RAISe: REST API Approach for IoT Services

Hiro Gabriel Cerqueira Ferreira, Caio César de Melo e Silva,
Rafael Timóteo de Sousa Junior, Cláudio Alexander Santoro Wunder
LATITUDE, Electrical Engineering Department, University of Brasilia (UnB)
Campus Darcy Ribeiro - Asa Norte Brasília, DF - Brazil, 70910-900
hiro.ferreira@gmail.com, caio.cmsilva@gmail.com, desousa@unb.br, claudio.santoro@redes.unb.br

***Abstract*—This paper proposes a REST API approach for IoT services based on UPnP's device model. Here we define methods and procedures of an application programming interface to allow a UPnP like controlling and eventing through a RESTful service. In order to encompass devices diversity, the proposed API introduces a uniform abstractions model to bring a common manner to manage objects. Moreover, RAISe yields an abstract service interface to provide features instead of devices. In other words, It gives transparent access to devices capabilities, known as device's services, hiding the provider object from API users. RAISe has been modeled to ease IoT Smart Objects remote management and, as a module of UIoT Middleware, it has been deployed in real scenarios demonstrating fast responses and low resources usage.**

***Index Terms*—Internet of Things, REST API, Service, UIoT, UPnP, RESTful Services.**

I. Introduction

Predicted by Mark Weiser [1] in the beginning of 90's [2], Computing is going through its third large Phase, the Ubiquitous Era. Such event accounts multiple processors/devices acting to provide seamless services to humans. It is predicted that more then 1 trillion devices will be connected to the internet by 2020 [3].

The term Internet of Things (IoT) was created by [4] Kevin Ashton contextualizing RFID applications. IoT has grown larger and nowadays has many acronyms to caracterize different points of view like WoT, EoT, IIoT, HIoT, etc.

To permit simplified access to devices resources, thus allowing true context-aware environments, smart application programming interfaces (API) must be deployed. Without good API models, the communication and longevity of IoT can get compromised.

This work brings an efficient API for IoT based in RESTful Services, named RAISe. It is constructed over UPnP Device Architecture and web system standards and it provides uniform access to resources (despite devices diversity), uses low computacional resources and allow dynamic device controlling through simple querying system.

This paper is organized as follows. Section II introduces concepts and references of technologies related to this work. Section III presents the proposed REST API Approach for IoT Services. Section IV brings results collected in real scenario deployments and tests. Finally, on section V, conclusions and future works are brought to discussion.

II. Related Concepts

In this section, we raise some concepts related to IoT architectures, REST services and UPnP specification in order to characterize the scenario which fits our proposal. References contain information to allow deeper study on presented subjects.

1) IoT Architectures: There are plenty intelligent and complete IoT existent Architectures to allow device controlling and monitoring. They are important to allow encompassing legacy and latter developed device in a intelligent way, also bringing already smart devices to IoT Networks. Reference [5] brings an interesting overview on the topic of middleware and architecture initiatives and internet of things in general. This work has its prototype based on UIoT Middleware [6] and Communication Model, because it easily allow new modules to be added in a transparent way, through the usage of web sockets and standardized communication model. UIoT's middleware has considerations on scalability, transparency, diversity, mobility, longevity, performance, ease of use, expansibility and it is open source built upon widespread technology.

2) UPnP: Universal Plug and Play (UPnP)[7] is a protocol stack and standard to allow universal transparent access to logical objects resources. It runs in 6 main phases (Addressing, Discovery, Description, Control, Eventing and Presentation) and is based on TCP/IP. It abstracts devices in a precise Device Architecture and it is mostly used to share audio and video resources among machines of the same local network. It functions over SSDP,GENA and SOAP to perform most of its controlling, discovery and eventing activities.

3) REST Services: We have followed Roy Fields proposal of Representational State Transfer(REST) to build RAISe[8]. It is a client-server model, transparent to operational system and programming language, runs over TCP/IP, allows usage of caches and has uniform interface. Fits perfectly on IoT.

III. RAISe: REST API for IoT Services

This section presents the REST API Approach for IoT services. It is divided as follows: Section III-A presents RAISe's logical abstraction of devices, the base for all provided services. Section III-B presents the abstract model of RAISe, conceptualizing its layers, modules and components. Section III-C explains how information flow occurs in the API, more specifically, it presents all REST routes(for device discovery, device controlling and subscription for state changes) and involved parts of the abstract model for each request.

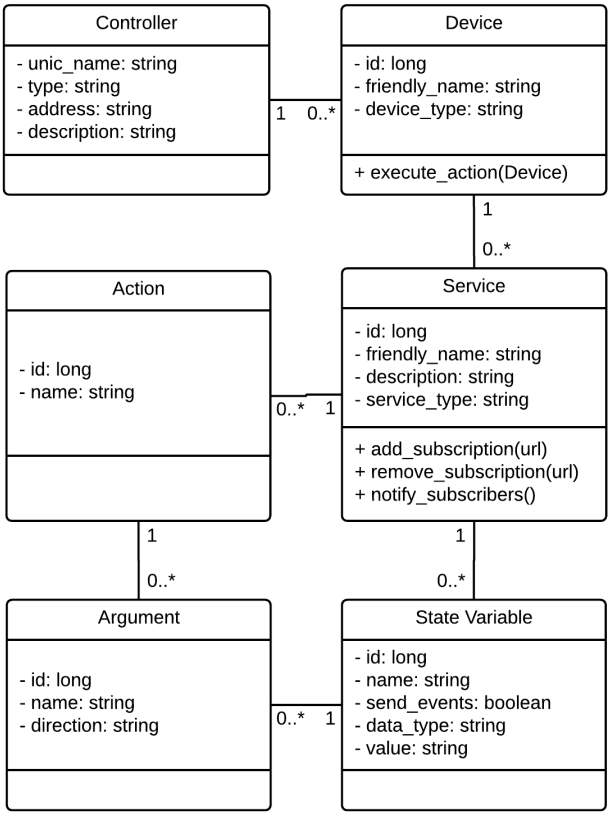

Figure 1. RDA - RAISe Device Architecture.

