Research Article

International Journal of Distributed Sensor Networks

International Journal of Distributed Sensor Networks
2017, Vol. 13(3)

DOI: 10.1177/1550147717698969
journals.sagepub.com/home/ijdsn
SAGE

Run-time deployment and management of CoAP resources for the Internet of Things

Cristian Martín, Manuel Díaz and Bartolomé Rubio

Abstract

The continuous growth of the Internet of Things in recent years has meant it is increasingly more present, as Internet of Things scenarios such as smart homes and smart cities become part of our everyday lives. The Internet of Things devices involved can be divided into two categories in most Internet of Things scenarios. The devices can constitute a black box with specific sensors which complicates their configuration, for example, wearable products. Other Internet of Things devices can be composed through configurable microcontrollers, enabling customizable environments to be designed. However, the necessary tools and knowledge for programming and configuring microcontrollers are not accessible to everyone. This article proposes a run-time deployment and management system through the Constrained Application Protocol that bridges the gap between end users and customizable environments. With our system, end users can incorporate new sensors or actuators in their installed microcontroller without having to access and program the microcontroller board. Rather, they can manage the resources of the Constrained Application Protocol servers through an accessible and transparent Web user interface.

Keywords

Internet of Things, Constrained Application Protocol, Constrained Application Protocol resources, run-time deployment, sensors and actuators

Date received: 22 November 2016; accepted: 16 February 2017

Academic Editor: Suat Ozdemir

Introduction

The Internet of Things (IoT) is a recent group of technologies, which comprises already established technologies such as radio-frequency identification (RFID) and wireless sensor networks (WSNs).[1,2] The IoT is present in a wide range of diverse areas such as manufacturing, smart cities, and connected health.[3] The continuous advances in electronics, processes, and software engineering have all contributed significantly to reduce the production costs and increase the number of IoT devices involved. However, the large numbers of released devices in recent years have caused some heterogeneity, in addition to vendor lock-in issues.[4] This has led to a lack of standards in the IoT for interconnecting and communicating with most of the underlying devices involved.[5]

The Internet Engineering Task Force (IETF) has contributed with many standards for constrained devices and the IoT. Specifically, its Constrained RESTFul Environments (CoRE) working group was responsible for converting REST web services in

Department of Computer Sciences and Languages, University of Málaga, Málaga, Spain

Corresponding author:
Cristian Martín, Department of Computer Sciences and Languages, University of Málaga, Boulevar Louis Pasteur 35, 29071 Málaga, Spain.
Email: cmf@lcc.uma.es

constrained devices, releasing the Constrained Application Protocol (CoAP).[6] Nowadays, REST web services are one of the main accessible and standard ways to intercommunicate systems over the Internet. Therefore, CoAP, although it is intended to follow the same style as REST web services, also uses User Datagram Protocol (UDP), thereby reducing the Transmission Control Protocol (TCP) overhead and header. CoAP also defines the guidelines for a mapping with the Hypertext Transfer Protocol (HTTP), thereby furthering the development of the IoT toward the Web of Things (WoT).

The resources in CoAP represent an operation on a constrained web service. Resource operations can contain both sensors (e.g. a temperature sensor) and actuators (e.g. a light actuator). Normally, resources are defined in compile time which does not permit managing the resources in run-time. Although CoAP defines the guidelines for creating resources in run-time through a POST request, it does not establish the rules for managing and deploying resources with physical components such as sensors and actuators. The goal of this article is to leverage CoAP's capabilities for creating and serving resources, and provide mechanisms for controlling the resources with physical components in run-time. The latter does not require programming or installing external components, so end users could incorporate new sensors or actuators in their installed microcontroller without having to access and program the microcontroller board.

The run-time CoAP resource control system proposed in this article has also been integrated with an HTTP-CoAP cross-proxy and a Web user interface (UI), so end users can manage the resources of the CoAP servers through an accessible and transparent Web UI. We have also taken into account the constraint of storage in the resource-constrained devices, because some sensors or actuators require extra libraries to operate. Thus, each CoAP server in our architecture defines an objective middleware. We have also modified the CoAP protocol to support communication in constrained devices, in this case ZigBee. ZigBee[7] is a communication protocol based on IEEE 802.15.4. ZigBee has been chosen because it provides a low-power operation, and a secure and global communication for the IoT. Nevertheless, as mentioned in 11 IoT protocols you need to know about,[8] the IoT can be composed by different means. The abstraction of these different means in the architecture is included in our roadmap. Currently, ZigBee is the only one supported, but the system has been designed to easily include new ones. The work shown here is also part of our work on the λ-CoAP architecture.[9] In fact, the Smart Gateway, the Web UI, and the CoAP Middleware (CoM) have been extended to support the run-time CoAP resources control system proposed. CoM is an IoT middleware based on the CoAP protocol which provides an embedded system to interact with and manage physical resources in IoT microcontrollers. CoMs also provide a run-time management system to manage and interact with physical resources dynamically. Thus, the main contribution of this work consists of providing an embedded system to manage and deploy physical sensors and actuators in the IoT.

The remainder of this article is structured as follows. In section "Related work," we discuss the related work on the run-time CoAP resource control system. The "CoAP" section presents an introduction of CoAP. In section "λ-CoAP architecture," an introduction of our work on λ-CoAP is given. The overall architecture for the proposed scenario is presented in section "Run-time deployment and management of CoAP resource architecture." Section "Implementation and evaluation" describes the implementation of support for the aforementioned scenario and the evaluation. Finally, section "Conclusion and future work" presents our conclusions and outlines our plans for future work.

Related work

Traditionally, deploying and obtaining information from the physical world implies using highly engineered instruments and wired control protocols.[10] However, the newest challenges focus on systems which support programming and manipulating the physical world so that devices can form themselves as general-purpose devices and support different application types.

The most recent proposals focus on integrating the physical world into the digital world and the Internet. One example is the SENSEI project.[11] The SENSEI project integrates WSNs into a business-driven architecture and offers applications and services over them. OpenIoT[12] is another project which offers a middleware for integrating and implementing IoT solutions with a range of functionalities. Although the integration of the IoT with the Internet is addressed in many approaches, including this proposal, the run-time management and deployment of physical components in both is still necessary to control physical components.

Other proposals focus on deploying plug-and-play infrastructures.[13] Nevertheless, even though plug-and-play makes physical components instantaneously available, the components involved in the plug-and-play process correspond to autonomous devices that go beyond the sensors and actuators used in our proposal. In Bröring et al.,[14] another plug-and-play sensor infrastructure for deploying sensors is presented. The infrastructure can plug-in sensors and interpret their values through a sensor interface description. Sensors are subsequently available in the system when the drivers are

defined. However, it focuses on the USB and IEEE 1451 standards which are usually found in computers instead of microcontrollers. The CoM proposed here is one example of a plug-and-play component. Yang et al.[15] recognized the inefficiency of most plug-and-play protocols used, in terms of energy and memory usage, and presented μPnP, a hardware and software solution for the automatic integration of peripherals in resource-constrained devices. The hardware solution is based on the passive electrical characteristics of the components of additional hardware circuit added in the peripherals to uniquely identify them. The current prototype uses mini high-definition multimedia interface (HDMI) connectors to connect the peripherals. Once the peripherals have been plugged into the system, their drivers are automatically installed and downloaded when they are not locally available. Once each driver has been installed, it is loaded and the peripherals are ready to use. However, although μPnP enables automatic peripheral identification and use, the solution requires additional hardware circuits and connectors, and the interconnection is only allowed with Advanced Direct Connect (ADC), Inter-Integrated Circuit (I^2C), Serial Peripheral Interface (SPI), and universal asynchronous receiver/transmitter (UART) protocols. The solution presented in this article does not require additional hardware to install sensors, and the management can be done remotely through a web on a smart-phone or tablet if the sensors are already connected to the microcontroller.

