain, 2011, pp. 855–860.

[32] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki – a lightweight and flexible operating system for tiny networked sensors," in *IEEE LCN 2004*, Florida, USA, 2004, pp. 455–462.

[33] Z. Shelby and Sensinode, "Constrained RESTful Environments (CoRE) Link Format," RFC 6690, August 2012.

[34] "Node.js". Internet: http://nodejs.org/ [Dec. 28, 2013].

[35] J. Krikke, "T-Engine: Japan's ubiquitous computing architecture is ready for prime time," *IEEE Pervasive Computing*, vol. 4, no. 2, pp. 4–9, 2005.

Luca Mainetti is an associate professor of software engineering and computer graphics at the University of Salento. His research interests include web design methodologies, notations and tools, services oriented architectures and IoT applications, and collaborative computer graphics. He is a scientific coordinator of the GSA Lab - Graphics and Software Architectures Lab and IDA Lab - IDentification Automation Lab at the Department of Innovation Engineering, University of Salento.

Vincenzo Mighali received the "Laurea" Degree in Computer Engineering with honors at the University of Salento, Lecce, Italy, in 2012. Since January 2009 he collaborates with IDA Lab — IDentification Automation Laboratory at the Department of Innovation Engineering, University of Salento. His activity is focused on the definition and implementation of new tracking system based on RFID technology and on the design and validation of innovative communication proctocol aimed to reduce power consumption in Wireless Sensor Networks. He is also involved in the study of new solutions for building automation. He authored several papers on international journals and conferences.

Luigi Patrono received his MS in Computer Engineering from University of Lecce, Lecce, Italy, in 1999 and PhD in Innovative Materials and Technologies for Satellite Networks from ISUFI-University of Lecce, Lecce, Italy, in 2003. He is an Assistant Professor of Network Design at the University of Salento, Lecce, Italy. His research interests include RFID, EPCglobal, Internet of Things, Wireless Sensor Networks, and design and performance evaluation of protocols. He is Organizer Chair of the international Symposium on RFID Technologies and Internet of Things within the IEEE SoftCOM conference. He is author of about 80 scientific papers published on international journals and conferences and four chapters of books with international diffusion.