Finally, in Section III-D, it is described how the security module operates in the API layers.

A. RAISe Device Architecture

RAISe has a precise manner to logically abstract its devices in order to ensure uniformity on resources access. Named as Raise Device Architecture(RDA), it inherits most of it's schema from UPnP Device Architecture [9]. RDA is composed by UPnP sub-entities (devices, services, actions, state variables and arguments) and, additionally, introduces a sixth entity, named controller.

In the case of UPnP, a device provides services to users. A service is issued through actions. Actions have parameters, named arguments, to orientate how a device state should be changed. A service has state variables which are altered by actions and represented as input and output arguments. An example of a device abstracted in this Architecture would be a smart refrigerator. The device would be the refrigerator and it 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 specified value". The action "set temperature to 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's state variable "temperature". Action "set temperature to specified value" could have the following output argument: "value that temperature was set to", which would hold the value of state variable "temperature" after the action has been performed.

To help encompassing legacy plain objects, turned into smart object through micro-controllers usage, RAISe introduces a new entity to UPnP's Device Architecture: the controller. As a matter of better resource usage, it is assumed that a controller can hold more than one device. Even though UPnP Device Architecture originally supports devices embedded in a root device, what could solve this issue, we believe that it is more didactic, formal and clear to separate the case of a multi-functional physical device, like a printers with scanner within the same physical device, from multiple physically separated devices managed by a micro-controller, like a raspberry pi controlling all appliances of a room. In the case of refrigerator's example, it could be added a controller named "kitchen controller" which would have 3 devices attached to it: "Kitchen light", "Coffee pot" and "Refrigerator".

Case a device is already a smart object and, thus, need no additional controllers, it can be abstracted without entity controller, just like UPnP refrigerators first example.

Even considering devices' diversity, RDA brings uniformity to resources access. With its sub entities and their relationships, RDA allows the API to have well defined methods to search, control and monitor state of all encompassed devices. Figure 1 shows the relations among sub entities that compose RAISe Device Architecture.

B. RAISe Abstract Model

Since REST demands the use of hypertext, RAISe API, shown in figure 2, was initially designed to increase scalability on IoT architectures, taking advantage of loose coupling between client and server [10]. Furthermore, RAISe was modeled to manipulate uniformly IoT smart objects, usufructing the device architecture specification presented in section III-A. The API also predicts a security module which has components in all different layers, providing protection to trafficked information from the first entry in the API.

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

RAISe layers (presentation, business logic and data access) were designed based on traditional web systems standards [11] with some fundamental differences. For instance, 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.

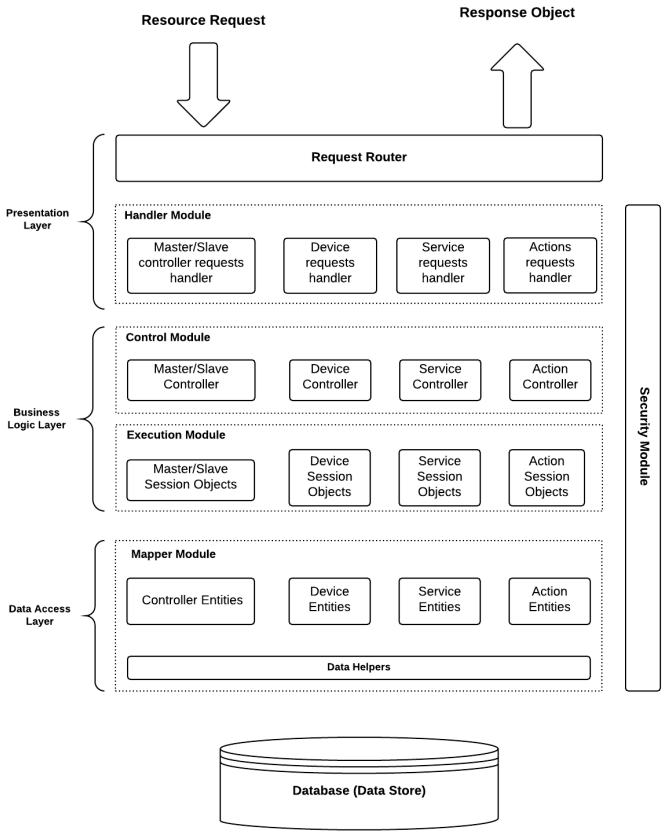

Figure 2. RAISe API Modules.

In RAISe, the presentation layer is also a bridge to business logic layer, but the system does not deal with user interactions. Instead, it allows clients to send a plurality of requests for smart objects in an IoT architecture. In the following subsections, the characteristics and features of RAISe layers and their respective components are presented.

1) Presentation layer: The presentation layer (PL) contains the components that implement and display the client requests interface and manage client solicitations. This layer includes controls to client demands, as well as components that route the submitted requests. Thus, PL is composed of two main logical modules referred as request router (RR) and handler module (HM). Figure 2 shows how the presentation layer fits into the RAISe application architecture.

The main function of RR is to redirect client requests to handler module components. The request router collects all the information in the REST request, creates an object populated with the parameters collected, and send it to the component which handles the requested entity. For example, suppose that the request has been as follows: ”GET controllers/0125/devices/”, the RR will created a instance of a route object, as shown in figure 3, and send it to *Master/Slave controller requests handler* component.