An extension of the global sensor network (GSN) middleware, a middleware which facilitates the deployment and programming of sensor networks, is proposed in Perera et al.[16] GSN provides a connection with a microcontroller and sensors which is defined semantically through virtual sensors. Virtual sensors abstract the implementation details from accessing sensors through the semantic definition of data stream. Once a virtual sensor has been defined, GSN looks to see whether there is one wrapper for each data stream defined in the wrapper repository. If the wrapper exists, it is instantiated in the middleware, and a new connection with the desired sensor is established. Although GSN allows semantic connection with new sensors without programming, the authors identify that this process requires the wrappers to be developed manually and there is no code sharing between wrappers. The authors have defined a new sensor device definition which generates and compiles new wrappers based on semantic definitions. Therefore, the programming and sharing code problems in wrappers apparently disappear. However, even though this solution enables sensor connection semantically and without programming, this solution is focused on the connection with sensors and microcontrollers which offer application programming interfaces (APIs) or third-party libraries to access their sensor readings and does not provide the mechanisms to instantiate and manage new sensors and actuators.

Employing web services in embedded devices could constitute another solution for managing sensors and actuators in run-time, for example, the device profile for web services (DPWS)[17] and RESTful web services.[18] Web services are currently one of the most open and extended ways of connecting things over the Internet. Nevertheless, the complexity due to the HTTP and TCP protocols can be too heavy for embedded devices. Our protocol for interconnecting things, CoAP, can provide the benefits of the RESTful web services while simultaneously displaying a better performance than the protocols that use other web services.[19] The complexity of the web services can also be moved out to the Gateways,[20] but it implies a dependency on Gateways. The Gateway involved in our solution facilitates the deployment and access to the underlying devices, since it translates protocols and connects to the cloud. Nevertheless, the underlying devices can be accessed directly without the Gateways through CoAP.

The run-time reconfiguration and upgrading of resource-constrained devices is a hot topic addressed in many papers.[21,22] Ruckebusch et al.[21] present GITAR, a system that enables run-time upgrading of network stacks and applications in resource-constrained devices. REMOWARE, a composed-based middleware for reconfiguring sensor networks dynamically, is also presented in Taherkordi et al.[22] Network and firmware updates are necessary requirements to maintain the IoT running over time and actuate over possible bugs or security breaches. In the scope of these approaches, updates can incorporate new libraries into the system, but there is still a lack of a component which can provide the mechanism to instantiate and manage libraries through well-defined interfaces, end users, and even external applications as presented in this article.

CoAP

The efforts of the CoRE working group to apply the REST architecture in resource-constrained devices (e.g. microcontroller with memory and processing limits) have resulted in CoAP. The main goal of CoAP is to provide resource-constrained devices with the REST architecture, optimizing it for machine-to-machine (M2M) applications.

A CoAP request, as with HTTP, is initiated by a client and includes an action over a resource identified through a Uniform Resource Identification (URI). Once the server has processed the request, a response with a status code is sent. However, CoAP provides an asynchronous communication with optional reliability

through UDP. CoAP defines four types of messages: non-confirmable, confirmable, acknowledgement, and reset. Therefore, the reliability is provided in CoAP through the confirmable and acknowledgement messages. Moreover, CoAP also supports built-in discovery and multicast.

One of the main advantages of CoAP with respect to HTTP is its header size. The CoAP header is only 4 bytes which includes information such as the CoAP version, the message type, the operation type, the token size, and a message ID. Optionally, a token for matching requests and responses, options for adding parameters such as the URI and the content format, or a payload for inserting data in both requests and responses can be included.

CoAP makes use of the GET, POST, PUT, and DELETE operations in a similar way to HTTP. The GET method retrieves the information of the resources identified by the request URI. PUT methods process the information provided in the request. The POST method updates or creates the resource of the request. And finally, the DELETE method deletes the resource identified in the request. That semantic has been leveraged in our design to interact over the sensors or actuators (GET and PUT, respectively), creating and editing resources (POST and PUT, respectively), and removing them (DELETE). The CoAP response codes only include a subset of the HTTP response codes, just the most significant ones such as the content information, the changes, or some error codes.

CoAP also enables the operation to be observed whereby clients can subscribe to resources accordingly. In other words, the clients can subscribe to the resources in which they are interested through a CoAP request with the observe option. Once they have subscribed, CoAP servers send the status of the subscribed resource asynchronously when it changes or the set time expires. Clients can unsubscribe from the observation with a reset message.

λ-CoAP architecture

Even though CoAP can abstract the heterogeneous issues and make devices accessible through the Internet, processing, storing, and networking are still limited due to the large amount of data generated by the IoT. Data from the IoT have grown exponentially with respect to the number of devices connected[23] which is predicted to reach anywhere from 20 to 50 billion connected devices according to various estimations. Due to the continuous expansion of IoT and the needs of applications and environments which require analyzing and storing large amounts of IoT data, including real-time requirements, it is necessary to look to the future.

Cloud computing is an emerging and prominent technology which is present in many areas including the IoT, thanks to the virtually unlimited set of configurable resources offered (storage, processing, networking, etc.). Cloud computing is an ideal technology to complement and reduce the limitations of the IoT, a paradigm also known as the Cloud of Things.[24] The need to protect large amounts of data from human error and enable real-time processing of arbitrary functions over arbitrary data led Nathan Marz to define the Lambda architecture.[25] The Lambda architecture is a paradigm composed of cloud computing components which provides the capabilities for consuming, processing, and analyzing large amounts of arbitrary data in real-time. A key concept in the Lambda architecture is the precomputed view which contains the knowledge resulting from processing data. The Lambda architecture splits the data flow into three layers: the *real-time layer*, which processes the real-time data and generates real-time precomputed views; the *batch layer*, which processes the historical data and creates complex batch precomputed views; and finally, the *serving layer*, which offers a way to access and display all generated views. Real-time processing and analysis requirements are crucial in certain areas such as critical systems and connected health, so the IoT needs to leverage that.

Our work on the λ-CoAP architecture[9] combines the best of both worlds, the CoAP and the Lambda architecture. On the one hand, the IoT heterogeneity inside the underlying devices involved is abstracted at the same time as the lightweight web services are enabled so as to access them through CoAP. On the other hand, the Lambda architecture enables the real-time processing, actuation, and analysis of the large amount of data generated by the IoT. The work is complemented with a Smart Gateway to interconnect the Lambda architecture, HTTP, and CoAP; a CoM to install the CoAP in resource-embedded devices; and external components such as a Web UI and an action component to manage and actuate over the underlying IoT, respectively. The work presented in this article complements the λ-CoAP architecture enabling the deployment and management of resources in run-time.

Run-time deployment and management of CoAP resource architecture

As mentioned above, the overall architecture comprises three differentiated components. On the one hand, the Web UI provides a user-friendly and accessible way to manage the resources belonging to the associated CoMs. Through the Web UI, users can either create a new resource on a CoM or edit and remove the previously created resources. On the other hand, a Smart Gateway provides a proxy for translating HTTP to CoAP which is not only used by the Web UI to control the CoMs but could also be used for external HTTP

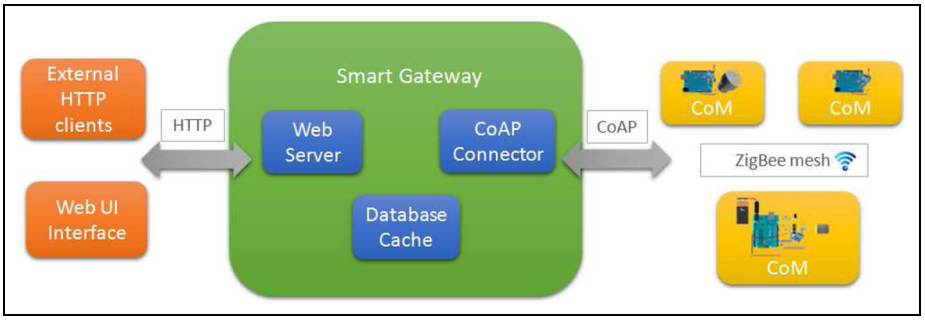

Figure 1. Architecture overview

clients through its REST interface. The Smart Gateway also comprises a Database/Cache which stores information about the resources of the connected middlewares and their data in order to protect them from high access rates. Finally, the CoM offers a CoAP server and a set of resources which can be managed by the Smart Gateway and the Web UI.

