Wireless Pers Commun
DOI 10.1007/s11277-015-3168-6

Design and Evaluation of a Services Interface for the Internet of Things

Caio César de Melo Silva[1] · Hiro Gabriel Cerqueira Ferreira[1] · Rafael Timóteo de Sousa Júnior[1] · Fábio Buiati[1] · Luis Javier García Villalba[2]

Abstract This paper proposes an application programming interface (API) for accessing services within the internet of things (IoT) through both REST and SOAP protocols. This API provides methods and procedures to allow its usage for performing IoT control and event monitoring operations. In order to encompass devices diversity, the proposed API introduces a uniform abstraction model that constitutes a common standard view to manage objects. An abstract device services interface is then available instead of device commands, thus providing transparent access to devices capabilities and hiding the physical aspects of provider devices. The API has been designed to ease the remote management of IoT smart objects and was implemented as a module of an existing IoT middleware (UIoT). Experimental evaluation of both protocol implementations yields results showing the REST services with faster response time and lower resources usage than similar SOAP services.

✉ Luis Javier García Villalba
javiergv@fdi.ucm.es

Caio César de Melo Silva
caio.silva@redes.unb.br

Hiro Gabriel Cerqueira Ferreira
hiro.ferreira@gmail.com

Rafael Timóteo de Sousa Júnior
desousa@unb.br

Fábio Buiati
fabio.buiati@redes.unb.br

[1] Electrical Engineering Department (ENE), University of Brasilia (UnB), Campus Darcy Ribeiro - Asa Norte, Brasília, DF 70910-900, Brazil

[2] Group of Analysis, Security and Systems (GASS), Department of Software Engineering and Artificial Intelligence (DISIA), Faculty of Information Technology and Computer Science, Office 431, Universidad Complutense de Madrid (UCM), Calle Profesor José García Santesmases, 9, Ciudad Universitaria, 28040 Madrid, Spain

Published online: 12 January 2016

Keywords Internet of things (IoT) · IoT application programming interface (IoT API) · IoT middleware · REST services · SOAP services · UPnP devices

1 Introduction

The internet of things (IoT) concept evolved from its original sense, associated with RFID applications [1], to generically include all application comprising objects or devices that can interact with other objects and applications over the internet. Computing is going through its third large Phase, the ubiquitous era [2] which is characterized by the existence of multiple processors/devices acting to provide seamless services to humans. A forecast by [3] asserts that more than 1 trillion devices will be connected to the internet by 2020.

Considering the IoT potential number of objects and their heterogeneity, it is important to develop and deploy IoT application programming interfaces (IoT API) to permit straightforward access to devices resources and operations. Indeed it can be argued that IoT magnitude and longevity can get compromised if well designed APIs are not available.

This work proposes an IoT API, which is designed according to the Universal Plug and Play (UPnP) [4] and web services access standards, thus allowing its development both by means of the REpresentational State Transfer (REST) architecture and the Simple Object Access Protocol (SOAP). The proposed API considers IoT usage scenarios presented in a previous work [5] to specify the necessary services that are designed and developed to provide uniform access to resources and operations, despite IoT devices diversity. Our design is oriented to optimize computational resources utilization to allow dynamic IoT device controlling through a simple querying system. The resulting RESTful and SOAP style implementations are then experimentally evaluated.

This paper is organized as follows. Section 2 discusses related works and points out this paper contributions. Section 3 introduces concepts and references to technologies related to this work. Section 4 explains the components of the REST/SOAP API for IoT services. Section 5 presents the specification of services and processing flow within the REST/SOAP API for IoT services. Section 6 explains results collected using the API in real scenario deployments and tests. Finally, Sect. 7 presents our conclusions and brings future works to discussion.

2 Related Work

It is important to clarify the IoT service definition adopted in this paper, since this concept is the motivational factor of the research work presented here. As discussed in [6] there are several definitions and still no consensus on the subject. Therefore, this paper uses the IoT service definition found in [6], which states: "An IoT-Service is a transaction between two parties, the service provider and the service consumer. It causes a prescribed function enabling the interaction with the physical world by measuring the state of entities or by initiating actions which will cause a change to the entities."

As shown in [7], a unique identification of objects, and the representation, and storing of exchanged information is the most challenging issue for IoT middlewares. One of the

possible solutions to overcome these challenges is to create a service layer in the middleware that provides historical information about devices and actions on an IoT environment. However, most research describe the provision of services in IoT based on general concepts or on the needs of a specific application, and not through a well-established pattern.

Gronbaek [8, 9] defines an API that decouples service logic from protocols and network elements. It defines the following service elements: micro-payment, storage and retrieval, presentation service, session service, transport service, event reporting, connected objects presence, location and status, mobility, QoS control, data transmission, security association creation and basic IP bearer. However, primitives are defined to manage network nodes in specific groups. In addition, [8] does not define a standard set of requests to manage devices and actions.

Authors in [10] present a broad view of the internet of things: from the data acquisition layer at the bottom to the application layer at the top. Specifically, the article discusses the design of a SOA-based architecture for a IoT middleware with two service components: service composition and service management. The article describes that it is extremely important for provider and requester to communicate robustly despite the heterogeneous nature that surrounds the foundations of IoT. But, this research does not define standards for creating and using services in IoT middlewares.

Also, in [11] a service-oriented architecture is proposed for IoT. The architecture is divided in four layers named: sensing, networking, service and interface. The service layer includes four components:

- *service discovery*: find objects which can provide services
- *service composition*: enables the interaction and communication among connected things.
- *trustworthiness management*: aims to determine trust levels for information provided by other services.
- *service APIs*: supports the interactions between services.

The research elucidates several aspects of service layers design in IoT middlewares. But, again, it does not provide a standardization of requests for Connected Objects management.

Kim et al. [12] present WoO, a semantic ontology model representing devices, resources and services. This work is based on several research works which address the management of devices and services through ontological models (SENSEI: [13], IoT-A: [14], iCore: [15], BUTLER: [16]). The research work also defines the components that should be present in an IoT service domain model, very similar to the UPnP definition. In addition, it is shown an example of a service deployed on the WoO platform. However, the work does not define a pattern of communication between the service layer and the various layers of an IoT architecture.

Noting in this overview the need for a generic and standardized definition for accessing IoT management services and conforming to established standards and technologies, the present paper presents an API that seeks to provide information on devices and actions in an IoT network. The API is based on the UPnP entity model, as well as the REST and SOAP approaches to communications and the standard data formats JavaScript Object Notation (JSON) and Extensible Markup Language (XML).

3 Related Concepts

This section comprises a brief overview of concepts related to IoT architectures, including the UPnP specification, and REST/SOAP services, in order to characterize the technologies and standards that support our proposed IoT API.

3.1 IoT Architectures

Different proposals for the structure and functioning of the IoT have been published, including possible components and their interactions, thus constituting alternative architectures for IoT networks. Reference [17] presents a broad overview on the topics of IoT middleware and architecture initiatives as well as general ideas for IoT research and development.

Existent IoT Architectures common elements include device representation rules and functions for device controlling and monitoring. They are important to allow encompassing legacy and newly developed devices, also integrating existing smart devices to IoT networks.

The work presented is this paper comprises prototyping the proposed IoT API using an architecture associated to the UIoT Middleware and Communication Model [18], since this middleware allows new modules to be added in a transparent way, because its open source software is built upon widespread technology and standards, such as UPnP, and allows the implementation of different services interfaces for the implemented modules and operations.

3.2 UPnP

Universal Plug and Play [4] is a set of networking protocols that permits networked devices to automatically establish working configurations with other devices, including the discovery of each others presence on the network and the setting of services that each device provides for interoperating with other devices and applications.

Indeed, UPnP is a TCP/IP based protocol stack designed as a standard for universal transparent access to logical objects resources whose operations comprise 6 main phases, namely Addressing, Discovery, Description, Control, Eventing and Presentation. Also, UPnP proposes a precise Device Architecture for the abstraction of devices that is used to perform these operations throughout the Simple Service Discovery Protocol (SSDP), the Simple Object Access Protocol (SOAP) and the General Event Notification Architecture (GENA).

3.3 REST Services

REST is defined in [19] as an coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations.

The key abstraction in REST is the resource, which is a conceptual mapping to a set of entities. Anything that can be named can be a resource and the semantics associated to a resource is required to be static, though entities behind a resource may change over time [20].

As described in [20] REST comprises three types of architectural elements: data types, connecting elements, and processing elements. In REST, the architecture components exchange information transferring representations of the current or desired state of data elements. These transfers use the four existing HTTP message types and access available services by referencing the corresponding URLs linked to these services. Thus, REST provides a simple straightforward interface to access web services. Responses from these services can come in a variety of forms, such as Command Separated Value (CSV), JavaScript Object Notation (JSON) and Really Simple Syndication (RSS), so the output is ready to be directly parsed within the service consumer application.

3.4 SOAP Services

SOAP is an XML-based protocol for messaging and remote procedure calls (RPCs). Rather than define a new transport protocol, SOAP works on existing transfer protocols, such as HTTP, SMTP, and MQSeries. The SOAP specification defines a model that dictates how recipients should process SOAP messages. The message model also includes actors, which indicate who should process the message. A message can identify actors that indicate a series of intermediaries that process the message parts meant for them and pass on the rest [21].

As a means to cope with possible complexities in the XML used to make requests and receive responses, in SOAP the Web Services Description Language (WSDL) is used to provide a definition of how the Web service works, thus reducing the effort required to create the request and to parse the response.

4 Components of the REST/SOAP API for IoT Services

This section presents the structure of our proposed REST/SOAP API for IoT services. It is organized as follows: Sect. 4.1 presents the logical abstraction of devices, the base for all provided services. Section 4.2 presents the API abstract model, conceptualizing its layers, modules and components. Afterward, Sect. 5 explains the information flow throughout the API, specifically presenting all REST and SOAP routes (for device discovery, device controlling and subscription for state changes) and the API abstract model components involved in each request.

4.1 API Device Architecture

Our proposed API has a precise manner to logically abstract IoT devices in order to ensure uniformity in accessing resources. This feature, which is called API Device Architecture (ADA), is structured by inheritance from the UPnP Device Architecture [22]. Thus, ADA is composed by UPnP sub-entities (devices, services, actions, state variables and arguments) and, additionally, introduces a sixth entity, the named controller. Figure 1 shows the relations among components of the API Device Architecture.