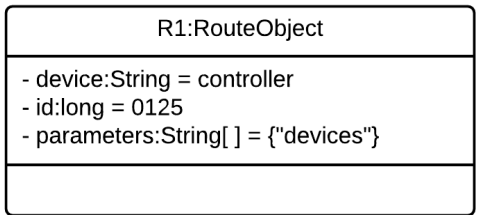

Figure 3. Route Object.

Broadly, the handler module receives a route object sent by RR and sends a request to the business layer. Particularly, a HM component (request handler) send the type of the REST request (GET, POST, PUT, DELETE), the route object identifier and the parameters of the request to the corresponding component in the control module, which search for the requested session object in execution module. The handler module also performs a validation on the submitted route object, explained in detail in section III-D.

2) Business logic layer: The business logic layer (BLL) implements the core functionality of the system, and encapsulates the relevant business logic. BLL consists of two modules: control and execution, jointly these modules manage active smart objects on IoT infrastructure. It is important to point out that, unlike traditional web services, RAISe business logic layer does not expose service interfaces, which are only accessible through the presentation layer.

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

Execution module (EM) manages the session objects currently being used on IoT network. Session objects can be defined as instances of UPnP entities that stores 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 respond to the requests made by the control module. If certain information is not available at execution module, a request is sent to the data access layer, and a new session object is created.

3) Data access layer: The data access layer (DAL) provides access to data hosted within the boundaries of the system. DAL exposes interfaces that the components in the business layer can consume. The data access layer has only one module, referred to as mapper module(MM), which has the role 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 a IoT network, which might be within a middleware. This module also boasts a data helper component which abstract the logic required to access the underlying data stores. It centralize common data access functionality in order to make the system easier to

configure and maintain.

The idea of the mapping done by data helpers is to retrieve requested data from any sort of data store (like traditional databases, NoSQL databases, TXTs, XMLs, JSONs, etc) and convert them into a local well known and defined model object. Figure 2 represented model objects as *Entities* inside the MM block, which are converted to session objects when requested.

C. Request Processing Flow and REST Routes

Understood how RAISe does to build logical representations of smart objects, how it internally works and how internal modules interact, it is time to explain and show RAISe's allowed routes for REST requests and the flux to process them. RAISe was modeled to run seventeen request processing flows. Fourteen of them are destined to retrieve information about entities of RDA and they are presented in section III-C1. One flow is meant to issue control commands to devices and it is presented in section III-C2. The two remaining are destined to allow applications' subscription for receiving notifications of new values assumed by state variables, thus, getting real time access to devices' states in a PUSH like system. Subscription flows are presented in section III-C3. Table I brings an abstract of all possible Discovery Requests.

1) Discovery Requests: RAISe's Discovery Requests are meant to allow applications to learn about encompassed IoT networks capacities and to allow them to search for specific devices or services. With this type of request it is possible to collect data about states of a device and to know which arguments/parameters should be sent to control such device through Control Requests presented in Section III-C2. RAISe allows dynamic searches for device types and, as a separated feature, it permit searches for service types that can get things done independently of which device is going to perform the desired final action. Those latest manners are interesting because they allow to monitor and perform actions without knowing the entire IoT network.

When a Discovery request is made to RAISe, a common internal flow is followed to send the right response. When an application request reaches RR, it builds a RouteObject and send it to HM. HM converts the request to a internal known query object (Controller, Device, Service or Action) and send it to CM. CM searches for desired objects on MM, querying specific databases through Data helpers, to find desired Entities. Once that is done, MM returns Found Entities to CM. Case those Entities aren't yet Session Objects, CM will convert them to become it. Next, CM send Session Objects pointer back to HM. HM will build JSON Response Objects containing the Session Objects pointed by CM and send it back to requesting application.

Before proceeding, it is important to explain the representation used for all REST requests shown in this work. Figure 4 shows a generic REST Request. Its 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 amount sent,

Table I
RAISE URI FOR REST API

Method	Resource URI	Expected result
GET	/controllers/	JSON Array with JSON Objects of all existent controllers.
GET	/controllers/unique_name/	JSON object with controller's information where controller unique name equals to parameter unique_name of URI.
GET	/controllers/unique_name/ devices/	JSON Array with JSON Object of devices associated to controller with unique name equal to parameter unique_name of URI.
GET	/devices/	JSON Array with JSON Object of all existent devices.
GET	/devices/device_id/	JSON Object with device's information where device id equals to parameter device_id of URI.
GET	/devices/device_id/services/	JSON Array with JSON Object of existent services associated to device with id equal to parameter device_id of URI.
GET	/services/	JSON Array with JSON Object of existent services
GET	/services/service_id/	JSON object with service's information where service id equals to parameter service_id of URI.
GET	/services/service_id/actions/	JSON Array with JSON Object of actions associated to service with id equal to parameter service_id of URI.
GET	/services/service_id/ state_variables/	JSON Array with JSON Object of state variables associated to service with id equal to parameter service_id of URI.
GET	/actions/	JSON Array with JSON Object of existent actions
GET	/actions/action_id/	JSON Object with data related to action and its arguments where action id equals to parameter action_id of URI.
GET	/state_variable	JSON Array with JSON Object of existent state variables.
GET	/state_variable/ state_variable_id/	JSON Object with data related to state variable with id equal to parameter state_variabel_id of URI.
PUT	/actions/action_id/	Perform action, where action id equals to parameter action_id of URI, on correct device.
POST	/services/service_id/	Add a new notification url to ervice where service id equals to parameter service_id of URI.
DELETE	/services/service_id/	Remove an existent notification url to ervice where service id equals to parameter service_id of URI.

and optional Response Object with its parameters and amount returned.