Figure 1 shows an overview of the architecture with the aforementioned components and the data flow between them. The data flow starts with an HTTP request for an operation or interacts with a specific resource from the Web UI or an external HTTP client. Then, the Smart Gateway translates the HTTP request to CoAP and transmits it to the objective CoM. Finally, the Smart Gateway responds through HTTP when the CoM responds with the operation's result.

CoM

The CoM is one of the main components in the system, so it serves the resources registered to it and enables the management and deployment of these resources. The CoM is allocated in the microcontrollers deployed. Therefore, the CoM provides the way for end users to install and access resources.

Connecting sensors or actuators in a microcontroller involves connecting them in some connector/s or pin/s. That is, the main idea with which we have designed this approach. Users can connect a sensor or an actuator to connectors in a microcontroller. Once the sensor or the actuator has been correctly connected to the microcontroller, users can register the installed resource and its connectors in order to establish the resource available for interacting with it in the CoM. Therefore, users can easily create a new resource, connecting it to the microcontroller and registering it on the Web UI.

Although some sensors and actuators do not need external libraries to interact with them, such as some analogical sensors, many others do require them. The available libraries for sensors and actuators in microcontrollers could require a large amount of storage. Hence, microcontrollers with storage limitations are far from being supported. To alleviate this, we propose the concept of *objective middleware*. An objective middleware is a middleware conceived for a specific environment, that is, a programmed middleware that supports a set of suitable sensors or actuators in a certain situation. Objective middlewares will form as many groups as there are situations and possible scenarios constituting the IoT, since the system architecture is designed to be easily increased. Furthermore, generic libraries with support for well-known protocols such as I^2C, SPI, or UART can also be created in order to reduce the library size and enable different components to be installed with the same library.

The CoM has been designed to support the creation and association of new sensors and actuators without modifying the source code, so it avoids the human errors that are produced when modifying code and provides an easy way to define new components in the system. To define new components, a new class should extend and implement the virtual methods defined in the CoapSensor class. The CoapSensor class provides the mechanisms to access and manage the sensors and actuators defined for the CoM. A new sensor or actuator can be defined, implementing the virtual methods of the CoapSensor class shown in Code 1.

The init method in the Coap Sensor class is responsible for initializing the necessary components to deploy the desired component based on the connectors' information provided as argument. The get_value and set_value methods enable the query or the actuation in the sensor or the actuator, respectively. The CoM detects the nature of the component implemented (sensor or actuator) based on the methods implemented. Hence, in the case of defining a new sensor, it is only necessary to implement the get_value method, and the set_value method in the case of a new actuator. Finally, the disable method uncouples the components connected.

New libraries can be created dynamically in the CoM through CoAP requests. When a CoM receives a

```
Class CoapSensor

//Initialize the sensor/actuator
Public Subprogram init(name, connectors)
...
End Subprogram

//Set the value input_data received as argument
Public Subprogram set_value(input_data,
    input_data_len, output_data, output_data_len);
...
End Subprogram

// Get the current value of the sensor
Public Subprogram get_value(output_data,
    output_data_len)
...
End Subprogram

// Uncouple the sensor/actuator
Public Subprogram disable()
...
End Subprogram
End Class
```

Code 1: CoapSensor class pseudocode

CoAP POST request with a valid payload for creating libraries, the CoM gets the library type and finds the target library by means of the Registry Software Design Pattern.[26] The Registry Pattern allows the CoM to get references and create classes dynamically through the class name, so it avoids modifying the solution's source code and enables the creation of new classes just by including them in the project solution. Once the libraries have been instantiated in the CoM, their references are saved in the memory and can be updated or removed through CoAP PUT and DELETE requests, respectively. If something fails during an operation (e.g modules activated wrongly), the libraries are not instantiated and a CoAP error response is sent to inform the users of the operation's result. Figure 2 shows the behavior of each CoM when it receives a CoAP request. In order to reduce the flow's complexity, the error cases that can be thrown after each operation have been excluded.

Each CoM has, by default, one resource called lib, which returns the supported sensors and actuators which can be installed in it. The resource lib response replies with the CoRE link format shown in Code 2.[27]

A new link-extension parameter, *a*, has been added to the CoRE link format to enable the supported libraries in each library response. The lib response includes the URI for accessing the resource and the list of available ID types separated by spaces in the *a* parameter. Moreover, the Smart Gateway will add information such as the name and the number of connectors to each supported library when the response is received. The latter also prevents the overload of the CoMs since they only need to send the ID type and the rest of the information is completed by the Smart Gateway.

```
RES: 2.05 Content
</lib>;a="0 1 3 ..."
```

Code 2: lib response example

```
RES: 2.05 Content
</gas>;rt="3";if="core.s";p="5 6 1",
</light>;rt="2";if="core.a";p="12"
```

Code 3: Resource discovery response example

The CoAP resource discovery has also been modified to incorporate additional information from the CoAP resources created, such as the resource type deployed and the connectors used in the installation. A new link-extension parameter *p* is in charge of indicating the connectors used, separated by spaces, and the resource type *rt* attribute has been reused to indicate the type of resource deployed. The *p* parameter is also included in the resource discovery process to inform users in which connectors and sensors are deployed. Hence, a possible response from a CoM with two resources is shown in Code 3:

This response shows the resource discovery of a CoM with two resources: gas and light. The gas resource has been installed in the connectors 5, 6, and 1 with an ID type 3 (CO gas sensor), and it is a sensor resource (core.s, sensor in CoRE link format). The resource light is an actuator (core.a, actuator in CoRE link format) and has been deployed in connector 12 with the ID type 2 (light actuator).

When resources are created in the architecture, they are available for interaction, but when microcontrollers are reloaded or stopped for some reason (power, error, damage, etc.), the changes may be lost. For this reason, the resource persistence also has to be taken into account. We have therefore used the EEPROM (electrically erasable programmable read-only memory) storage for the majority of the microcontrollers to store the information of the resources. Therefore, when CoM starts, first of all it checks whether or not there are indeed resources to be restored in the EEPROM storage. In the case that there are resources in the EEPROM, they are restored based on all the information provided during their creation (name, resource