In UPnP a device provides services to users. A service is delivered through actions. Actions have parameters, called arguments, to guide how a device state should be changed. A service has state variables which are modified by the execution of actions and are represented in interactions as input and output arguments. For example, considering a smart refrigerator abstraction according to this architecture, the representation would have

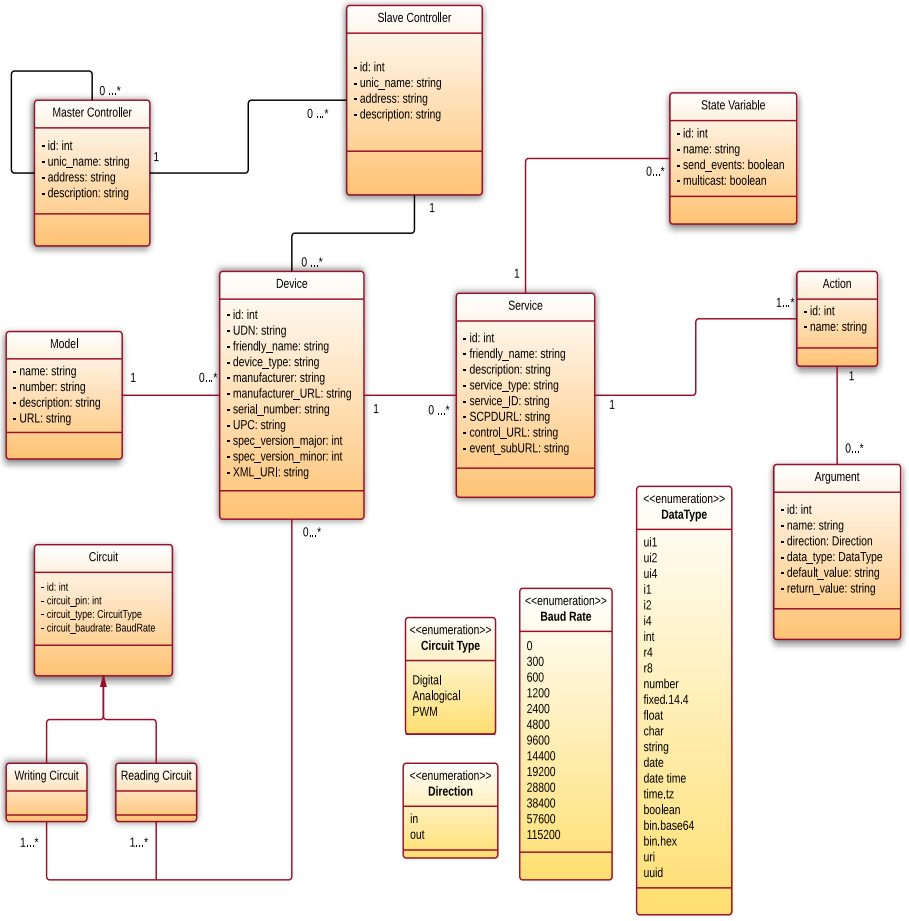


Fig. 1 API device architecture (ADA)

the refrigerator as a device that could have two services: “control refrigerator’s temperature” and “control refrigerator’s power”. The service “control refrigerator’s temperature” could have three actions: “increase temperature”, “reduce temperature” and “set temperature to a specified value”. The action “set temperature to a specified value” could have two input arguments: “new temperature value in Celsius degrees” and “time to wait before changing temperature”. The input argument “new temperature value in Celsius degrees” would change the service state variable “temperature”. Action “set temperature to a specified value” could have as output argument the “value the temperature was set to”, which would hold the value of the state variable “temperature” after the action has been performed.

To help encompassing legacy plain objects, turned into smart object through the mediation of micro-controllers, ADA introduces a new entity extending the UPnP Device Architecture: the controller. To attain better resource usage, it is assumed that a controller can hold more than one device. Even though the UPnP Device Architecture originally supports devices embedded in a root device, a feature that could solve this issue, we believe that it is more didactic, formal and clear to distinguish the case of multi-functional

physical devices, such as a printer with scanner within the same physical device, from multiple physically separated devices managed by a micro-controller, such as a raspberry pi processor controlling all appliances in a room. Still considering the case of a smart refrigerator, it could be associated to a controller named "kitchen controller" which would have some devices attached to it, such as "Kitchen light", "Coffee pot" and "Refrigerator".

As a solution to cope with the diversity of IoT devices, ADA brings uniformity to resources access since, with its sub-entities and their relationships, ADA allows the API to have well defined methods to search, control and monitor the state of all encompassed devices.

4.2 API Abstract Model

The API initial design, shown in Fig. 2, was intended to increase the scalability on IoT configurations. In this sense, since both REST and SOAP can exchange information over

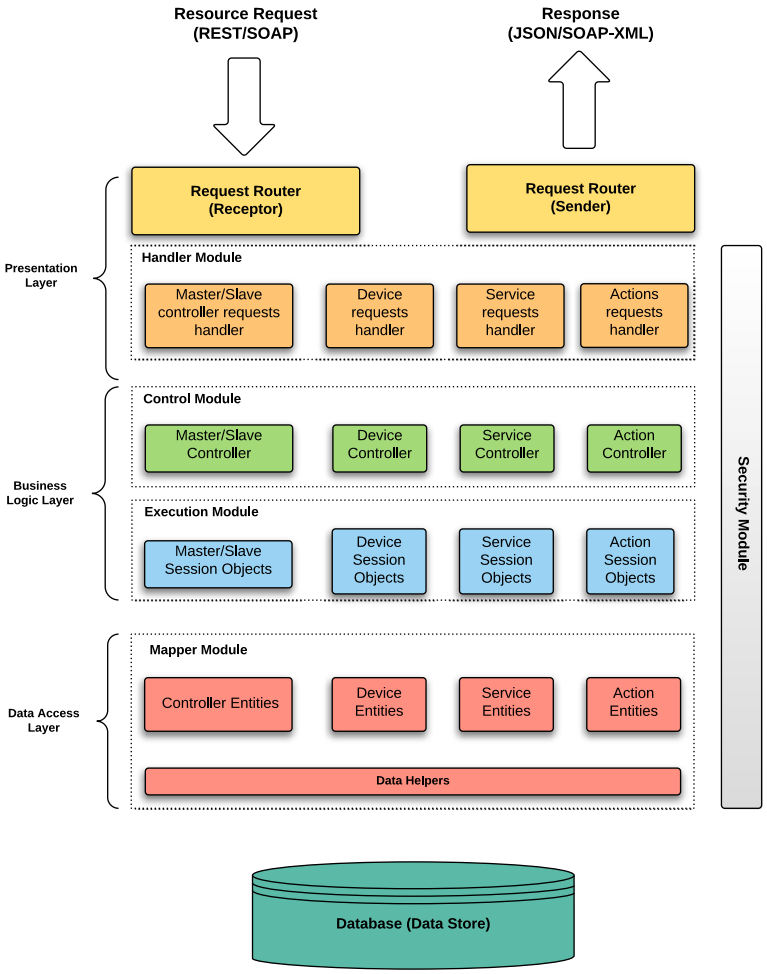


Fig. 2 IoT API architecture: layers, modules and components

HTTP, this choice of a transfer protocol results in loose coupling between client and server, a characteristic that contributes to the desired scalability [23]. This choice reinforces the principle of modeling the API to uniformly handle IoT objects, be them smart objects or legacy ones, by means of the device architecture specification presented in Sect. 4.1. Moreover, the API is designed to integrate a security module which has components in all API layers, providing protection to the processed information from the very moment it enters the API.

Requests submitted to the API are handled by all of its layer modules. Each layer has components that are dedicated to manipulate and provide specific information about each entity presented on ADA. The requests router is assigned to transparently redirect the client request to the appropriate component. Therefore, the API allows a separate management of each entity presented in IoT architectures, through a uniform communication standard.

The presentation, business logic and data access layers were designed according to traditional web systems standards [24] although they present some fundamental differences. For instance, the web presentation layer generally consists of components that provide a common bridge into the core business logic encapsulated in the business layer, and usually manages user interaction with the system. In our design, the presentation layer is also a bridge to the business logic layer, but the system does not deal with user interactions. Instead, it allows client applications to send various requests for smart objects in an IoT architecture. In the following subsections, the characteristics and features of our API layers and their respective components are presented.

4.2.1 Presentation Layer

The presentation layer (PL) contains the components that implement and display the client requests interface and manage client solicitations, throughout both REST and SOAP. This layer includes components for controlling client demands, as well as for routing the submitted requests, being composed by two main logical modules referred as request router (RR) and handler module (HM), as shown in Fig. 2.

The main function of the RR module is to redirect client requests to the HM components. The request router collects all the information in the REST or SOAP request, creates an object populated with the collected parameters, and send this object to the component which handles the destination entity. For example, if the request is "GET `controllers/0125/devices/`", then the RR will created an instance of a route object, and send it to a *Master/Slave controller requests handler* component.

The handler module receives a route object sent by the RR and sends a request to the business layer. Specifically, an HM component (request handler) sends the type of the REST/SOAP request (GET, POST, PUT, DELETE), the route object identifier and the parameters of the request to the corresponding component in the control module, this last one being responsible for searching the requested session object in the execution module. It is noteworthy that the handler module also validates the submitted route object, a security feature that is explained in detail in Sect. 5.4.

4.2.2 Business Logic Layer

The business logic layer (BLL) implements the core functionality of the API, and encapsulates the relevant business logic. BLL comprises two modules, namely control and execution, which jointly manage active smart objects on the IoT infrastructure. It is important to point out that, unlike traditional web services, the API business logic layer

does not expose service interfaces, which are only accessible through the presentation layer.

The BLL control module (CM) comprises controller components which can send commands to change the state of session objects, as well as to retrieve information from these objects. CM may also communicate with the data access layer if the requested information can not be delivered by any session object.

The BLL execution module (EM) manages the session objects currently being used on the IoT network. Session objects can be defined as instances of UPnP entities that store information needed for a particular client service. The main advantage of using session objects is to retrieve information about the devices and services without the need to consult the database, allowing faster access to the requested information. Thus, each EM component responds to the requests made by the control module. If some information is not available at the execution module, a request is sent to the data access layer, and a new session object is created.

4.2.3 Data Access Layer

The data access layer (DAL) provides access to data hosted within the system. DAL provides interfaces that can be used by business layer components. The data access layer has only one module, referred to as mapper module (MM), whose role is to map all devices and services that can be requested by a customer. Thus, the entities present in this module represent all smart objects that can be used to compose an IoT network, and which might be within an IoT middleware.

This module also comprises a data helper component which abstracts the logic required to access the underlying data stores. It centralizes common data access functions in order to make the system easier to configure and maintain. The objective of performing this mapping by means of data helpers is to retrieve requested data from any sort of data store (such as traditional databases, NoSQL databases, text, XML and JSON files, etc.) and convert them into a local well known and defined model object. Figure 2 represents model objects as *Entities* inside the MM block, which are converted to session objects when requested.