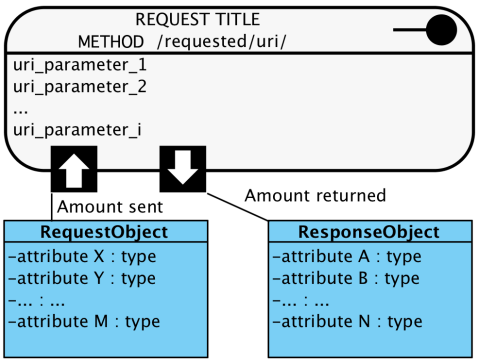

Figure 4. REST Request Representation

Repetition of parameters is allowed by the usage of array operand "[]" to permit queries of type OR. As an example: parameters "*?name[]=light&name[]=room*" would return objects containing the attribute *name* with value equal to *light* as well as objects with attribute name equal to *room*.

Another important feature is the wildcard represented by the % symbol. It is meant to be placed within the sent value of a parameter to permit queries of type AND. It is a substitute for zero or more characters on the sent value. In URLs, the representation of % character is %25 but, for the sake of simplicity, we are going to represent it only as % when working with URLs and URIs. As some possible combinations using % wildcard, we present the following examples: parameter "*&name=%room*" would return objects containing attribute *name* ending with the word *room*, parameter "*&name=kitchen%*" would return objects containing attribute *name* starting with the word *kitchen*, parameter "*&name=%car%*" would return objects containing attribute *name* with the word *car* present in any position(beginning, middle or ending), parameter "*&name=light%pole*" would return objects containing attribute *name* starting with with the word *light* and ending with the word *pole*, parameter "*&name=%coffee%pot*" would return objects containing attribute *name* having the word *coffe* in any position and ending with the word *pot*, parameter "*&name=%energy%efficient%*"would return objects containing attribute *name* with the word *energy* followed by the word *efficient*.

Now that the Discovery Request Flow has been presented and the REST Request Representation has been clarified, among with the optional usage of operand [] and wildcard % on URI parameters, following paragraphs present each different request and explain their purpose.

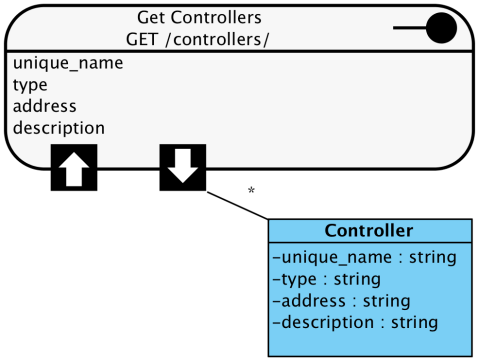

Figure 5. Discovery Request - Get Controllers

Figure 5 shows the route "GET /controllers/" used to read all data about controllers available on encompassed IoT Network. It returns an array of JSON Objects (with attributes unique name, type, address and description) of found Controllers Session Objects. To allow specific queries, URI parameters *unique_name*, *type*, *address*, and *description* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the controllers containing received values on their related attributes. For instance, a request like "GET /controllers/?type=%zigbee%" would return an array of all controllers containing *zigbee* in its type, in other words, we could have access to all zigbee connected controllers inside the IoT Network.

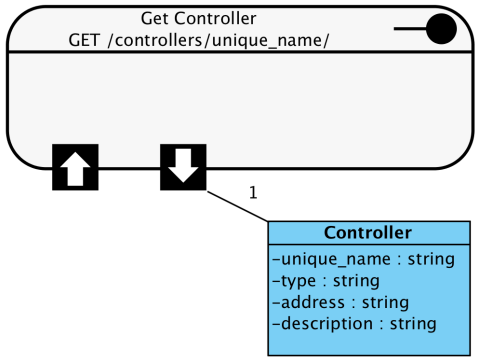

Figure 6. Discovery Request - Get Controller

Figure 6 shows the route "GET /controllers/unique_name/" used to read all data about one specific controller available on encompassed IoT Network. It returns a JSON Object (with attributes unique name, type, address and description) representing the Session Object that has the attribute unique name equals to URI part represented by *unique_name*. An example would be the request "GET /controllers/garage/". It would return a JSON representation of the *Master/Slave Session Object* with attribute *unique_name* equals to *garage*. *Get Controller* request does not allow any URI parameter.

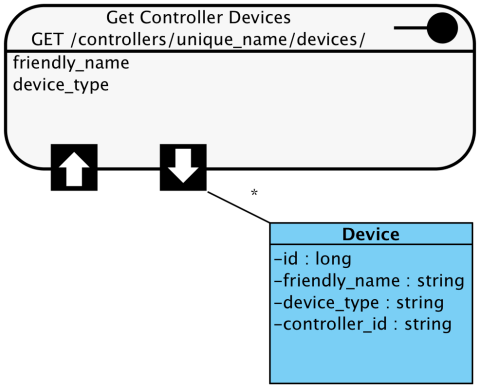

Figure 7. Discovery Request - Get Controller Devices

Figure 7 shows the route "GET /controllers/unique_name/devices/" used to read all data about devices within one specific controllers available on encompassed IoT Network. It returns an array of JSON Objects (with attributes id, friendly name, device type and encompassing controller id) of found Device Session Objects belonging to controller with *unique name* equals to URI part *unique_name*. To allow specific queries, URI parameters *firendly_name* and *device_type* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the controller's devices containing received values on their related attributes. For instance, a request like "GET /controllers/children_room/devices/?device_type[]=%lighting%" would return an array of all devices, encompassed by controller *children_room*, containing *lighting* in its type, in other words, we could have access to all light related devices inside controller of children's room.