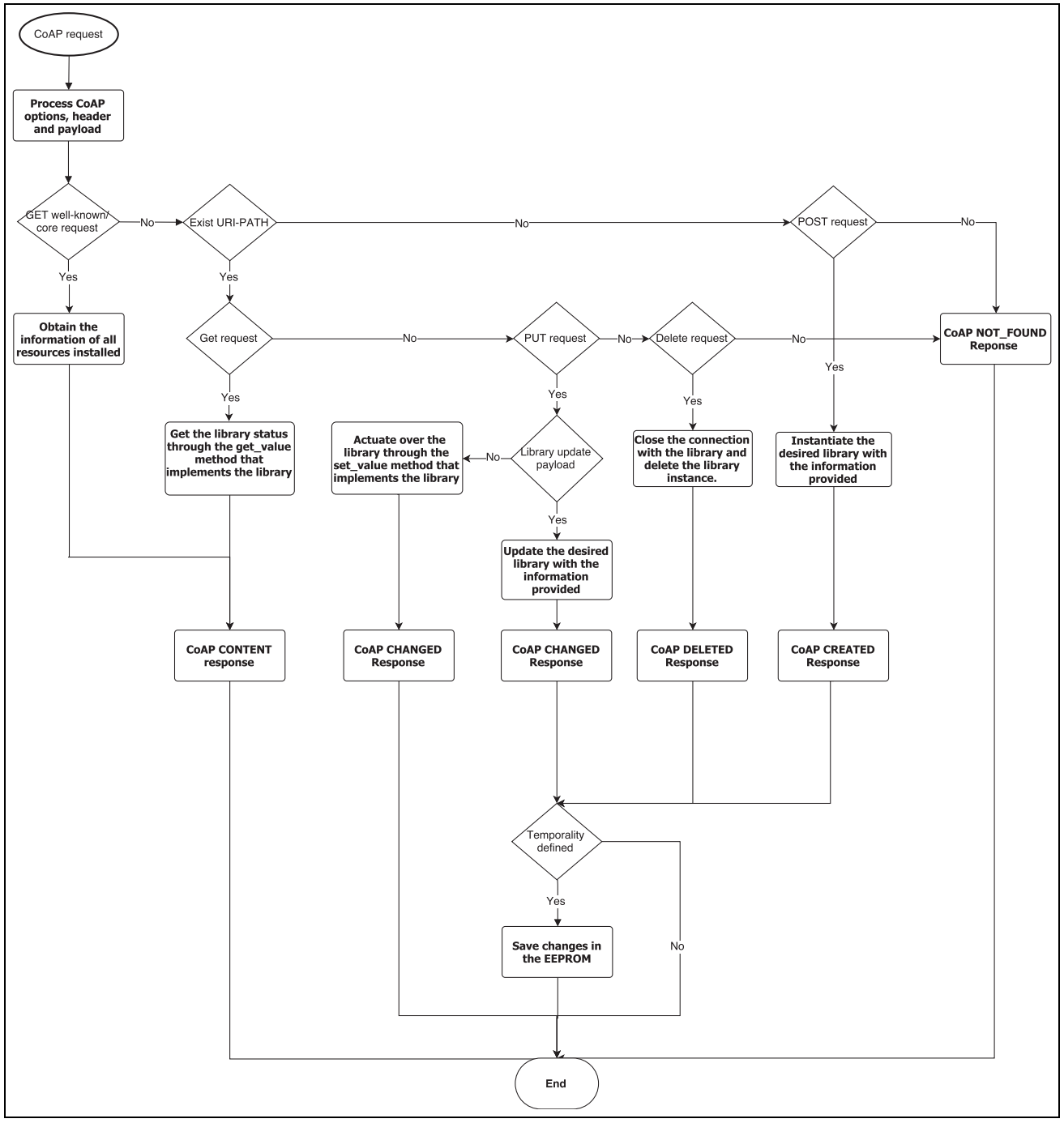

Figure 2. Flow chart with the CoM behavior on CoAP request.

type, and connectors). This avoids the changes being lost when the CoM deviates from its normal behavior, which in turn provides resource persistence.

Smart gateway

The Smart Gateway provides a cross-proxy to convert HTTP requests into CoAP.[28,29] This avoids having to know the format or additional information from the CoAP request. As all the information is gathered in an URI, it sends the format of the CoAP requests which are hidden from the end users and enables an accessible way to communicate with them through well-defined URIs.

In order to carry out tests or simply an operation which is not required in a reload of the CoM, a new concept of *temporality* is introduced into the system. Temporality implies that all changes made in one CoM

Table 1. Smart Gateway API

URI	HTTP method	Description
/nodes	GET	Obtain the CoM list from the Smart Gateway
/CoM/.well-known/core	GET	Obtain the CoM resource list
/CoM/lib	GET	Obtain the available libraries from the CoM
/CoM/res	GET	Obtain the data of the resource res in the CoM
/CoM/res/val	PUT	Actuate over res with the value val
/CoM/res/typ? p=[v1;v2;..]& t=[01]	POST	Create a resource res with type typ, connections contains in *p* and temporality *t* defined
/CoM/res/new_res/ typ?p=[v1;v2;..]& t =[01]	PUT	Update the resource res with a type typ, a new name in new_res, connections contains in *p* and temporality *t* defined
/CoM/res?t=[01]	DELETE	Remove the resource res with the temporality *t* defined

(creations, updations, and deletions) can optionally persist in the CoM depending on the user's needs. Accordingly, users can create, update, or remove a resource in a CoM and decide whether or not to apply these changes throughout its useful life. This also protects the storage in the case of multiple uses, like, for example, testing, because usually storage has a limited number of writings. It has been implemented with a new link-extension parameter *t* which indicates with the value 1 that the operation is temporal, and 0 if not.

All HTTP operations in the proxy follow a well-defined interface shown in Table 1. The API is used by the Web UI to manage the CoAP resources in the CoMs, and it could also be integrated into external applications.

As can be seen in Table 1, the Smart Gateway enables a set of operations for interacting with resources, as well as managing and deploying them. All operations can be done through a URI, so it is not necessary to learn any format or include any payload more than the URI itself. Some operations for managing the resource observations are also included. To ensure the matching between requests and responses, each request sent by the Smart Gateway includes a unique token which is validated with the token of the corresponding response received.

Creating and updating resources requires some information in the CoAP payload. That information includes the connectors' information *p*, the resource type *rt*, its new resource URI, and its temporality *t*. DELETE requests only require the temporal parameter to remove the resource. The translation in Code 4 shows an example of the translation done by the Smart Gateway to temporally update a resource.

Finally, a Database/Cache in the Smart Gateway stores information on the CoM resources, as well as data from its own in some cases. Periodically, with a configurable timeout, the Smart Gateway sends a resource discovery request to the registered CoMs to update its resources. The latter also maintains the contact between the Smart Gateway and the CoMs and can detect possible service interruptions. CoMs are registered with the Smart Gateways when they start up, through a confirmation message. When data from an observable resource are received in the Smart Gateway, it stores the data used in a request for this resource. As a result, the number of requests to the CoMs is reduced and this prevents overload and reduces its power consumption.

```
HTTP PUT /temp/new_uri/0?p=[1;4]&t=1

CoAP PUT /temp
</new_uri>rt="0";p="1 4";t="1"
```

Code 4: Translation between HTTP and CoAP

Web UI

The Web UI enables registered users to straightforwardly access the CoMs and CoAP resources involved. The Web UI also avoids having to use HTTP clients and learn the URI formats for managing and actuating over CoAP resources. Moreover, the responsive design allows a proper visualization on many devices, thus the Web UI can be accessed on smart-phones, PCs, and tablets.

The Web UI is organized on several levels: the associated CoMs, the resources of the CoMs, and the interaction with theirs resources. Once the registered users have accessed the system using their credentials, a form with the associated CoMs in the Smart Gateway is provided. Users can visualize the resources belonging to each CoM by selecting the option of view resources in each CoM. If the option is selected, a form, as shown in

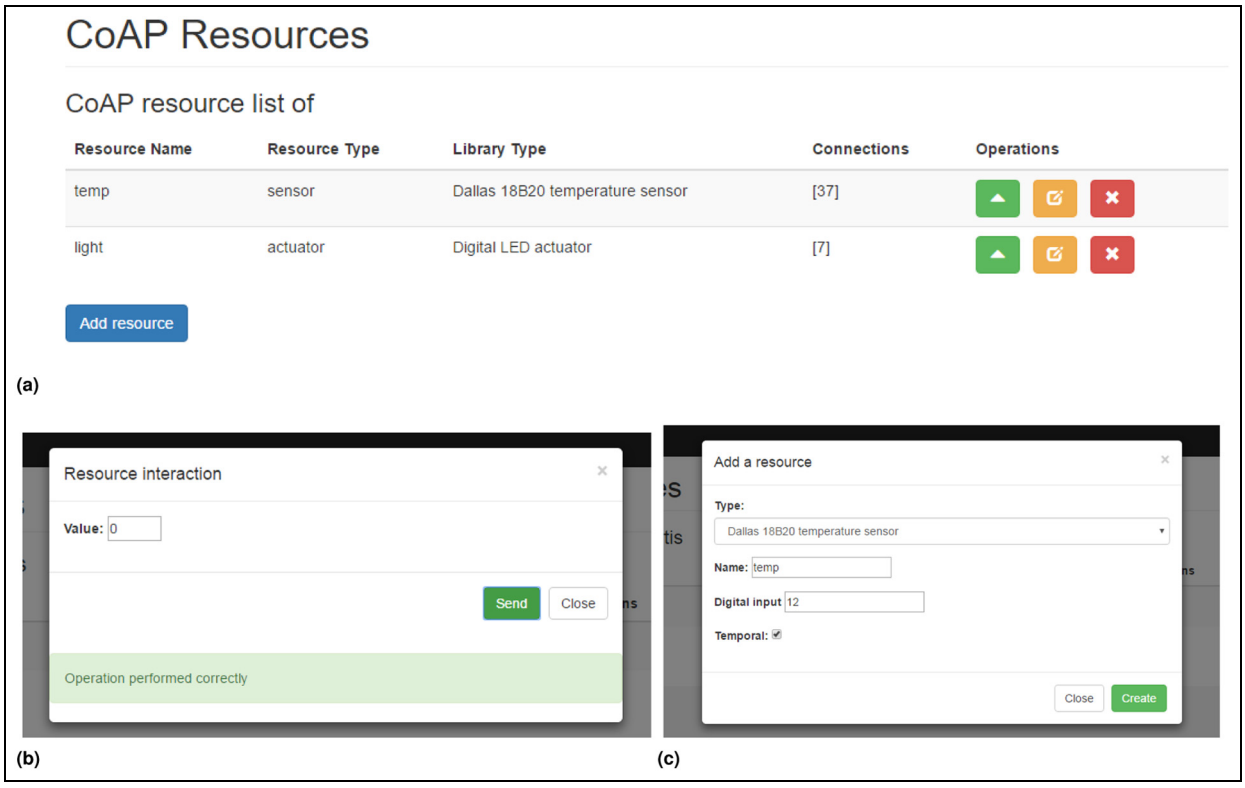

Figure 3. Screenshots of the Web UI: (a) resource list of a CoAP middleware, (b) pop-up window for actuation with a actuator resource, and (c) pop-up window for adding a resource.