5 Specification of Services and Processing Flow Within the REST/SOAP API for IoT Services

Based on the described API logical representation of smart objects, and its internal modular structure, this section is aimed at specifying the API allowed routes for REST and SOAP requests and the corresponding process flow. Our design was modeled to support seventeen request processing flows, which are summarized in Table 1. Fourteen of these requests are destined to retrieve information about ADA entities, as presented in Sect. 5.1. One of the remaining flows is meant to issue control commands to devices, as presented in Sect. 5.2, while the other two are destined to allow the subscription of applications for receiving notifications of new values assumed by devices state variables. This subscription allow applications getting real time access to devices' states in a PUSH style system whose Subscription flows are presented in Sect. 5.3. We present details of the REST approach, since REST requires the direct specification of URI, and respective methods and

Table 1 Requests supported by the IoT API

Method	Resource URI	Expected result
GET	/controllers/	Array of all existent controllers
GET	/controllers/ unique_name/	Object with information of a controller whose name equals the URI parameter unique_name
GET	/controllers/ unique_name/ devices/	Array of devices associated to a controller indicated by the URI parameter unique_name
GET	/devices/	Array of all existent devices
GET	/devices/device_id/	Object with information of devices identified by the URI parameter device_id
GET	/devices/device_id/ services/	Array of existent services associated to device identified by the URI parameter device_id
GET	/services/	Array with object of existent services
GET	/services/service_id/	Object with information of a service identified by the URI parameter service_id
GET	/services/service_id/ actions/	Array of actions associated to the service identified by the URI parameter service_id
GET	/services/service_id/ state_variables/	Array of state variables associated to the service identified by the URI parameter service_id
GET	/actions/	Array with objects of existent actions
GET	/actions/action_id/	Object with data related to action and its arguments where action id equals to parameter action_id of URI
GET	/state_variable	Array of existent state variables
GET	/state_variable/ state_variable_id/	Object with data related to a state variable identified by the URI parameter state_variabel_id
PUT	/actions/action_id/	Perform the action identified by the URI parameter action_id, on the correct device
POST	/services/service_id/	Add a new notification URL for a service identified by the URI parameter service_id
DELETE	/services/service_id/	Remove an existent notification URL for a service identified by the URI parameter service_id

parameters, while SOAP needs this same information embedded in XML envelops. Finally, Sect. 5.4 describes how a security module operates within the API layers.

5.1 REST and SOAP Discovery Requests

Discovery Requests are meant to allow applications to learn about IoT network capacities and to allow the applications to search for specific devices or services. Using this type of request it is possible to collect data regarding states of a device and to acquire knowlegde about parameters that should be sent to control such device through the Control Requests presented in Sect. 5.2. The API allows dynamic searches for device types and, as a separated feature, it permit searches for service types that can perform some action independently of which specific device is effectively to perform the desired action. This last feature is interesting because it allows an application to monitor and perform actions without knowing the entire IoT network.

When a Discovery Request is submitted to the API, a common internal flow is followed to obtain the right response. When an application request reaches the RR, it builds a RouteObject and send it to the HM, which in turn converts the request to a internal known query object (Controller, Device, Service or Action) and sends it to the CM. This last searches for desired objects on MM, querying specific databases through Data helpers, to find the concerned Entities. As a result, the MM returns the Found Entities to CM, which will ensure that that these Entities are Session Objects, converting them otherwise. Then, the CM sends a Session Objects pointer back to the HM. The HM will build Response Objects containing the Session Objects pointed by CM and send the result back to the requesting application.

It is noteworthy to show the representation used for all REST and SOAP requests. Figure 3 shows a generic REST Request, which is composed by a required title, a required method(GET, POST, PUT and DELETE), a required request URI (/requested/resource/uri/), optional URI parameters (?parameter_1=value1&parameter_2=value2...), optional Request object, with its parameters and sent value, and optional Response Object with its parameters and returned value.

In the requests, a list of parameters can be specified by the array operand "[]" thus permiting queries applying the OR operator to these parameters. As an example, the list of parameters "*?name[]=light&name[]=room*" would return objects containing the attribute *name* with value equal to *light* or to *room*.

Also in the requests, a wildcard represented by the % symbol can be placed within the sent parameter value to permit queries applying the AND operator. This symbol is a substitute for zero or more characters in the sent value. In URLs, the exact representation of % character is 25 % but, for the sake of simplicity in this paper, it is represented just as % in expressions of URLs and URIs.

The wildcard is a flexible means for getting combined results from requests, such as: parameter value "*&name=%room*" would return objects containing the attribute *name* ending with the word *room*, "*&name=kitchen%*" would return objects containing the attribute *name* starting with the word *kitchen*, "*&name=%car%*" would return objects containing the attribute *name* with the word *car* present in any position (beginning, middle or ending), "*&name=light%pole*" would return objects containing the attribute *name* starting with the word *light* and ending with the word *pole*, "*&name=%coffee%pot*" would return objects containing the attribute *name* having the word *coffee* in any position and ending with the word *pot*, "*&name=%energy%efficient%*" would return objects containing the attribute *name* with the word *energy* followed by the word *efficient*.

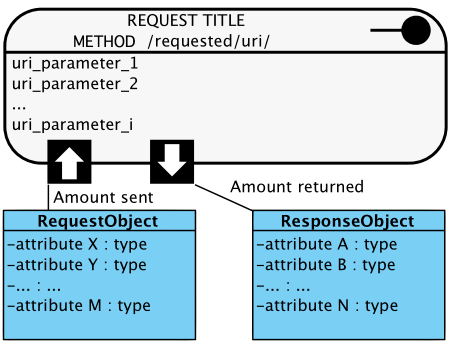


Fig. 3 REST request representation

It is worth to notice the necessary API feature for performing the same operations throughout SOAP. In order to perform SOAP remote procedure calls for the requests presented above, SOAP envelopes are used as described in [21]. More specifically, XML is used in SOAP requests both to represent the procedure call and the answer, as illustrated in Fig. 4. For a better visualization this figure only show parts of the XML files, which indicate the actions to be performed, though in the IoT API the header file conveys more information such as, for example, the sender's credentials and correlation information. In these SOAP requests, HTTP headers encompass the *SOAP:Envelope* element, specifically using the *HTTP POST* message, which browsers also use to submit forms. The POST header is followed by an optional *SOAPAction* header that indicates the message intended purpose. If there is a synchronous response, the HTTP response type is *text/xml*, as declared in the *Content-Type* header and may contain a SOAP message with the response data. Alternatively, the recipient could deliver the response message later (asynchronously) [21].

Given the described Discovery Request Flow, together with the REST and SOAP Request Representation and URI parameters formatting, we present and explain the purpose of each different request included in our IoT API design. The general strategy of an application for using the IoT API comprises the following expected sequence of steps: to learn about an IoT network capacities, an application start querying about the available controllers, then get information about devices associated to a subset of controllers, then get services provided by these devices, then get data related to actions provided by such services and, finally, read which arguments are needed for an action to be performed and which arguments this action will return after have being performed. Hence, we hereafter specify each corresponding API request, presenting possible usage scenarios that elucidate and justify the choices in our design.

Figure 5 shows the route "GET `/controllers/`" used to read data about controllers available in the IoT Network. It returns an array of Controller Objects, containing the

SOAP CALL

```
POST /requested/uri/
SOAPAction: "http://www.uiot.org/ACTION_NAME/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m: SEND_MESSAGE
   xmlns:m="http://www.uiot.org/ACTION_NAME"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <uri_param_1
xsi:type="xsd:long">value1</uri_param_1>
        <uri_param_2
xsi:type="xsd:long">value2</uri_param_2>
        <uri_param_3
xsi:type="xsd:long">value3</uri_param_3>

      </m:SEND_MESSAGE>
   </SOAP:Body>
</SOAP:Envelope>
```

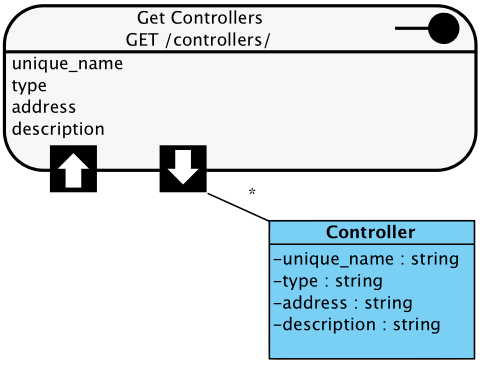
SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:RESPONSE_MESSAGE
         xmlns:m="http://www.uiot.org/ACTION_NAME"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <responseContent
            type="xsd:type">RESPONSE_VALUE
         </responseContent>
      </m:RESPONSE_MESSAGE>
   </SOAP:Body>
</SOAP:Envelope>
```

Fig. 4 Example of a SOAP request and response

attributes unique name, type, address and description of existing Controllers Session Objects. To allow specific queries, the URI parameters *unique_name*, *type*, *address*, and *description* can be sent, optionally using the wildcard % and/or array [] operators, with the desired query values. The API is then able to return the customized lists of matching controllers. For instance, a request such as "GET `/controllers/?type=%zigbee%`" would return an array of all controllers containing *zigbee* as their type, hence, put in other words, the requesting application can have access to all zigbee enabled controllers within the IoT Network.

Figure 6 shows the route "GET `/controllers/unique_name/`" used to read all data about one specific controller available in the IoT Network. It returns a Controller



(a)

SOAP CALL

```
POST /controllers
SOAPAction: "http://www.uiot.org/controllers/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetControllersInfo

   xmlns:m="http://www.uiot.org/controllersinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <uniqueName
xsi:type="xsd:string">U.N.</<uniqueName>
         <type xsi:type="xsd:string">TYPE</<type>
         <address
xsi:type="xsd:string">ADDRESS</address>
         <description
xsi:type="xsd:string">DESC</description>
      </m:GetControllersInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

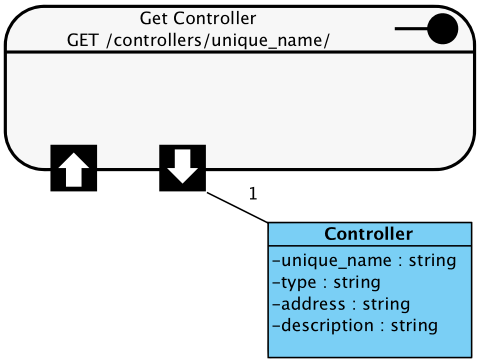
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetControllersInfoResponse

   xmlns:m="http://www.uiot.org/controllersinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <controllersInfo

   type="controllerxsd:controllerArray">CONTROLLER[].
         </controllersInfo>
      </m:GetControllersInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 5 Discovery request get controllers. **a** REST get controllers. **b** SOAP get controllers

Object (with attributes unique name, type, address and description) representing the Session Object corresponding to the *unique_name* specified in the URI. For instance, the request "GET `/controllers/garage/`" returns a JSON representation of the *Master/Slave Session Object* that has a *unique_name* equal to *garage*. The *Get Controller* request does not allow any URI parameter.