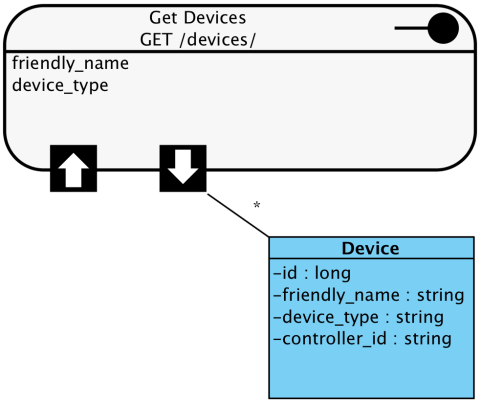

Figure 8. Discovery Request - Get Devices

Figure 8 shows the route "GET /devices/" used to read all data about devices available on encompassed IoT Network, including those attached and not attached to controllers. It returns an array of JSON Objects (with attributes id, friendly name , device type and encompassing controller id) of found Device Session Objects. To allow specific queries, URI parameters *friendly_name* and *device_type* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the devices containing received values on their related attributes. For instance, a request like "GET /devices/?device_type[]=%temperature%" would return an array of all devices containing *temperature* in its type, in other words, we could have access to all temperature related devices in the IoT Network.

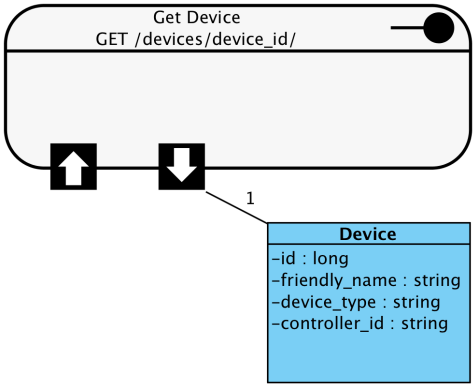

Figure 9. Discovery Request - Get Device

Figure 9 shows the route "GET /devices/device_id/" used to read all data about one specific device available on encompassed IoT Network, including those attached and not attached to controllers. It returns a JSON Object (with attributes id, friendly name, device type and encompassing controller id) representing the Session Object that has the attribute *id* equals to URI part represented by *device_id*. An example would be the request "GET /device/32/". It would return a JSON representation of the *Device Session Object* with attribute *id* equals to *32*. *Get Device* request does not allow any URI parameter.

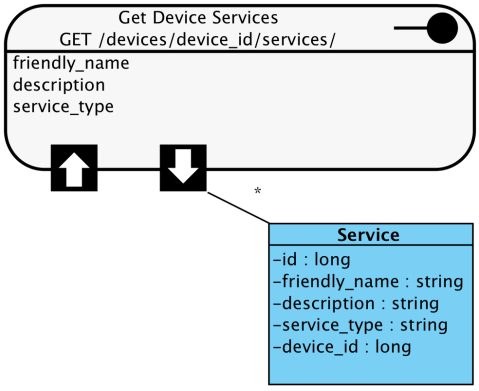

Figure 10. Discovery Request - Get Device Services

Figure 10 shows the route "GET /devices/device_id/services/" used to read all data about services within one specific device available on encompassed IoT Network, including those attached and not attached to controllers. It returns an array of JSON Objects (with attributes id, friendly name, description, service type and service provider as device id)

of found Service Session Objects belonging to device with *id* equals to URI part *device_id*. To allow specific queries, URI parameters *firendly_name*, *description*, and *service_type* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the device's services containing received values on their related attributes. For instance, a request like "GET /devices/16/ services/?service_type[]=%humidity%manager%" would return an array of all services, within the device with attribute *id* equal to *16*, containing attribute type with word*humidity* followed by *manager*. In other words, we could have access to all services of device 16, lets say that this device is the greenhouse's humidity handler, to handle the greenhouse humidity.

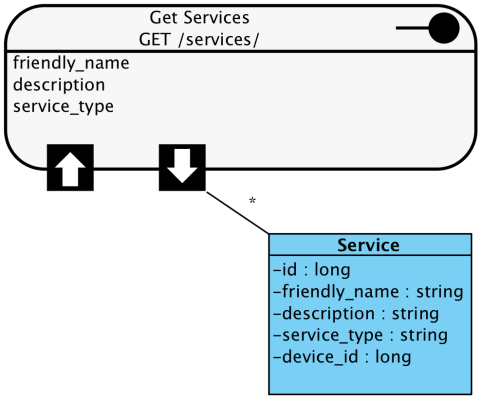

Figure 11. Discovery Request - Get Services

Figure 11 shows the route "GET /services/" used to read all data about services provided by devices on encompassed IoT Network. It returns an array of JSON Objects (with attributes id, friendly name, description, service type and service provider as device id) of found Service Session Objects. To allow specific queries, URI parameters *friendly_name*, *description* and *service_type* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the services containing received values on their related attributes. For instance, a request like "GET /services/?service_type[]=%library%access%" would return an array of all services containing *library* followed by *access* in its type, in other words, we could have access to all services present on the IoT Network that provide access to the library, independently of which device will issue the final action.

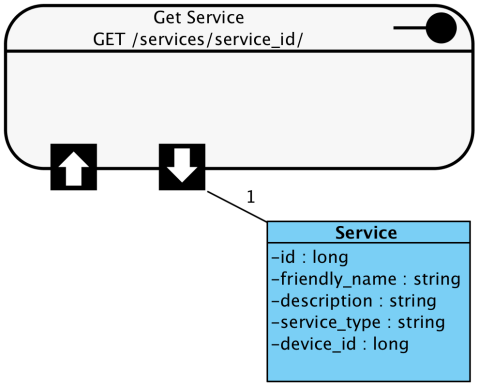

Figure 12. Discovery Request - Get Service

Figure 12 shows the route "GET /services/service_id/" used to read all data about one specific service available on encompassed IoT Network. It returns a JSON Object (with attributes id, friendly name, description, service type and service provider as device id) representing the Service Session Object that has the attribute *id* equals to URI part represented by *service_id*. An example would be the request "GET /services/64/". It would return a JSON representation of the *Service Session Object* with attribute *id* equals to *64*. *Get Service* request does not allow any URI parameter.