Figure 3(a), with the list of resources is provided. This list shows the resources that belong to the selected CoM with its corresponding information such as its resource type, connectors used, and four types of operations—create a new resource, interact, update, or delete one of them—which are available for each of the resources.

The interaction with the resource can differ, depending on the resource type. In the case of resources which contain a sensor, the communication will be established through GET requests, both for HTTP and CoAP. When a user decides to interact with a sensor resource, a GET request is sent to the Smart Gateway and a pop-up window shows the data response received. In the case of resources associated with actuators, the communication is done through PUT requests. The pop-up window in a PUT request (Figure 3(b)) first enables the insertion of a value to control the state of the resource, for example, a behavior in one actuator resource could be to turn it on with a 1 value, turn it off with a 0 value, and create special behaviors with other values. The behaviors will depend on the possibilities of each resource type and the possible values for interacting with them will be indicated to the end users. When the value is inserted and the user confirms the operation, the PUT request is sent to the Smart Gateway and the pop-up window shows the result of the operation when the response is received. These behaviors follow the good practices of RESTFul web services.

Creating resources, Figure 3(c), enables the creation of new resources from the resource types available in the CoM selected. In the resource updates, users can update the access name and connectors of a resource, choosing the temporality of the operation. Equally, users can remove each resource, establishing its temporality.

When an end user decides to create a new resource in the form of list of CoM resources (Figure 3(a)), he or she only has to press the new resource button to create it. As the CoMs can have different objective middleware depending on the scenario, the first step is for the Web UI to send a request to the CoM to obtain its supported libraries. When the Web UI obtains the information requested, a pop-up window to create the new resource is displayed (Figure 3(c)). In the creating resource window, the user can select the resource type, introduce its access name and connector information, and decide whether or not the changes will be temporal. When the user introduces the information and presses the confirm button, the information is sent to the corresponding Smart Gateway. Then, the Smart Gateway translates the request to the CoAP format defined and responds to the Web UI when it receives the response from the CoM. If the resource has been

correctly created, the user will see the resource in the resource list form and interaction with it is allowed from this moment onwards.

Implementation and evaluation

The CoM is based on the implementation of the CoAP library for Arduino with XBee radio support.[30] This implementation follows the 8th version of CoAP and supports the observe option and provides a well-defined abstraction of the CoAP protocol. The CoM has introduced changes in the implementation provided, such as the capacity to manage and deploy resources in runtime in addition to ZigBee communication support and the EEPROM management.

The Smart Gateway receives the HTTP requests through the Python open-source Wheezy Web server. Wheezy Web is a highly concurrent, lightweight, and high-performance Web framework that enables a REST API to access the operations in each Smart Gateway. The CoAP support in the Smart Gateway has been developed from scratch with ZigBee communication. The information contained in the Database/Cache is stored in the memory with access control.

Finally, the Web UI has been designed in two different components through the widely-used back-end/front-end paradigm. The open-source Python Web framework Django has been chosen to allocate the back-end. Django allows Web applications or REST APIs to be created with very little code and very quickly. The open-source JavaScript frameworks AngularJS and Bootstrap have been used for developing the front-end. While AngularJS enables the creation of dynamic Web applications and the back-end interaction, Bootstrap contains a large set of components and utilities for creating responsive Web applications with very little effort. Some information on the Web UI, like the registered users, is contained in an SQLite database.

Smart home case study

The evaluation has been done in one of the main case studies in the IoT, the Smart Home.[31] The system was evaluated in a room where three CoMs connected through batteries were deployed in order to enable the deployment of three different components (a humidity sensor, a temperature sensor, and a light actuator) in each one of them. In the case study, the behavior of the aforementioned components has also been tested with the three operations presented in this article to manage sensors and actuators in run-time: creation, updation, and elimination. Therefore, the Smart Home case study comprises a real IoT system with two IoT sensors (temperature and humidity) and one IoT light actuator.

The Smart Gateway was deployed in a laptop in the same room with a ZigBee connection. Once each CoM had been connected to a battery, its corresponding sensor or actuator was attached to some free connectors in the microcontroller and registered in the CoM as indicated above in Figure 3(c). Once each sensor and actuator had been attached in the CoM and registered in the system, the evaluation was done through an HTTP client which sends requests to the Smart Gateway. In the evaluation, the HTTP client carries out the same operations as those in the Web UI since it also sends requests to the same API in the Smart Gateways. But it was chosen so as to carry out multiple operations in a batch and test the system with a higher stress. Moreover, the Web UI is not the only client that can manage the system, since any other system can be integrated into the proposed system through the Smart Gateways API, so the evaluation also tests the system as it integrates with external applications. Figure 4 shows the case study proposed in the system evaluation.

We have used an Arduino Mega 2560 as the microcontroller for deploying the CoMs. Arduino is part of the open hardware microcontroller board family which is present in a large number of IoT projects due to its reduced price and its large community. A lot of IoT microcontrollers offer support for Arduino such as Libelium[32] and Particle,[33] so the CoM could also be tested in other microcontrollers other than Arduino. Moreover, as the implementation is done in C/C++, it can be easily extrapolated to another platform. Four XBee Pro S2B modules configured with API mode with escaping and a baud rate of 19,200 bps have also been used to create a ZigBee mesh (one coordinator connected to the Smart Gateway and three end devices integrated in Arduino Mega). The Smart Gateway, the Web UI, and the HTTP client have been installed on a Windows 7 OS PC with 8 GB RAM. The following subsections present the memory footprint, data transmission, communication delay, and power consumption evaluations performed.

Memory footprint

The memory footprint is one of the most critical parts of resource-constrained devices, since their available memory is normally highly limited. In order to evaluate the memory consumption when new supported libraries are added to the CoM, an evaluation with three libraries of different sizes has been done. These libraries include an LM35 temperature sensor which does not require external libraries to work, a DHT temperature and humidity sensor with medium-sized external libraries, and a DS18B20 temperature sensor which requires large-sized external libraries.

Figure 5(a) and (b) shows, respectively, the flash and SRAM (static random-access memory) footprints of the uploaded sources to the microcontroller by the Arduino IDE in the aforementioned scenarios. The