Figure 7 shows the route "GET `/controllers/unique_name/devices/`" used to read all data about devices pertaining to one specific controller available on the IoT Network. It returns an array of device Objects (with attributes id, friendly name, device type and encompassing controller id) of existing Device Session Objects belonging to the controller specified in the URI *unique name*. To allow specific queries, URI parameters *friendly_name* and *device_type* can be sent, optionally using the wildcard % and/or array []



(a)

SOAP CALL

```
POST /controller/unique_name/
SOAPAction: "http://www.uiot.org/ctrl_info/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetControllerInfo

    xmlns:m="http://www.uiot.org/controllerinfo"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
           xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
           <uniqueName
xsi:type="xsd:string">U.N.</<uniqueName>
       </m:GetControllerInfo>
    </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

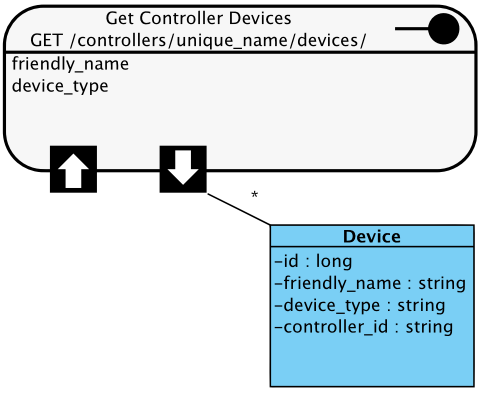
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetControllerInfoResponse

    xmlns:m="http://www.uiot.org/controllersinfo"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <controllerInfo

    type="controllerxsd:controller">CONTROLLER
           </controllerInfo>
       </m:GetControllerInfoResponse>
    </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 6 Discovery request get controller. **a** REST get controller. **b** SOAP get controller

operators, with the desired query values. For instance, "GET `/controllers/children_room/devices/?device_type[]=%lighting%`" would return an array of all devices, encompassed by the controller *children_room* that has *lighting* as its type, it is possible to access all light related devices inside the controller of a children's room (Fig. 7).

Figure 8 shows the route "GET `/devices/`" used to read all data about devices available in the IoT Network, including those linked and not linked to controllers. It returns an array of device Objects (with attributes id, friendly name, device type and encompassing controller id) of existing Device Session Objects. To allow specific queries, URI parameters *friendly_name* and *device_type* can be sent, optionally combined by wildcard % and/



(a)

SOAP CALL

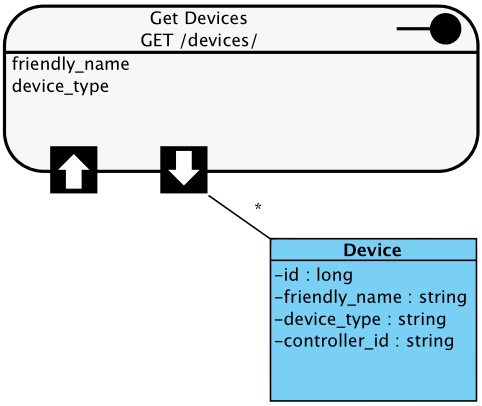
```
POST /controllers/unique_name/devices/
SOAPAction: "http://www.uiot.org/ctrl_devices/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetControllerDevices

   xmlns:m="http://www.uiot.org/controllerinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <uniqueName
xsi:type="xsd:string">U.N.</<uniqueName>
         <deviceType
xsi:type="xsd:string">D.T.</deviceType>
         <friendlyName
xsi:type="xsd:string">F.N.</<friendlyName>
      </m:GetControllerDevices>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetControllerDevicesResponse

   xmlns:m="http://www.uiot.org/devicesinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <devicesInfo
            type="devicexsd:device">DEVICES[]
         </devicesInfo>
      </m:GetControllerDevicesResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 7 Discovery request get controller devices. **a** REST get controller devices. **b** SOAP get controller devices



(a)

SOAP CALL

```
POST /devices
SOAPAction: "http://www.uiot.org/devices/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetDevicesInfo

   xmlns:m="http://www.uiot.org/devicesinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <friendlyName
xsi:type="xsd:string">F.N.</friendlyName>
         <deviceType
xsi:type="xsd:string">D.T.</deviceType>
      </m:GetDevicesInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

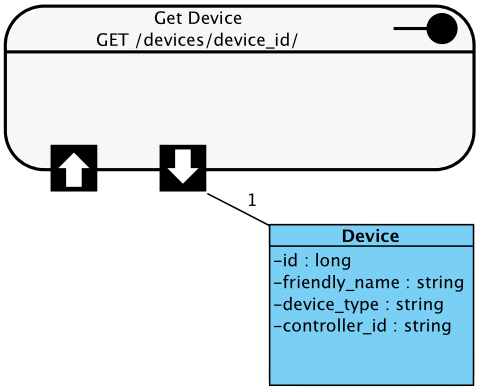
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetDevicesInfoResponse

   xmlns:m="http://www.uiot.org/devicesinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <devicesInfo
            type="devicexsd:device">DEVICE[]
         </devicesInfo>
      </m:GetDevicesInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 8 Discovery request get devices. **a** REST get devices. **b** SOAP get devices

or array [] operators. For instance, "GET `/devices/?device_type[]=%temperature%`" would return an array of all devices containing *temperature* as their type, i.e., this request refers to all temperature related devices in the IoT Network.

Figure 9 shows the route "GET `/devices/device_id/`" used to read all data about one specific device available in the IoT Network, including those linked and not linked to controllers. It returns a Device Object (with attributes id, friendly name, device type and encompassing controller id) representing the Session Object that has the URI attribute *device_id*. A request "GET `/device/32/`" returns a JSON representation of the *Device Session Object* whose *id* equals *32*. The *Get Device* request does not allow any URI parameter.



(a)

SOAP CALL

```
POST /device/device_id/
SOAPAction: "http://www.uiot.org/dev_id/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetDeviceInfo

   xmlns:m="http://www.uiot.org/deviceinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <deviceID
xsi:type="xsd:long">DevID</<deviceID>
      </m:GetDeviceInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

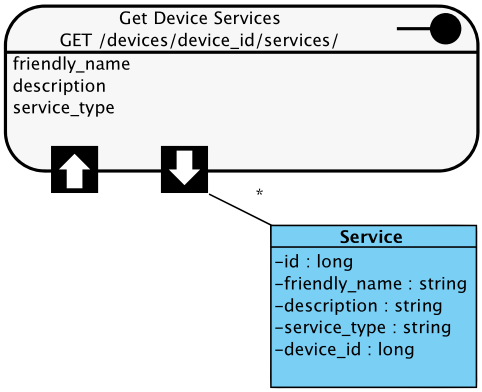
SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetDeviceInfoResponse
            xmlns:m="http://www.uiot.org/deviceinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <deviceInfo
            type="devicexsd:device">DEVICE
         </deviceInfo>
      </m:GetDeviceInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 9 Discovery request get device. **a** REST get device. **b** SOAP get device

Figure 10 shows the route "GET `/devices/device_id/services/`" used to read all data about services within one specific device available in the IoT Network, including those linked and not linked to controllers. It returns an array of Service Objects (with attributes id, friendly name, description, service type and service provider as device id) of existing Service Session Objects belonging to the device with the URI *device_id*. URI parameters *friendly_name*, *description*, and *service_type* can be optionally combined using wildcard % and/or array [] operators. For instance, the request "GET `/devices/16/services/?service_type[]=%humidity%manager%`" would return an array of all services, within the device with the attribute *id* equal to *16*, containing its



(a)

SOAP CALL

```
POST /device/device_id/services/
SOAPAction: "http://www.uiot.org/dev_services/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetDeviceServices

    xmlns:m="http://www.uiot.org/devservices"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
           xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
          <friendlyName
xsi:type="xsd:string">F.N.</friendlyName>
          <serviceType
xsi:type="xsd:string">S.T.</serviceType>
          <description
xsi:type="xsd:string">description</<description>
        </m:GetDeviceServices>
    </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

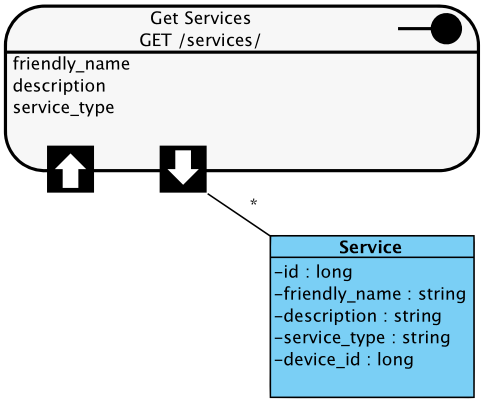
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetDeviceServicesResponse

    xmlns:m="http://www.uiot.org/devservices"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <deviceServices
                type="servicexsd:service">SERVICE[]
           </deviceServices>
       </m:GetDeviceServicesResponse >
    </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 10 Discovery request get device services. **a** REST get device services. **b** SOAP get device services

attribute type with word*humidity* followed by *manager*. In other words, the request gives access to all services of device 16, say a greenhouse's humidity handler, to handle the greenhouse humidity.

Figure 11 shows the route "GET `/services/`" used to read all data about services provided by devices in the IoT Network. It returns an array of service Objects (with attributes id, friendly name, description, service type and service provider as device id) of existing Service Session Objects. URI parameters *friendly_name*, *description* and *service_type* can be combined as described before. For instance, "GET `/services/?service_type[]=%library%access%`" would return an array of all services containing *library* followed by *access* in its type. This request accesses all services present



(a)

SOAP CALL

```
POST /services/
SOAPAction: "http://www.uiot.org/services/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServicesInfo

   xmlns:m="http://www.uiot.org/servicesinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
          <friendlyName
xsi:type="xsd:string">F.N.</friendlyName>
          <serviceType
xsi:type="xsd:string">S.T.</serviceType>
          <description
xsi:type="xsd:string">description</<description>
       </m:GetServicesInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

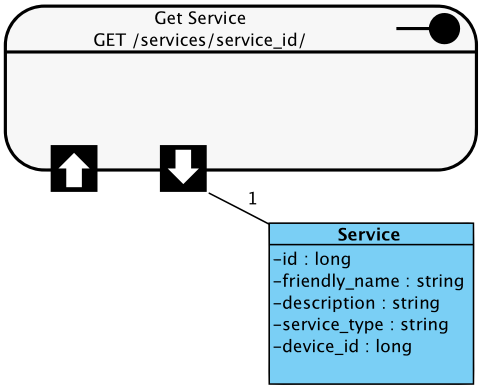
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServicesInfoResponse

   xmlns:m="http://www.uiot.org/servicesinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <servicesInfo
                type="servicexsd:service">SERVICE[]
          </servicesInfo>
      </m:GetServicesInfoResponse >
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 11 Discovery request get services. **a** REST get services. **b** SOAP get services

in the IoT Network that provide access to the library, independently of which device will issue the final action.