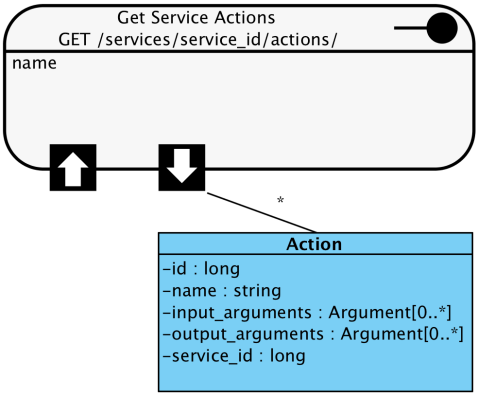

Figure 13. Discovery Request - Get Service Actions

Figure 13 shows the route "GET /services/service_id/ actions/" used to read all data about actions executable on specific service available on encompassed IoT Network, independently of which device performs the service's action. It returns an array of JSON Objects (with attributes id, name, input arguments array, output arguments array and provider service id) of found Action Session Objects belonging to service with *id* equals to URI part *service_id*. To allow specific queries, URI parameter *name* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the device's services containing received values on their related attributes. For instance, a request like "GET /services/128/actions/?name=%set%power%" would return an array of all actions, within the service with attribute *id* equal to *128*, containing attribute name with

word*set* followed by *power*. In other words, we could have access to all actions of service 128, lets say that this service is the kitchens coffee pot handler's power service, to change the power state of the coffee pot.

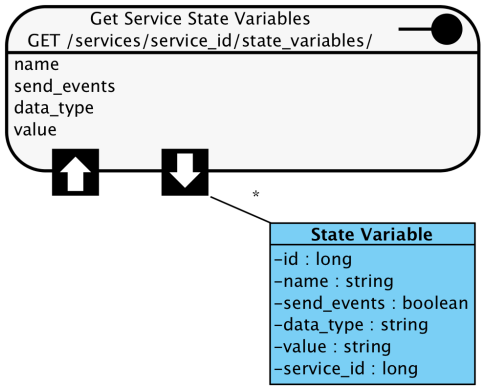

Figure 14. Discovery Request - Get Service State variables

Figure 14 shows the route "GET /services/service_id/state_variables/" used to read all data about state variables maintained by a specific service available on encompassed IoT Network, independently of which device actually has such state. It returns an array of JSON Objects (with attributes id, name, send events, data type, value and service id it belongs to) of found State Variables within Services Session Objects with *id* equals to URI part *service_id*. To allow specific queries, URI parameters *name*, *send_events*, *data_type* and *value* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the service's state variables containing received values on their related attributes. For instance, a request like "GET /services/128/state_variables/?name=%power" would return an array of all state variables , within the service with attribute *id* equal to *128*, containing attribute name with word*power*. In other words, we could have access to all states of service 128, lets say that this service is the kitchens coffee pot handler's power service, related to power state of the coffee pot.

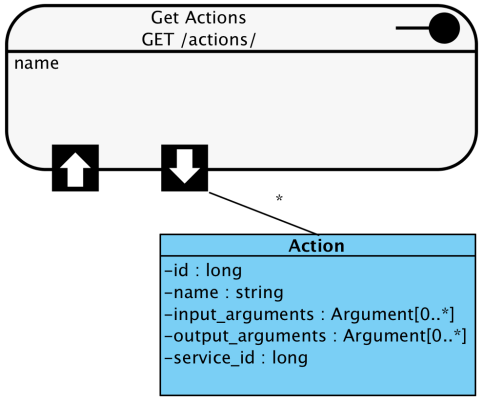

Figure 15. Discovery Request - Get Actions

Figure 15 shows the route "GET /actions/" used to read all data about actions provided by services on encompassed IoT Network. It returns an array of JSON Objects (with attributes id, name, input arguments array, output arguments array and provider service id) of found Action Session Objects. To allow specific queries, URI parameter *name* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the actions containing received values on their related attributes. For instance, a request like "GET /actions/?name[]=%start%motor%" would return an array of all actions containing *start* followed by *motor* in its name, in other words, we could have access to all actions present on the IoT Network that can start a motor.

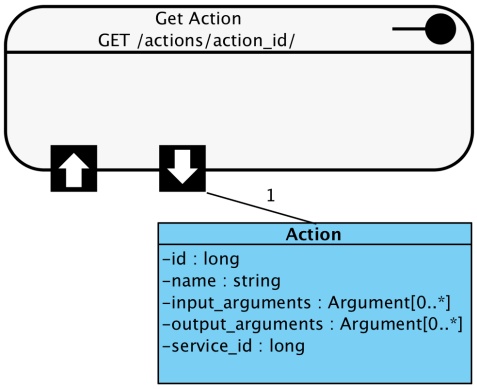

Figure 16. Discovery Request - Get Action

Figure 16 shows the route "GET /actions/action_id/" used to read all data about one specific action available on encompassed IoT Network. It returns a JSON Object (with attributes id, name, input arguments array, output arguments array and provider service id) representing the Action Session Object that has the attribute *id* equals to URI part represented by *action_id*. An example would be the request "GET /actions/256/". It would return a JSON representation of the *Action Session Object* with attribute *id* equals to *256*. *Get Action* request does not allow any URI parameter.