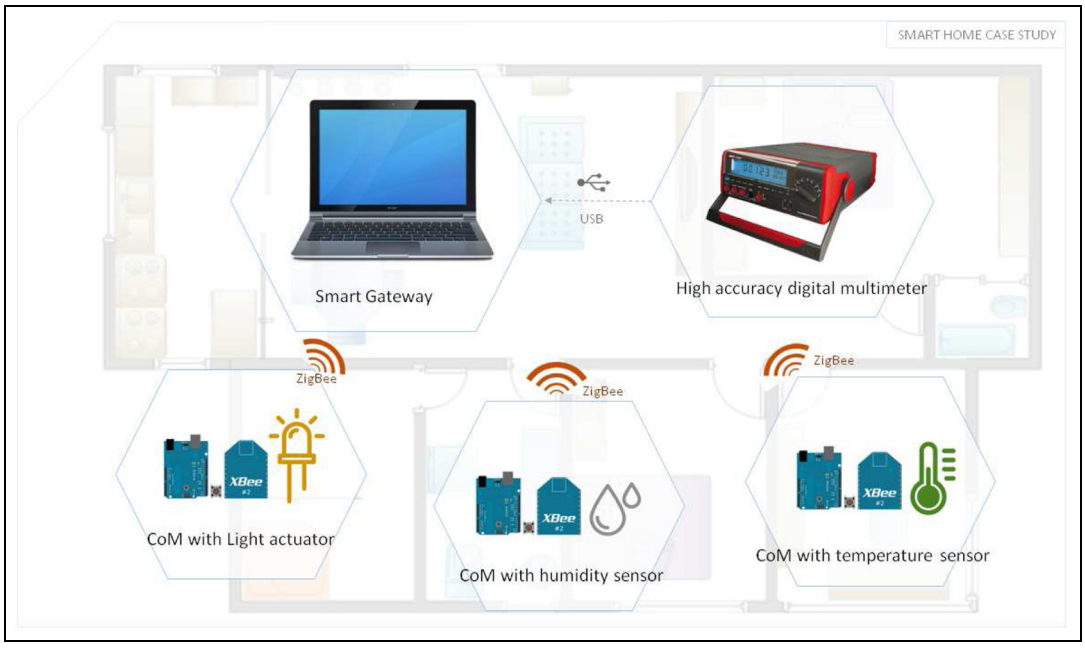

Figure 4. Smart Home case study

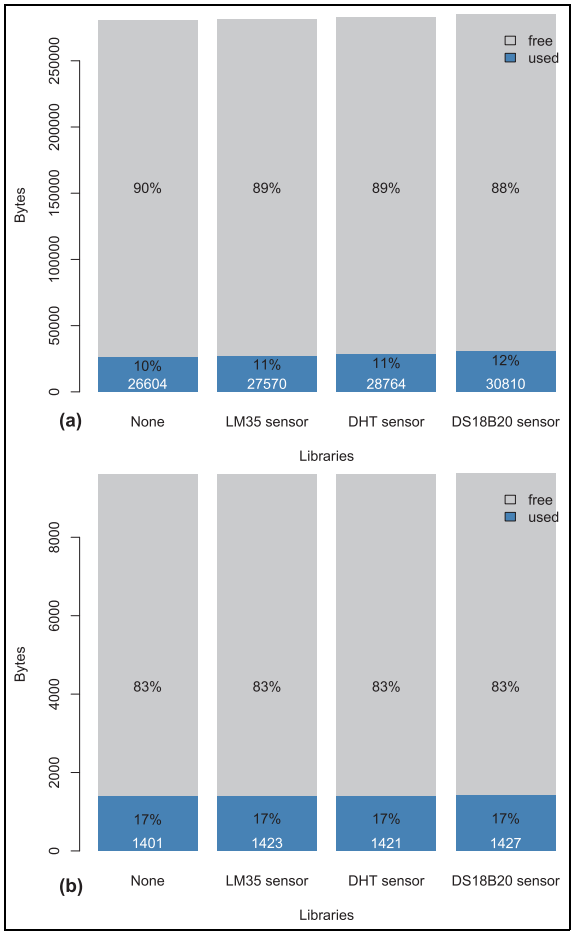

Figure 5. Memory footprint with different library types: (a) Flash-consumption and (b) SRAM consumption.

evaluation demonstrates that the flash footprint in Arduino Mega 2560 varies from 1% to 2% with each library depending on its requirements. The SRAM footprint barely changes with each new supported library. Therefore, the CoM runs comfortably on the Arduino Mega platform and enables support for around 45–90 types of libraries approximately.

Data transmission

The data transmission is another crucial requirement in resource-constrained devices, since the transferred data are restricted most of the time and large data transmissions require high-consumption. In fact, some configurations with ZigBee, with security and in API-enabled mode, can restrict data transmission by as much as 62 bytes. Moreover, the Maximum Transmission Unit (MTU) in IEEE 802.15.4, on which are based protocols such as 6LoWPAN and ZigBee, is established at 127 bytes.

Each data packet exchanged in the run-time CoAP resources control system is composed of the following components: CoAP header, token, URI-Path, and payload. The CoAP Header, as mentioned, is fixed at 4 bytes. Although the token size can vary between 0 and 8 bytes, in the current CoAP version, our implementation uses a fixed token of 2 bytes. One byte to indicate the token option and another to generate 255 different tokens which is sufficient for the proposed scenarios. The URI-Path depends on the URI provided in the Web UI by the users; however, it is recommended that short URIs like '/*sen*' with 4 bytes are used. The

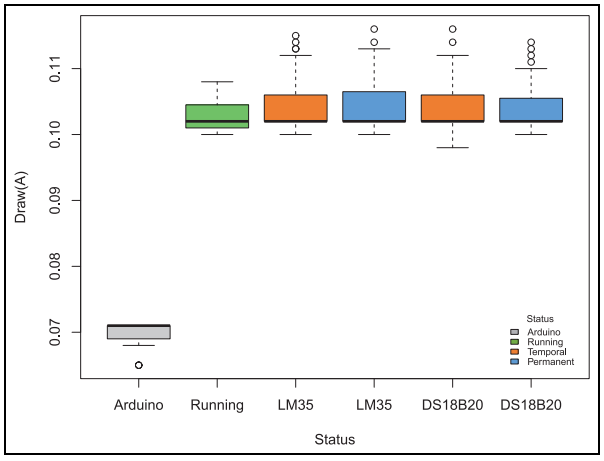

Figure 6. Power consumption with different behaviors

URI-Path is included as an option in the CoAP packet, so a minimum of 1 byte is also added. Finally, the payload size changes with the number of connectors used, the URI-Path and the operation being done (some operations do not require all the parameters). A payload of a creation operation with a URI-Path of 4 bytes, one connector with two bytes and temporally defined (</send>rt = "0"; p = "12"; t = "1") is formed by 24 bytes. Therefore, the packet size of one CoAP operation can be composed as 4 + 2 + 5 + 24 = 35 bytes.

In general terms, the exchanged packet size in the operations offered by the system is fitted into only one data packet and only sent once. Therefore, it can be sent in the most resource-constrained networks and it also avoids other mechanisms such as fragmentation, which overloads the network.

Power consumption

The power consumption has been measured with a high-accuracy digital multimeter which measures the circulating draw provided by the external batteries to the CoMs. In the previous tests, an HTTP client executes a testbed with a set of creations, updations, and deletions with different resource types and behaviors. The power consumption of the Arduino Mega was measured, running the CoM without any operations and with an empty loop in order to contrast the results provided by the microcontroller consumption. The multimeter has a USB interface to upload data to the PC. Figure 6 shows the power consumption results.

The evaluation in Figure 6 shows an average consumption of the CoM of 103 mA. This power consumption can be considered as high, because a normal battery of 2200 mAh will be exhausted in approximately 2200/103 = 21.35 h. This is because the average consumption in Arduino Mega with an empty loop is 70 mA, which is more than two-thirds of the total consumption of the CoM running. The idle current of ZigBee Pro S2B of 15 mA is also included. Due to the high power consumption in Arduino Mega, other low-power microcontrollers like Moteino Mega will be considered in future tests.

The average operation consumption does not vary to any great extent with respect to the CoM consumption. Some outliers between 114 and 116 mA in all cases were found, which could be due to high communication rates, but the overall consumption is constant, close to 104 mA in all scenarios. Therefore, the runtime CoAP resources control system does not cause a high power consumption in the management operations with Arduino Mega.

The total energy consumed by the microcontroller can be estimated using the following formula[34]

$$E = \sum_{j \in \mathcal{M}} i_j . v . \Delta t_j \quad (1)$$

where $\mathcal{M}$ is the set of operation modes of the microcontroller (active, idle, RX, and TX); i_j is the current in state j; v is the nominal voltage of the battery; and Δt_j is the time the microcontroller spent in operation mode j. Specifically, the XBee-PRO S2B radio transceiver used in the case study consumes an average of 54.5 mA in read mode (RX), a peak 212.5mA in writing mode (TX), and 15 mA in active mode. The consumption of the CoM in active mode is 103 mA, including the ZigBee active mode based on our evaluation. Finally, the battery voltage is 7.4 V.