Figure 12 shows the route "GET `/services/service_id/`" used to read all data about one specific service available in the IoT Network. It returns a service Object (with attributes id, friendly name, description, service type and service provider as device id) representing the Service Session Object corresponding to the URI *service_id*. The request "GET `/services/64/`" returns a JSON representation of the *Service Session Object* with the attribute *id* equal to *64*. The *Get Service* request does not allow any URI parameter.



(a)

SOAP CALL

```
POST /services/service_id
SOAPAction: "http://www.uiot.org/serv_info/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServiceInfo

   xmlns:m="http://www.uiot.org/serviceinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <serviceID
xsi:type="xsd:long">F,N,</serviceID>
      </m:GetServiceInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

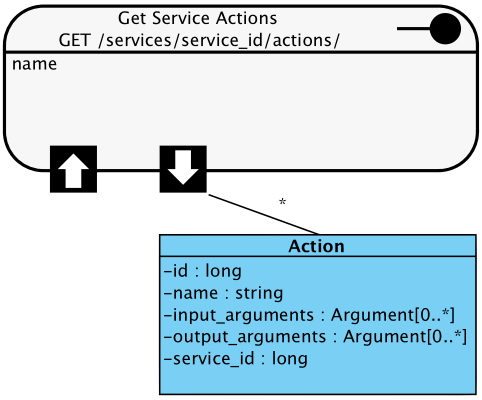
SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServiceInfoResponse

   xmlns:m="http://www.uiot.org/serviceinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <serviceInfo
            type="servicexsd:service">SERVICE
         </serviceInfo>
      </m:GetServiceInfoResponse >
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 12 Discovery request get service. **a** REST get service. **b** SOAP get service

Figure 13 shows the route "GET `/services/service_id/actions/`" used to read all data about executable actions within a specific service available in the IoT Network, independently of which device performs the service action. It returns an array of action Objects (with attributes id, name, input arguments array, output arguments array and provider service id) of existing Action Session Objects belonging to the service specified in the URI *service_id*. This URI parameter can combine matching criteria using the wildcard `%` and/or array [] operators. For instance, "GET `/services/128/actions/?name=%set%power%`" returns an array of all actions, within the service with attribute *id* equal to *128*, containing the word*set* followed by *power* in their name attribute. This way,



(a)

SOAP CALL

```
POST /services/service_id/actions
SOAPAction: "http://www.uiot.org/serv_actions/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetServiceActions

    xmlns:m="http://www.uiot.org/serviceactions"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
           xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
            <actionName
xsi:type="xsd:string">A.N.</actionName>
        </m:GetServiceActions>
    </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

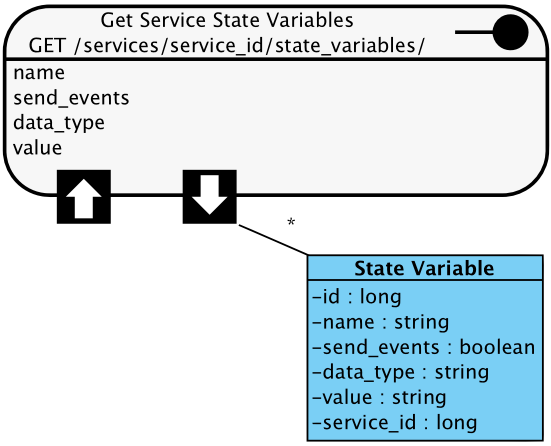
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetServiceActionsResponse

    xmlns:m="http://www.uiot.org/serviceactions"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <serviceActions
                type="actionxsd:action">ACTION[]
           </serviceActions>
       </m:GetServiceActionsResponse>
    </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 13 Discovery request get service actions. **a** REST get service actions. **b** SOAP get service actions

an application has access to all actions of service 128, say a kitchen coffee pot handler power service, to change the power state of the related coffee pot.

Figure 14 shows the route "GET `/services/service_id/state_variables/`" used to read data about state variables maintained by a specific service available in the IoT Network, independently of which device actually has such variables. It returns an array of state variable Objects (with attributes id, name, send events, data type, value and service id it belongs to) of the concerned State Variables specified in the URI. This URI parameter can combine matching criteria using the wildcard % and/or array [] operators. For instance, "GET `/services/128/state_variables/?name=%power`" returns an array of all state variables, within the service with attribute *id*



(a)

SOAP CALL

```
POST /services/service_id/state_variables
SOAPAction: "http://www.uiot.org/serv_state_var/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServiceStateVariables

   xmlns:m="http://www.uiot.org/servicestatevar"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
          xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <stateVarName
xsi:type="xsd:string">S.V.N.</stateVarName>
        <sendEvents
xsi:type="xsd:boolean">S.E.</sendEvents>
        <dataType
xsi:type="xsd:string">D.T.</dataTyoe>
        <value xsi:type="xsd:string">value</value>
      </m:GetServiceStateVariables>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

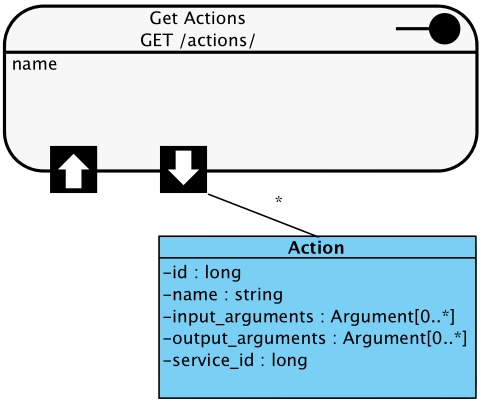
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetServiceStateVariablesResponse

   xmlns:m="http://www.uiot.org/servicestatevar"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <servStateVar
                type="statevarxsd:statevar">SV[]
          </servStateVar>
      </m:GetServiceStateVariablesResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 14 Discovery request get service state variables. **a** REST get service state variables. **b** SOAP get service state variables

equal to *128*, containing the word*power* in the name attribute of state variables. This request allows access to the power state of service 128, say a kitchen coffee pot handler power service, with its coffee pot power state variables.

Figure 15 shows the route "GET `/actions/`" used to read all data about actions provided by services in the IoT Network. It returns an array of action Objects (with attributes id, name, input arguments array, output arguments array and provider service id) of existing Action Session Objects. The URI *name* parameter can combine matching



(a)

SOAP CALL

```
POST /actions
SOAPAction: "http://www.uiot.org/actions/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetActionsInfo

   xmlns:m="http://www.uiot.org/actionsinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <actionName
xsi:type="xsd:string">A.N.</actionName>
      </m:GetActionsInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

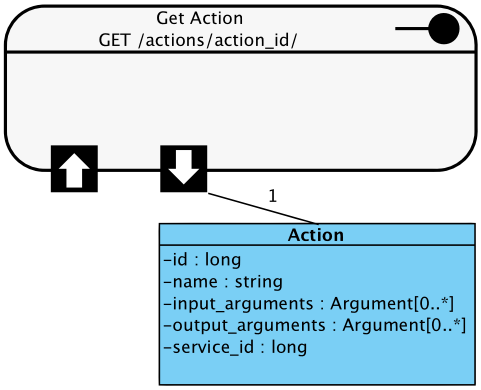
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetActionsInfoResponse

   xmlns:m="http://www.uiot.org/actionsinfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <actionsInfo
            type="actionxsd:action">ACTION[]
         </actionsInfo>
      </m:GetActionsInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 15 Discovery request get actions. **a** REST get actions. **b** SOAP get actions

criteria using the wildcard % and/or array [] operators. For instance, "GET `/actions/?name[]=%start%motor%`" returns an array of all actions containing *start* followed by *motor* in their name, thus allowing access to all actions that can start a motor in the IoT Network.

Figure 16 shows the route "GET `/actions/action_id/`" used to read data about one specific action available in the IoT Network. It returns an action Object (with attributes id, name, input arguments array, output arguments array and provider service id) representing the Action Session Object specified in the URI *action_id*. The request "GET /



(a)

SOAP CALL

```
POST /actions/action_id/
SOAPAction: "http://www.uiot.org/actioninfo/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetActionInfo

   xmlns:m="http://www.uiot.org/actioninfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <actionID
xsi:type="xsd:string">A.Id</actionID>
      </m:GetActionInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

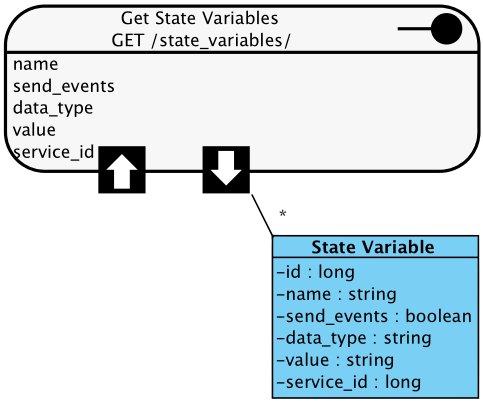
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetActionInfoResponse

   xmlns:m="http://www.uiot.org/actioninfo"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <actionInfo
            type="actionxsd:action">ACTION
         </actionInfo>
      </m:GetActionInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 16 Discovery request get action. **a** REST get action. **b** SOAP get action

`actions/256/`" returns a JSON representation of the *Action Session Object 256*. The *Get Action* request does not allow any URI parameter.

Figure 17 shows the route "GET `/state_variables/`" used to read data about state variables maintained by services available in the IoT Network, independently of which device actually has such state variables. It returns an array of state variable Objects (with attributes id, name, send events, data type, value and service id it belongs to) of existing State Variables within Services Session Objects. The URI *name*, *send_events*, *data_type*, *value* and *service_id* parameter can combine matching criteria using the wildcard % and/or array [] operators. For instance, the request "GET `/state_variables/?service_id[]=512&service_id[]=1024&name=%power%`" returns



(a)

SOAP CALL

```
POST /state_variables
SOAPAction: "http://www.uiot.org/state_varsinfo/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetStateVariablesInfo
          xmlns:m="http://www.uiot.org/statevarsinfo"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
           xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
          <stateVarName
xsi:type="xsd:string">S.V.N.</stateVarName>
          <sendEvents
xsi:type="xsd:boolean">S.E.</sendEvents>
          <dataType
xsi:type="xsd:string">D.T.</dataType>
          <serviceID
xsi:type="xsd:string">S.Id</serviceID>
          <value
xsi:type="xsd:string">value</value>
       </m:GetStateVariablesInfo>
    </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