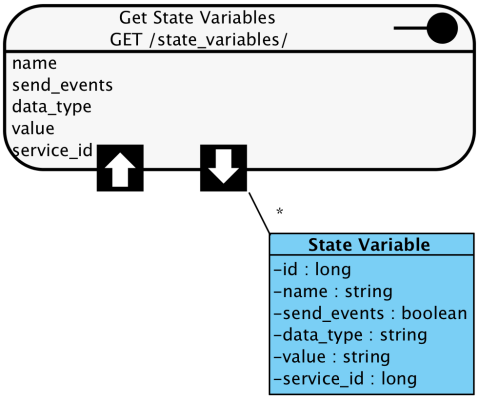

Figure 17. Discovery Request - Get State Variables

Figure 17 shows the route "GET /state_variables/" used to read all data about state variables maintained by services

available on encompassed IoT Network, independently of which device actually has such state. It returns an array of JSON Objects (with attributes id, name, send events, data type, value and service id it belongs to) of found State Variables within Services Session Objects. To allow specific queries, URI parameters *name*, *send_events*, *data_type*, *value* and *service_id* can be sent, optionally using wildcard % and/or array operand [], with desired query values to return customized list of the service's state variables containing received values on their related attributes. For instance, a request like "GET /state_variables/?service_id[]=512&service_id[]=1024&name=%power%" would return an array of all state variables containing *power* in its name and belonging to services with id equal to 512 or 1024. In other words, we could have access to state *power* of devices provider of services 512 and 1024 in one request.

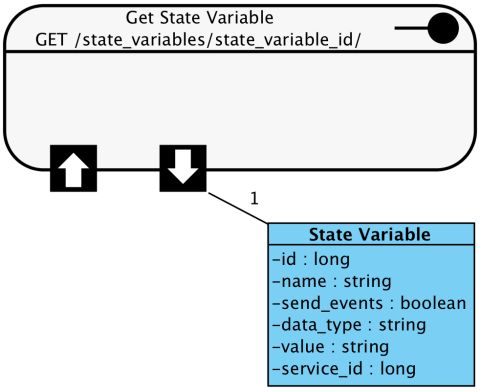

Figure 18. Discovery Request - Get State Variable

Figure 18 shows the route "GET /state_variables/state_variable_id/" used to read all data about one specific state variable maintained by services available on encompassed IoT Network. It returns a JSON 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, that has the attribute *id* equals to URI part represented by *state_variable_id*. An example would be the request "GET /state_variables/2048/". It would return a JSON representation of the State variable, resided in a *Service Session Object*, with attribute *id* equals to *2048*. *Get State Variable* request does not allow any URI parameter.

To learn about IoT networks' capacities we can start querying about available controllers, then get information about devices within targeted controllers, then get services of targeted devices, then get data related to actions provided by such service and, finally, read which arguments such action must receive to be performed and which arguments it will return after been performed.

2) Control Requests: The second type of REST route are destined to parse an application's Control Request to a device. When this type of request is made to RAISe, a specific internal flow is followed to send the right command and the response. When an application request reaches RR, it builds a RouteObject and send it to HM's *Actions requests handler*. It converts the request to a *Action Object* and send it to CM's corresponding *Device Controller*(DC). DC searches for desired objects within Device Session Objects and, if it does not find it there, *Device Controller* will ask for *Device Entities* of MM. MM will query specific databases, through Data helpers, to find the desired Entity and send it back to CM. CM converts it to a Session Object and runs a special method, named *ExecuteAction*, which receives as parameter the *Action Object* built in Presentation Layer. Device method *ExecuteAction* will run is responsible for communicating with the desired device and send to it the right commands to run the desired action. After this method is finished, *Device Controller* builds and sends a response Object to *Actions requests handler*, which will build the final Response JSON and send it back to the Requesting application.

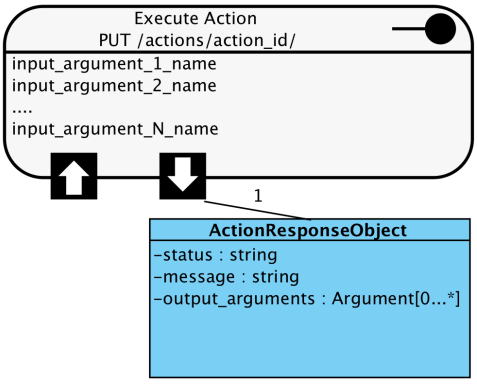

Figure 19. Control Request - Execute Action

Figure 19 shows the route "PUT /actions/action_id/" used to control devices. It receives parameters *input_argument_name*, one for each input argument needed by the Action with id equal to URI part represented by *action_id*. To represente execution Actions, 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.

In the case of the refrigerator example, section III-A, if the action named *set_temperature_to_specified_value* had id equal to *16*, it could be performed with the route "PUT /actions/16/?new_temperature_value_in_celsius_degrees=3&time_to_wait_before_changing_temperature=0".

The response JSON attribute *status* will assume values *SUCCESS* or *ERROR*, attribute *message* has open value and attribute *output_argumets* will contain only on element named *value_that_temperature_was_set_to"* which would hold the actual value of state variable *temperature*. Lets say that this state variable has the *id* equal to *8*, its value could be later checked through REST route "GET /state_variables/8/".

If an application would like to open the garage gate, it could first find the right service, using for example the discovery request "GET /services/?service_type=%garage%gate%", then find the right action, using for example the discovery request "GET /services/service_id/actions", then issuing the action,

using for example the control request "PUT /actions/action_id/?direction=open". An interesting fact of this example is that the application does not need to now the device which is performing the action. It just finds a plausible service and executes it, without knowing anything else about the IoT Network. For an efficient type like discovery request it is important to have well categorized devices and services like in UPnP's device and service type [12], USB device class [13] or Jini Services[14].