In order to estimate the energy consumption based on the number of requests over time, we make the following assumptions:

- Baud rate obtained in evaluations of 15389.78 bps;
- ZigBee API mode with escaping and no packet loss;
- CoM is always running, idle mode = 0;
- Other requests are ignored;
- 31 and 6 bytes for the CoAP requests and responses to the CoM, respectively;
- 1-year simulation.

Figure 7 presents a simulation of the energy consumption with different operation rates over a year. Although, in order to work for long periods, microcontrollers have to sleep to reduce the power consumption, the results show an exponential energy decrement with respect to the operation rate. However, the energy consumption remains high due to the microcontroller's consumption.

Communication delay

In order to measure the communication delay with the CoMs, a testbed with different types of physical sensors

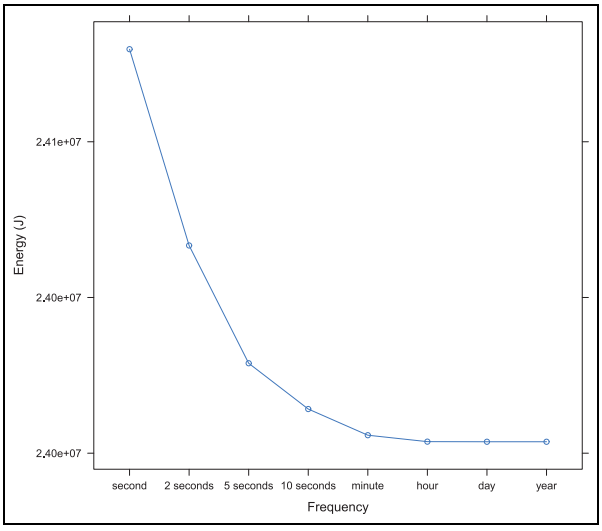

Figure 7. Energy simulation with several operation rates over a year.

and behaviors was also designed. In each operation, the HTTP and CoAP response time of the creation, updation, and elimination operations from an HTTP client to the Smart Gateway was measured 10 times. The results are represented as box plot graphs in Figure 8 which shows the median, outliers, and quartiles of each operation. LM35 and DS18B20 temperature sensors were chosen to evaluate the response time with sensors with different requirements. The temporal behavior was also evaluated for each sensor type.

Figure 8(a) shows the CoAP response time in the management operations in the LM35 and DS18B20 sensors, respectively. The creation and updation operations differ from each other between 4 and 5 cs. This is due to the delay in the instantiation of the libraries needed to register the DS18B20 sensor. Therefore, different sizes in the management operations only imply the consumption of some centiseconds. However, the deletion operation does not require the instantiation of

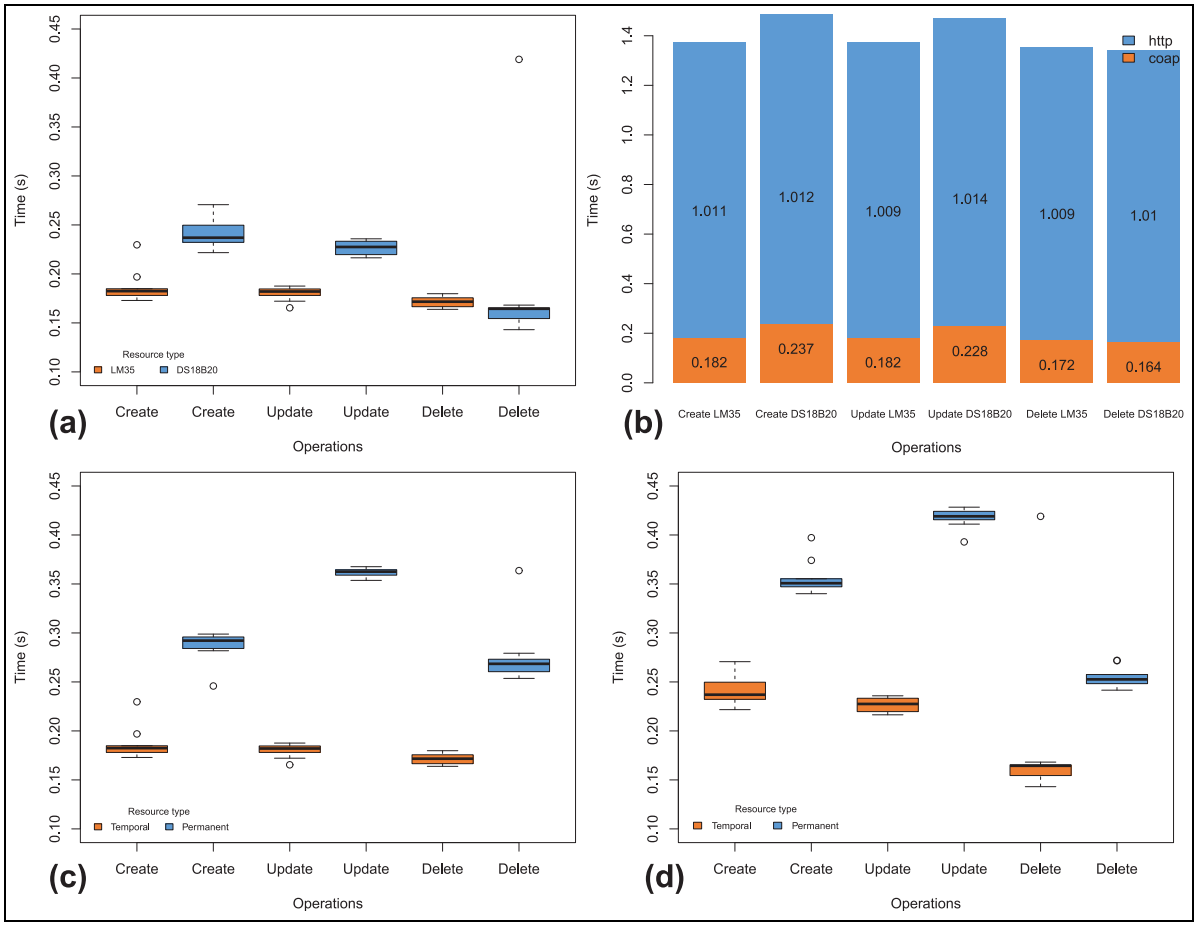

Figure 8. Communication delay measures: (a) CoAP response time with different operations, (b) median response time with different operations, (c) CoAP response time in LM35 with different temporality, and (d) CoAP response time in DS18B20 with different temporality.

new libraries, so the delay is smaller. In fact, the results show a slightly better result in the DS18B20 deletion.

In Figure 8(b), the median response time of HTTP and CoAP in each operation is shown. The main purpose of HTTP is to translate the HTTP request into CoAP, so resource types and behaviors do not affect it. This is the reason why the HTTP time only changes in milliseconds between operations and its time is not representative beyond measuring the overall response time in each operation. Nevertheless, the results reflect that the management operations can be done through the Web UI or an HTTP client in little more than 1 s in the deployment scenario, irrespective of the sensor type used.

As can be seen in Figure 8(c) and (d), the temporality affects the CoAP response time, to a greater extent, in both types of sensors. This is because when temporality is defined, changes are persisted in the EEPROM memory and the storage delay is present. Thus, the temporality not only protects the EEPROM memory but also reduces the response time in the management operations.

Comparison with other proposals

As mentioned in the "Related work" section, most current systems deploy sensors and actuators in compile time, or else, provide mechanisms to deploy them in run-time using standards suitable for computers instead of microcontrollers such as USB or IEEE 1451. The latter entails a difficult task to achieve a comparison with other systems in the same scenarios due to the anatomy of the plug-and-play components, which are not suitable for resource-constrained devices, and vice versa. Another solution could consist of modifying the compile-time deployment systems in order to provide the run-time deployment functionality; however, this goes beyond the scope of the work presented here.

To the best of our knowledge, only μPnP provides a system that allows the deployment of sensors and actuators in run-time for resource-constrained devices. However, μPnP requires specific software and hardware components to compose the system architecture and they are not accessible to make a system comparison. Based on the evaluation provided in the μPnP paper,[15] the system proposed in this article provides a lower RAM and a higher Flash consumption. In the Smart Home case study, the Flash consumption is not a drawback since there is about 90% free space with some libraries. The data transmission time shows an average time with the proposed scenario similar to our average time with the sensors and actuators tested and the temporality defined. The HTTP time is not included in the last comparison, but it enables the system to be managed with a large variety of clients, for example, the Web UI. Finally, in terms of energy consumption, the proposed system provides a higher energy consumption; however, the proposed system has been tested in a real scenario while the μPnP presents a simulation with ideal behaviors.