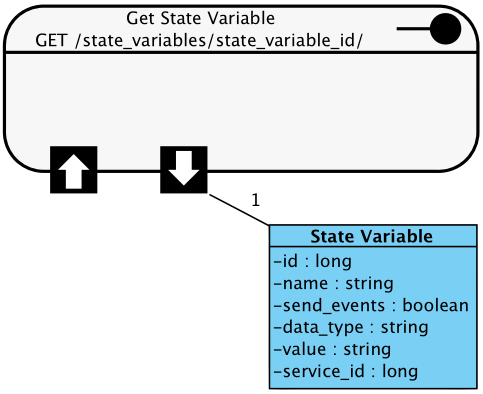
```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Body>
        <m:GetStateVariablesInfoResponse

    xmlns:m="http://www.uiot.org/statevarsinfo"
           SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <stateVarInfo
                 type="statevarxsd:statevar">SV[]
           </stateVarInfo>
       </m:GetStateVariablesInfoResponse>
    </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 17 Discovery request get state variables. **a** REST get state variables. **b** SOAP get state variables

an array of all state variables containing *power* in their name and belonging to services with id equal to 512 or 1024. In other words, the application accesses the state *power* of devices providing services 512 and 1024 in a sole request.

Figure 18 shows the route "GET `/state_variables/state_variable_id/`" used to read data about one specific state variable maintained by services available in the IoT Network. It returns a state variable Object (with attributes id, name, send events, data type, value and service id it belongs to) representing the State variables, within the Service Session Object, and having the attribute *id* equal to the URI *state_variable_id*. The request "GET `/state_variables/2048/`" returns a JSON representation of the State



(a)

SOAP CALL

```
POST /state_variable/state_variable_id
SOAPAction: "http://www.uiot.org/state_varinfo/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetStateVariableInfo
     xmlns:m="http://www.uiot.org/statevarinfo"
          SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
          xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <stateVarId
xsi:type="xsd:string">S.V.N.</stateVarId>
       </m:GetStateVariableInfo>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:GetStateVariableInfoResponse

   xmlns:m="http://www.uiot.org/statevarinfo"
          SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <stateVarInfo
                type="statevarxsd:statevar">SV
          </stateVarInfo>
      </m:GetStateVariableInfoResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

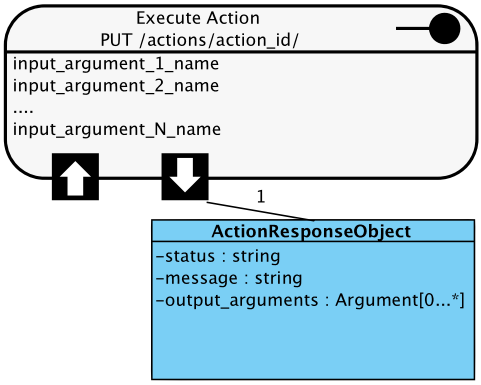
Fig. 18 Discovery request get state variable. **a** REST get state variable. **b** SOAP get state variable

variable, resided in a *Service Session Object*, with attribute *id* equal to *2048*. The *Get State Variable* request does not allow any URI parameter.

5.2 Control Requests

The second request type is destined to convey control operations sent from an application Control to a device. When this type of request is made to the API, a specific internal flow is followed to send the right command and to obtain the response. When an application request reaches the RR, this module builds a RouteObject and sends it to the HM *Actions*

requests handler. This last converts the request to a *Action Object* and sends it to the corresponding *Device Controller* in the CM (DC). This DC searches for the concerned objects within Device Session Objects and, if it does not find then there, the *Device Controller* will ask for *Device Entities* of the MM. The MM will query specific databases, through Data helpers, to find the desired Entity and will send it back to the CM. The CM converts the answer to a Session Object and runs a method named *ExecuteAction*, which receives as parameter the *Action Object* built in the Presentation Layer. The Device method *ExecuteAction* will run since it is responsible for communicating with the desired device to send the right commands necessary for the device to run the desired action. After this method is finished, the *Device Controller* builds and sends a response Object to the *Actions*



(a)

SOAP CALL

```
POST /actions/action_id/exec
SOAPAction: "http://www.uiot.org/action_exec/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:PerformAction

   xmlns:m="http://www.uiot.org/actionexec"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
          xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <actionID
xsi:type="xsd:string">A.Id</actionID>
         <parameter1
xsi:type="xsd:string">p1</parameter1>
         <parameter2
xsi:type="xsd:string">p2</parameter2>
          ........
         <parameterN
xsi:type="xsd:string">pN</parameterN>
      </m:PerformAction>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:ActionResponse
         xmlns:m="http://www.uiot.org/actionexec"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <actionResponse
            type="actionxsd:actionresp">ACTIONRESP
         </actionResponse>
      </m:ActionResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 19 Control request execute action. **a** REST execute action. **b** SOAP execute action

requests handler, which will build the final JSON or XML Response and send it back to the requesting application.

Figure 19 shows the route "PUT `/actions/action_id/`" used to control devices. It receives as parameters *input_argument_name*, one for each input argument needed by the Action with id equal to the URI *action_id*. To represent the execution of Actions, the method PUT has been chosen (instead of GET, POST and DELETE) because it is intuitive to think that a device will have its state altered/updated. For the SOAP approach it remains a RPC call over the POST method.

Still using the example of a refrigerator, as presented in Sect. 4.1, if the action named *set_temperature_to_specified_value* has the identification *16*, it can be performed by an application using the route "PUT `/actions/16/?new_temperature_value_in_celsius_degrees=3&time_to_wait_before_changing_temperature=0`".

In the response to control requests, the attribute *status* assumes either the value *SUCCESS* or the value *ERROR*, the attribute *message* has an open value and the attribute *output_argumets* contains only one element named "value_that_temperature_was_set_to" which would hold the actual value of the state variable *temperature*. If this state variable has its *id* equal to *8*, then its value could be later checked through the route "GET `/state_variables/8/`".

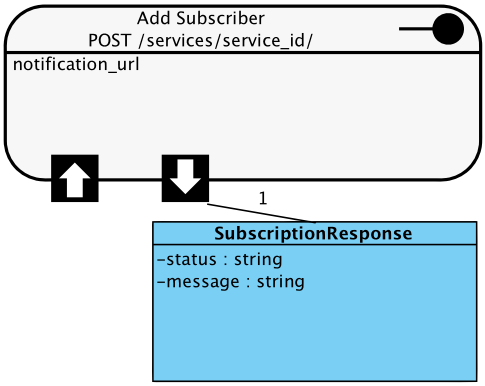
If an application needs to open a garage gate, for example, it could first find the right service, using the discovery request "GET `/services/?service_type=%garage%gate%`", then find the right action, using the discovery request "GET `/services/service_id/actions`", then issuing the action, using the control request "PUT `/actions/action_id/?direction=open`". An interesting idea of this example is that the application does not need to now the device which is performing the action, but just need to find an adequate service and request its execution. For the sake of efficiency in these requests, it is important to have well categorized devices and services such as the UPnP device and service type [25], the USB device class [26] or the Jini Services [27].

5.3 Subscription Requests

The third route type is destined to applications which need to receive updates of services' state variables. Two routes are available, one to set a subscriber for updates and the second to remove a subscription.

When this type of request is made to the API, a specific internal flow is followed to send data about the subscriber to the desired Session Service Object. In this case, When an application request reaches the RR, it builds a RouteObject and sends it to the HM *Service requests handler*. This last converts the request to a *URL Object* and sends it to the corresponding CM *Service Controller*(SC). This SC searches for desired objects within Service Session Objects and, if it does not find them, the SC will ask for *Service Entities* of the MM. The MM will query specific databases, through Data helpers, to find the desired Entity and send it back to CM. Then, the CM converts the Entity to a Session Object and runs either the service subscription methods *add_subscription* or *remove_subscription*, which receive as parameter the *URL Object* built the in Presentation Layer. The service method *add_subscription* creates a new subscription URL on the service subscribers list. The service method *remove_subscription* deletes an existing subscription URL from the service subscribers list.

After either those methods are performed, the *Service Controller* builds and sends a response Object to the *Service requests handler*, which will build the final Response and



(a)

SOAP CALL

```
POST /services/service_id/sub
SOAPAction: "http://www.uiot.org/service_sub/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:SubscribeService

   xmlns:m="http://www.uiot.org/servicesub"
          SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
          xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
         <serviceID
xsi:type="xsd:long">S.Id</serviceID>
         <notificationUrl
xsi:type="xsd:string">p1</notificationUrl >
      </m:SubscribeService>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:SubscribeResponse
         xmlns:m="http://www.uiot.org/servicesub"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <subResponse
            type="servicexsd:subresp"> SUBRESP
         </subResponse>
      </m:SubscribeResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 20 Subscription request add subscriber. **a** REST add subscriber. **b** SOAP add subscriber

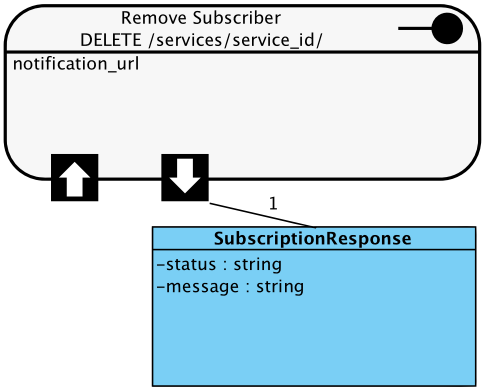
send it back to the requesting application. The JSON or XML response has an attribute *status* set either to *SUCCESS* or *ERROR* and an attribute *message* which is an open value string.

Figure 20 shows the route "POST `/services/service_id/`" used to add a URL to this notification system. It returns a JSON Object or a XML file representing the SubscriptionResponse Object. This request adds a new subscribing URL to the Service Session Object whose *id* equals to the URI *service_id*. An example is the request "POST `/services/2/?notification_url=192.168.0.4/receive_update`".

Every time a state variable has its value changed, the responsible Service Session Object sends an Object representing such State Variable, the same represented in Fig. 18, to each

URL present in its subscribers list. The concerned Session Object issues a POST request that is sent to the destination identified by the notification_url parameters taken from previous subscription requests. A response to the example request in the precedent paragraph would take the form "POST http://192.168.0.4/receive_update".

Figure 21 shows the route "DELETE `/services/service_id/`" used to remove a URL from the notification system. It returns an Object representing the SubscriptionResponse Object. This request removes an existing subscribing URL from the Service Session Object, this URL being represented by the *service_id*. An example would be the request "DELETE `/services/2/?notification_url=192.168.0.4/receive_update`".



(a)

SOAP CALL

```
POST /services/service_id/unsub
SOAPAction: "http://www.uiot.org/service_unsub/"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:UnsubscribeService

   xmlns:m="http://www.uiot.org/serviceunsub"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
         xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <serviceID
xsi:type="xsd:long">S.Id</serviceID>
        <notificationUrl
xsi:type="xsd:string">p1</notificationUrl >
      </m:UnsubscribeService>
   </SOAP:Body>
</SOAP:Envelope>
```

SOAP RESPONSE

```
HTTP 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns: SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <m:UnubscribeResponse

   xmlns:m="http://www.uiot.org/serviceunsub"
         SOAP:encodingStyle=

"http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <unsubResponse
               type="servicexsd:subresp"> SUBRESP
         </unsubResponse>
      </m:UnsubscribeResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

(b)

Fig. 21 Subscription request delete subscriber. **a** REST delete subscriber. **b** SOAP delete subscriber

5.4 Services Performed by the Security Module

The API security module (SM) is based on the proposal presented by [28], which introduces a REST security protocol to provide secure service communications. The API security module performs the encryption and signature of REST requests, as well as other defense measures described bellow.

Within the MM, the SM is responsible for maintaining data integrity, which consists in looking for differences between Session Objects and Data Base entities and providing updates for Session Objects when differences are found.

Within the HM, the SM ensures that requested URIs are valid and existent, checking passed parameters and their values. The SM filters values and solely allows known parameters to be passed for the subsequent processing thus protecting processing modules from data injection attacks.

The SM allows interaction with external entities only through the Request Router (for REST requests), Data helpers (to get device data) and the Session Object *execute_action* method (to perform actions in devices). All other internal components are private to the API and interact only with each other.

Also, the SM provides optional access control lists (ACL) and oAuth methods [29] as proposed in [30].

6 Experiments and Evaluation

The proposed IoT Services design was evaluated using an experimental approach. The API software has been developed and deployed in an experimental environment for testing its functions and for gathering performance data. To execute the tests, a server database representing a reference IoT network was populated with 5 IoT controllers, 17 device specifications, 73 types of services, 82 different actions. The experimental environment and the obtained results are presented, respectively, in Sects. 6.1 and 6.2.

6.1 Experimental Environment

The experimental environment is shown in Fig. 22 and its components technical specifications are presented in Table 2. The client side generates threads that each one simulates a

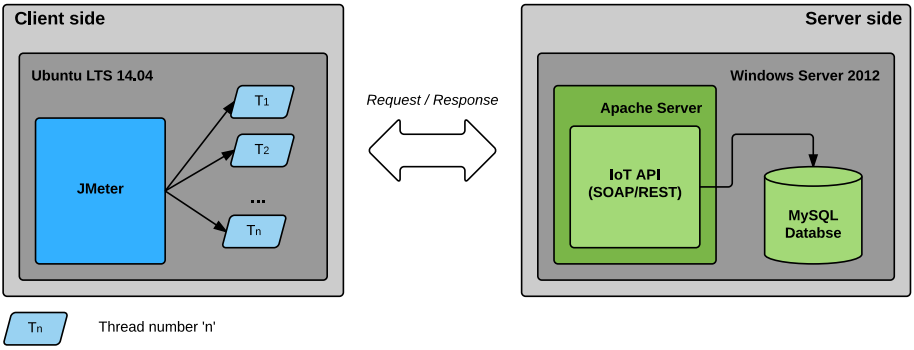


Fig. 22 Experimental environment

Table 2 Experimental environment specification

Node	OS	Processor	Memory	Storage	Environment	Language
Client	Ubuntu 14.04	Intel i5 2.5 GHz	4 GB	1 TB	JMeter	Java
Server	Windows Server	Intel XEON 2.79 GHz	32 GB	1 TB	Apache 2.4 MySQL 5.5	PHP

client using the system and sending several requests to the server. The software tool JMeter [31] manages thread creation and the submission of requests to the server, based on user configuration. Threads created by JMeter have listeners able to capture responses sent by the server. In the server side the IoT API was deployed running under the Apache server and storing data in a MySQL database. Client requests are received by the API, which performs information processing and sends the response back to concerned clients.

In order to reduce to a minimum the influence of external factors on the REST and SOAP communications, the Apache server has been set with basic high performance settings. Thus, the request response time, the processing load and server memory usage closely indicate real differences between REST and SOAP approaches. Specifically, the following settings were applied in the Apache server:

- *ThreadsPerChild: 350*—This directive sets the number of threads created by each child process. The child creates these threads at startup and never more.
- *MaxConnectionPerChild: 0*—This directive sets the limit on the number of connections that an individual child server process will handle. After MaxConnectionsPerChild connections, the child process will die. If MaxConnectionsPerChild is 0, then the process will never expire.
- *MaxRequestPerChild: 0*—This directive sets the limit on the number of requests that an individual child server process will handle. After MaxRequestsPerChild requests, the child process will die. If MaxRequestsPerChild is 0, then the process will never expire.

The results collected from the tests using this experimental environment are presented and discussed in Sect. 6.2.

6.2 Experiments and Collected Results

Given that they provide the same services, the REST and SOAP approaches were compared using performance indicators, different tests being applied to evaluate the request response time, the network traffic load, the processing load and memory occupation in the server. The tests were implemented in the Apache JMeter tool, consisting of 3 sets of tests, all of them using requests from Table 1 that were randomly chosen to be submitted to the server.

- *Response time test:* It consists of 200 clients (threads) randomly submitting different requests to the server, and verifying the response time of each request. Each client is active for 10 s and sends 20–35 requests. Every 5 s, a new customer is created to submit new requests.
  In response time tests, five thousand requests are sent to the server. The response time for each request type is shown in Fig. 23. SOAP requests present a response time in the range [5 ms, 37 ms], with an average of 17 ms and a standard deviation of 10 ms,

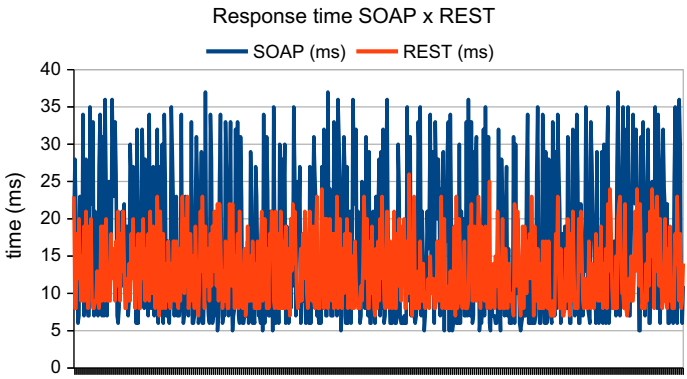


Fig. 23 Response time to complete requests

while REST requests present a response time in the range [7 ms, 26 ms], with an average of 13 ms and a standard deviation of 4 ms.

On average for the tests performed, the response time of REST requests is 23 % lower than the same requests performed with SOAP. Moreover, the standard deviation of the REST average response time is 60 % lower than the SOAP one, indicating that the REST approach is more stable regarding this indicator.

- *Data load test:* It consists of submitting different requests to the server, varying the type of requests, and checking the amount of bytes sent by server.

  In data load tests, all requests from Table 1 are sent to the server and the response data load for each request type is shown in Fig. 24. For all the different request types (GET, POST, PUT, DELETE) the number of bytes produced by SOAP is higher. On average, SOAP-XML requires approximately 33 % of extra bytes to send the same information than in the REST-JSON format. Moreover, as the amount of traffic grows, the difference between SOAP-XML and REST-JSON also increases.
- *Server load (stress test):* It consists of 1000 clients (threads) randomly submitting different requests to the server, and verifying the processing load and memory

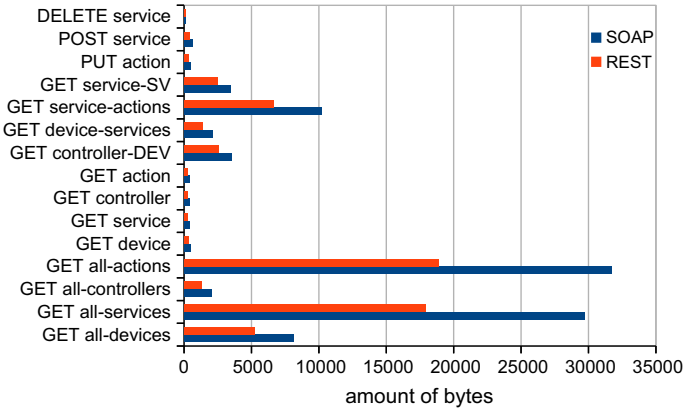


Fig. 24 Average data load per request in bytes

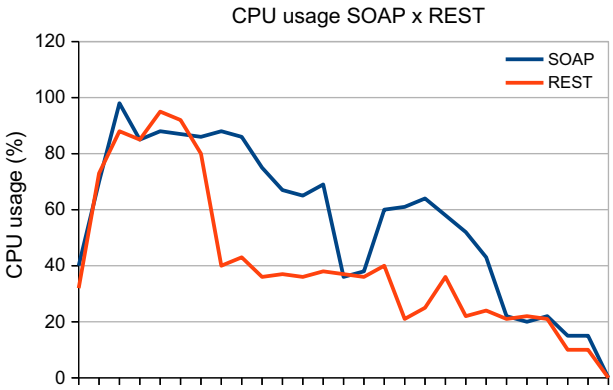


Fig. 25 CPU usage for SOAP and REST

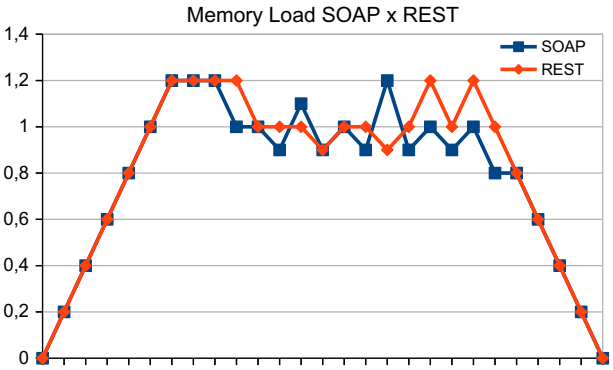


Fig. 26 Memory usage for SOAP and REST

occupation in server. Each client is active for 3 s and sends 40–50 requests. Every second, a new customer is created to submit new requests.

In server load tests, approximately 450,000 requests are sent to the server during 50 min. The CPU usage in both implementations is shown in Fig. 25. On average, REST consumed 40 % of CPU capacity and SOAP 55 %. Both, REST and SOAP, peak at the beginning of the simulation and gradually stabilize. This CPU utilization curve can be explained by the API ability to create session objects, not needing to reprocess requests previously performed.

The server memory load can be seen in Fig. 26. Both implementations used about the same amount of memory during the tests. On average, the SOAP implementation holds 0.9 MB of memory while the REST implementation holds 1 MB. Figure 26 shows that the memory consumption increases progressively until it attains stability, then remains almost unchanged until the end of the simulation. This behavior is also explained by the creation of session objects by the API, which avoids reprocessing requests already performed.

7 Conclusions and Future Work

The research work presented in this paper proposes an application programming interface for accessing IoT services through REST and SOAP communications. The API was developed following the UPnP specification, in order to allow its applicability in IoT architectures with a variety of devices.

This paper proposes a specification of the interesting services to be provided by an IoT API based on possible utilization examples, including services for abstracting devices, as well as services that provide IoT control and monitoring operations. Aiming to find the most efficient communication approach to these IoT services, the API was developed and deployed in an experimental environment. Then, comparative tests between REST and SOAP were conducted to evaluate response time, data traffic load and processing load. The experiments show that the response time of REST requests is on average 23 % lower than SOAP requests. Also, the information submitted in SOAP-XML format requires, on average, 33 % more bytes compared to REST-JSON. Regarding the API server stress tests, CPU utilization was 55 % for SOAP and 40 % for REST, while the memory usage was almost the same for both approaches.