Conclusion and future work

In this article, a system for run-time managing and deploying physical resources in microcontrollers has been presented. The goal of this solution is to reduce the gap between end users and customizable environments. Hereafter, end users can install their own sensors and actuators on their microcontrollers through well-defined interfaces. CoAP and ZigBee have been used as the embedded web service and communication means allocated in the microcontrollers, thereby enabling a lightweight and low-power approach for interacting with resources. The system provides a proxy to interconnect HTTP with CoAP in a Smart Gateway, in such a way that any HTTP client can carry out the operations provided by the Smart Gateway. Moreover, a Web UI has been incorporated to provide multi-device access to end users. Storage and temporal limitations have been taken into account in the design, so that the system provides objective middlewares and temporal behaviors.

In future work, we will ensure the compatibility of the system with different communication means for resource-constrained devices in order to evaluate them and establish the most suitable means in each environment. Updating the CoAP version to the current one and improving the URI and CoAP security are also included in our roadmap.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work has been funded by the Spanish projects TIC-1572 ("MIsTIca: Critical Infrastructures Monitoring based on Wireless Technologies") and TIN2014-52034-R ("An MDE Framework for the Design and integration of Critical Infrastructure Management Systems").

References

1. Gubbi J, Buyya R, Marusic S, et al. Internet of Things (IoT): a vision, architectural elements, and future directions. *Future Gener Comp Syst* 2013; 29(7): 1645–1660.
2. Díaz M, Martín C and Rubio B. State-of-the-art, challenges, and open issues in the integration of internet of

things and cloud computing. *J Netw Comput Appl* 2016; 67: 99–117.
3. Al-Fuqaha A, Guizani M, Mohammadi M, et al. Internet of Things: a survey on enabling technologies, protocols, and applications. *IEEE Commun Surv Tut* 2015; 17(4): 2347–2376.
4. Miorandi D, Sicari S, De Pellegrini F, et al. Internet of Things: vision, applications and research challenges. *Ad Hoc Netw* 2012; 10(7): 1497–1516.
5. Medaglia CM and Serbanati A. An overview of privacy and security issues in the Internet of Things. In: Giusto D, Iera A, Morabito G, et al. (eds) *The Internet of Things*. New York: Springer, 2010, pp.389–395.
6. Shelby Z, Hartke K and Bormann C. The constrained application protocol (coap), https://tools.ietf.org/html/rfc7252/ (accessed 13 May 2016).
7. ZigBee 3.0: the foundation for the Internet of Things, http://www.zigbee.org/ (accessed 13 March 2017).
8. 11 Internet of Things (IoT) protocols you need to know about, http://www.rs-online.com/designspark/electronics/knowledge-item/eleven-internet-of-things-iot-protocols-you-need-to-know-about (accessed 4 May 2016).
9. Díaz M, Martín C and Rubio B. λ-coap: An internet of things and cloud computing integration based on the Lambda architecture and coap. In: Guo S, Liao X, Liu F, et al. (eds) *Collaborative computing: networking, applications, and worksharing*. Cham: Springer, 2016, pp.195–206.
10. Weiser M. Connecting the physical world with pervasive networks. *IEEE Pervas Comput* 2002; 1: 59–69.
11. Tsiatsis V, Gluhak A, Bauge T, et al. The sensei real world internet architecture. In: Tselentis G, Galis A, Gavras A, et al. (eds) *Towards the future Internet*. Amsterdam: IOS Press, 2010, pp.247–256.
12. Openiot. https://github.com/OpenIotOrg/openiot (accessed 17 May 2015).
13. Aloulou H, Mokhtari M, Tiberghien T, et al. A semantic plug& play based framework for ambient assisted living. In: Donnelly M, Paggetti C, Nugent C, et al. (eds) *Impact analysis of solutions for chronic disease prevention and management*. New York: Springer, 2012, pp.165–172.
14. Bröring A, Maué P, Janowicz K, et al. Semantically-enabled sensor plug & play for the sensor web. *Sensors* 2011; 11(8): 7568–7605.
15. Yang F, Matthys N, Bachiller R, et al. μPnP: plug and play peripherals for the Internet of Things. In: *Proceedings of the tenth European conference on computer systems*, Bordeaux, 21–24 April 2015, p.25. New York: ACM.
16. Perera C, Zaslavsky A, Christen P, et al. Connecting mobile things to global sensor network middleware using system-generated wrappers. In: *Proceedings of the eleventh ACM international workshop on data engineering for wireless and mobile access*, Scottsdale, AZ, 20 May 2012, pp.23–30. New York: ACM.
17. Sleman A and Moeller R. Integration of wireless sensor network services into other home and industrial networks; using device profile for web services (DPWS). In: *Proceedings of the 3rd international conference on information and communication technologies: from theory to applications Damascus*, Syria, 7–11 April 2008, pp.1–5. New York: IEEE.
18. Dunkels A and Yazar D. Efficient application integration in IP-based sensor networks. In: *Proceedings of the first ACM workshop on embedded sensing systems for energy-efficiency in buildings*, Berkeley, CA, 3 November 2009, pp.43–48. New York: ACM.
19. Levä T, Mazhelis O and Suomi H. Comparing the cost-efficiency of coap and http in web of things applications. *Decis Support Syst* 2014; 63: 23–38.
20. Riedel T, Fantana N, Genaid A, et al. Using web service gateways and code generation for sustainable IoT system development. In: *Proceedings of the Internet of Things (IOT)*, Tokyo, Japan, 29 November–1 December 2010, pp.1–8. New York: IEEE.
21. Ruckebusch P, De Poorter E, Fortuna C, et al. Gitar: generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Netw* 2016; 36: 127–151.
22. Taherkordi A, Loiret F, Rouvoy R, et al. Optimizing sensor network reprogramming via in situ reconfigurable components. *ACM Trans Sens Netw* 2013; 9(2): 14.
23. Zaslavsky AB, Perera C and Georgakopoulos D. Sensing as a service and big data. In: *Proceedings of the international conference on advances in cloud computing (ACC-2012)*, Bangalore, India, 26–28 July 2012, pp.21–29. New York: ACM.
24. Aazam M, Khan I, Alsaffar AA, et al. Cloud of things: integrating internet of things and cloud computing and the issues involved. In: *Proceedings of the 11th international Bhurban conference on applied sciences and technology (IBCAST)*, Anchorage, AL, 14–18 January 2014, pp.414–419. New York: IEEE.
25. Marz N and Warren J. *Big data: principles and best practices of scalable realtime data systems*. Greenwich, CT: Manning Publications, 2015.
26. Fowler M. *Patterns of enterprise application architecture*. Boston, MA: Addison-Wesley, 2002.
27. Constrained RESTful environments (CoRE) link format, https://tools.ietf.org/html/rfc6690 (accessed 9 May 2016).
28. Ludovici A and Calveras A. A proxy design to leverage the interconnection of coap wireless sensor networks with web applications. *Sensors* 2015; 15(1): 1217–1244.
29. Colitti W, Steenhaut K, De Caro N, et al. REST enabled wireless sensor networks for seamless integration with web applications. In: *Proceedings of the 8th international conference on mobile adhoc and sensor systems (MASS)*, Valencia, Spain, 17–22 October 2011, pp.867–872. New York: IEEE.
30. Basic coap library for arduino, https://github.com/dgiannakop/Arduino-CoAP (accessed 12 May 2016).
31. The 10 most popular internet of things applications right now, http://iot-analytics.com/10-internet-of-things-applications/ (accessed 20 December 2015).
32. Libelium, http://www.libelium.com/ (accessed 16 November 2016).
33. Particle, https://www.particle.io/ (accessed 20 December 2015).
34. Cirani S, Picone M, Gonizzi P, et al. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. *IEEE Sens J* 2015; 15(2): 1224–1234.