Considering that IoT architectures aim to take advantage of the largest possible number of devices (clients), allowing them to communicate by exchanging data, it is arguable that the lower the time of communication and the data size, the better the performance of a IoT service. In this sense, even considering that SOAP may be a more mature approach, the REST approach is the most appropriate for communication between clients and servers in an IoT architecture.

As a future work, it would be interesting to add an additional ontology module on the presentation layer of the proposed IoT API to allow better and simpler control and discovery of devices and services. Also, it seems interesting to study algorithms to correlate actions with the same result to find the best device to actually perform them and balance the work load. Finally, experimental tests with multiple applications and multiple requests from different computers might bring interesting new results and point possible improvements.

Acknowledgments This work was supported by "Programa de Financiación de la Universidad Complutense de Madrid - Banco Santander para Grupos de Investigación UCM (Referencia: GR3/14)". In addition, the authors wish to thank the Brazilian research, development and innovation Agencies CAPES (Grant FORTE 23038.007604/2014-69), FINEP (Grant RENASIC/PROTO 01.12.0555.00) and the National Post-Doctorate Program (PNPD/CAPES) for their support to this work.

References

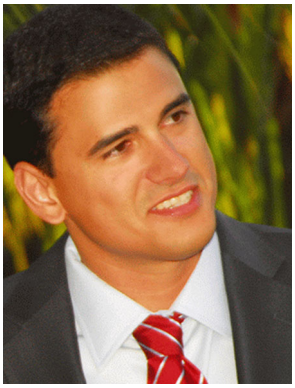
1. Ashton, K. (2009). That 'internet of things' thing. *RFiD Journal*, *22*(7), 97–114.
2. Want, R. (2000). Remembering mark weiser: Chief technologist, xerox parc. *IEEE Personal Communications*, *7*(1), 8–10.
3. Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *CISCO White Paper* (online). https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.
4. Plug, U., & Forum, P. (2000). *Understanding universal plug and play* (online). http://www.upnp.org/download/UPNP_understandingUPNP.doc.
5. Ferreira, H. G. C., Canedo, E. D., & de Sousa, R. T. (2014). A ubiquitous communication architecture integrating transparent UPnP and rest APIs. *International Journal of Embedded Systems*, *6*(2), 188–197.

6. Thoma, M., Meyer, S., Sperner, K., Meissner, S., & Braun, T. (2012). On iot-services: Survey, classification and enterprise integration. In *2012 IEEE international conference on green computing and communications (GreenCom)* (pp. 257–260).
7. Bandyopadhyay, D., & Sen, J. (2011). Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, *58*(1), 49–69. doi:10.1007/s11277-011-0288-5.
8. Gronbaek, I. (2008). Architecture for the internet of things (iot): API and interconnect. In *Second international conference on sensor technologies and applications, 2008 (SENSORCOMM'08)* (pp. 802–807), IEEE.
9. Grønbæk, I., Nord, M., & Jakobsson, S. (2007). Abstract service API for connected objects. *R&I Report*, *18*, 2007-06.
10. Atzori, L., Iera, A., & Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, *54*(15), 2787–2805.
11. Da Xu, L., He, W., & Li, S. (2014). Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, *10*(4), 2233–2243.
12. Kim, Y., Lee, S., & Chong, I. (2014). Orchestration in distributed web-of-objects for creation of user-centered iot service capability. *Wireless Personal Communications*, *78*(4), 1965–1980.
13. Presser, M., Barnaghi, P. M., Eurich, M., & Villalonga, C. (2009). The sensei project: Integrating the physical world with the digital world of the network of the future. *IEEE Communications Magazine*, *47*(4), 1–4.
14. Project, I.-A. (2009). *Internet of things architecture* (online). http://www.iot-a.eu/.
15. iCore Project. (2009). *icore—cognitive management framework* (online). http://www.iot-icore.eu/.
16. Project, B. (2009). *Butler* (online). http://www.iot-butler.eu/.
17. Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, *29*(7), 1645–1660.
18. Cerqueira Ferreira, H., Dias Canedo, E., & de Sousa, R. (2013). IoT architecture to enable intercommunication through rest api and UPnP using ip, zigbee and arduino. In *2013 IEEE 9th international conference on wireless and mobile computing, networking and communications (WiMob)* (pp. 53–60), IEEE.
19. Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, *2*(2), 115–150.
20. Jakl, M. (2008). *Rest representational state transfer. Technical report*. Vienna: University of Technology.
21. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, *2*, 86–93.
22. Plug, U., & Forum, P. (2004). *UPnP device architecture v1.0* (online). http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf.
23. Richardson, L., & Ruby, S. (2008). *RESTful web services*. Sebastopol: O'Reilly Media, Inc.
24. Fowler, M. (2002). *Patterns of enterprise application architecture*. Reading: Addison-Wesley Longman.
25. Plug, U. (1999). Play forum. In *About the UPnP plug and play forum*. http://www.upnp.org.
26. Axelson, J. (2009). *USB complete: The developer's guide*. Chicago: Lakeview Research.
27. Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., & Wollrath, A. (1999). *Jini specification*. Reading: Addison-Wesley Longman.
28. Serme, G., de Oliveira, A. S., Massiera, J., & Roudier, Y. (2012). Enabling message security for restful services. In *2012 IEEE 19th international conference on Web services (ICWS)* (pp. 114–121), IEEE.
29. Hardt, D. (2012). *The oauth 2.0 authorization framework* (online). http://tools.ietf.org/html/rfc6749.html.
30. Ferreira, H. G. C., de Sousa, R. T., de Deus, F. E. G., & Canedo, E. D. (2014). Proposal of a secure, deployable and transparent middleware for internet of things. In *2014 9th Iberian conference on information systems and technologies (CISTI)* (pp. 1–4), IEEE.
31. Halili, E. (2008). *Apache JMeter*. Birmingham: Packt Publishing.

Caio César de Melo Silva graduated in Computer Science from the Federal University of Santa Catarina, Brazil, in 2011. He received his Master Degree in Computer Science from the Federal University of Santa Catarina, Brazil in 2013, in the domain of computer systems and the specific area of context awareness. He is an assistant professor and researcher at the University of Brasilia, Network Engineering undergraduate course.

Hiro Gabriel Cerqueira Ferreira graduated in Communication Networks Engineering from University of Brasilia, Brazil in 2012. He received his Masters degree in Electrical Engineering at University of Brasilia with major domain in telecommunication and specific area on internet of things in 2014. He also is co-founder of Green Tecnologia, Brazilian Enterprise of Software Development, and a researcher at Decision Technologies Laboratory (LATITUDE), University of Brasilia.

Rafael Timóteo de Sousa Júnior was born in Campina Grande—PB, Brazil, on June 24, 1961. He graduated in Electrical Engineering from the Federal University of Paraíba—UFPB, Campina Grande—PB, Brazil, 1984, and got his Doctorate Degree in Telecommunications from the University of Rennes 1, Rennes, France, 1988. He worked as a software and network engineer in the private sector from 1989 to 1996. Since 1996, He is a Network Engineering Professor in the Electrical Engineering Department, at the University of Brasília, Brazil. From 2006 to 2007, supported by the Brazilian R&D Agency CNPq, He took a sabbatical year in the Group for the Security of Information Systems and Networks, at Ecole Superiéure d'Electricité, Rennes, France. He is a member of the Post-Graduate Program on Electrical Engineering (PPGEE) and supervises the Decision Technologies Laboratory (LATITUDE) of the University of Brasília. His field of study is distributed systems and network management and security.



Fábio Buiati received a Computer Science degree from the University of Goiania, Brazil, in 2000, M.Sc. degree from the University of Brasília, Brazil, in 2004, Ph.D. in Computer Science from the University Complutense of Madrid, Spain and a post-doctoral from the INRIA Rennes, France. He is a researcher at Decision Technologies Laboratory (LATITUDE) of the University of Brasília. His research interests are in data mining, Internet of Things, smart cities, machine learning and big data.

Luis Javier García Villalba IEEE Senior Member, received the Telecommunication Engineering degree from the Universidad de Málaga (Spain) in 1993, the M.Sc. degree in Computer Networks in 1996, and the Ph.D. degree in computer science in 1999, the last two from the Universidad Politécnica de Madrid (Spain). He was a Visiting Scholar at the Research Group Computer Security and Industrial Cryptography (COSIC), Department of Electrical Engineering, Faculty of Engineering, Katholieke Universiteit Leuven (Belgium) in 2000, and a Visiting Scientist at the IBM Research Division (IBM Almaden Research Center, San Jose, CA, USA) in 2001 and in 2002. He is currently an Associate Professor in the Department of Software Engineering and Artificial Intelligence at the Universidad Complutense de Madrid (UCM) and Head of the Complutense Research Group GASS (Group of Analysis, Security and Systems, http://gass.ucm.es), which is located at the Faculty of Computer Science and Engineering at the UCM Campus. His professional experience includes research projects with Hitachi, IBM, Nokia, Safelayer Secure Communications, VISA, and H2020 projects. His main research interests are in information security and computer networks. Dr. García Villalba is an Associate Editor in Computing for IEEE Latin America Transactions since 2004, participates in the editorial board of several journals (IET Communications, IET Networks, IET Wireless Sensor Systems, International Journal of Ad Hoc Networks and Ubiquitous Computing, The International Journal of Digital Crime and Forensics, International Journal of Security and Its Applications, International Journal of Multimedia and Ubiquitous Engineering, International Journal of Future Generation Communication and Networking, International Journal of Computational Science, among others) and is contributor of textbooks.