3) Subscription Requests: The third and last type of REST routes are destined for applications which would like to receive updates of services' state variables. It has two routes: the first is to add a subscriber and the second is to remove them. When this type of request is made to RAISe, a specific internal flow is followed to send data about the subscriber to the desired Session Service Object. When an application request reaches RR, it builds a RouteObject and send it to HM's *Service requests handler*. It converts the request to a *URL Object* and send it to CM's corresponding *Service Controller*(SC). SC searches for desired objects within Service Session Objects and, if it does not find it there, *Service Controller* will ask for *Service Entities* of MM. MM will query specific databases, through Data helpers, to find the desired Entity and send it back to CM. CM converts it to a Session Object and runs service's subscription special methods(*add_subscription* and *remove_subscription*) which receives as parameter the *URL Object* built in Presentation Layer. Service's method *add_subscription* creates a new subscription url on service's subscribers list. Service's method *remove_subscription* deletes an existing subscription url from service's subscribers list. After those methods are performed, *Service Controller* builds and sends a response Object to *Service requests handler*, which will build the final Response JSON and send it back to the Requesting application. The response JSON has attributes *status* and *message*. *status* assume values *SUCCESS* and *ERROR* and *message* is an open value string.

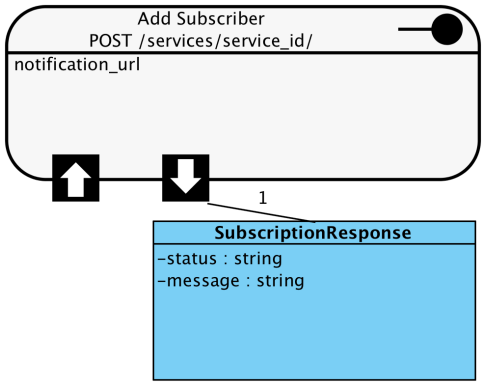

Figure 20. Subscription Request - Add Subscriber

Figure 20 shows the route "POST /services/service_id/" used to add a URL to notification system. It returns a JSON Object representing the SubscriptionResponse Object. This request adds to the Service Session Object, that has the attribute *id* equals to URI part represented by *service_id*, a new subscribing URL. An example would be 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 JSON Object representing such State Variable, the same of Figure 18, to all notification url present in its subscribers list. It issues a POST REST request, with the state variable's JSON as input RequestObject, with URI equal to notification_url parameter saved previous "POST /services/service_id/" requests. In the example of past paragraph, it would look like "POST https://192.168.0.4/receive_update".

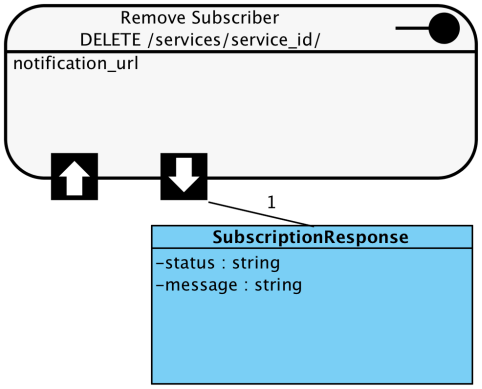

Figure 21. Subscription Request - Remove Subscriber

Figure 20 shows the route "DELETE /services/service_id/" used to remove a URL from notification system. It returns a JSON Object representing the SubscriptionResponse Object. This request removes from the Service Session Object, that has the attribute *id* equals to URI part represented by *service_id*, an existing subscribing URL. An example would be the request "DELETE /services/2/?notification_url=192.168.0.4/receive_update".

D. Security Module

The RAISe API security module (SM) was based on the proposal presented in [15], which introduce a REST security protocol to provide secure service communication. RAISe security module performs the encryption and signature on REST requests. Besides that there are other special functionalities that SM performs. Following paragraphs presents an overview about them.

In MM, Security module is responsible for maintaining data integrity. SM looks for difference between Session Objects and Data Base entities. When differences are found, Session Objects are updated.

In HM, SM ensures that requested URIs are valid and existent, and it also checks passed parameters and their values. SM escapes values, to protect following modules ,and it only allow know parameters, excluding unknown ones.

SM allows interaction with external entities only through Request Router (for RESt requests), Data helpers (to get device data) and Session Object *execute_action* method (to perform

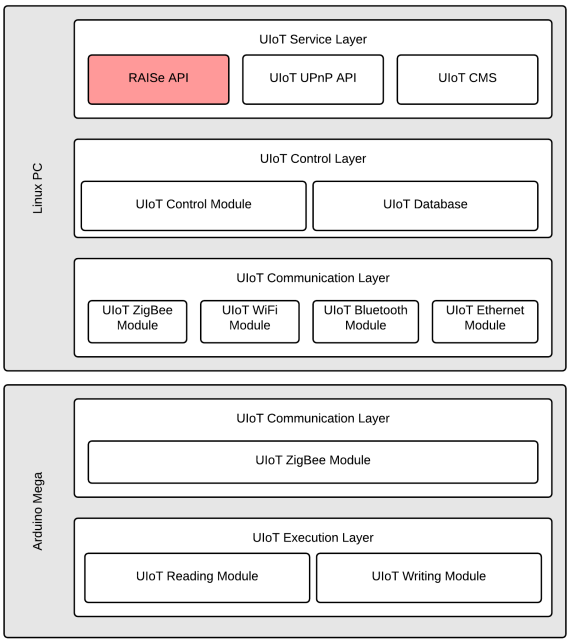

Figure 22. UIoT setup to perform RAISe tests

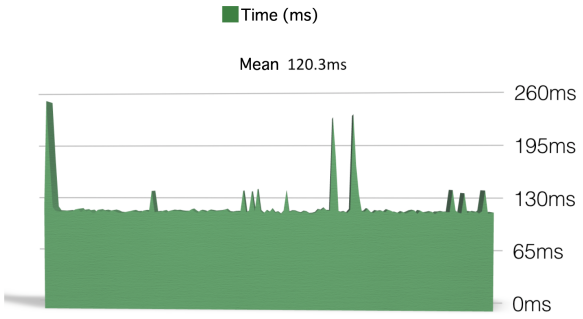

Figure 23. Time to complete REST requests during tests

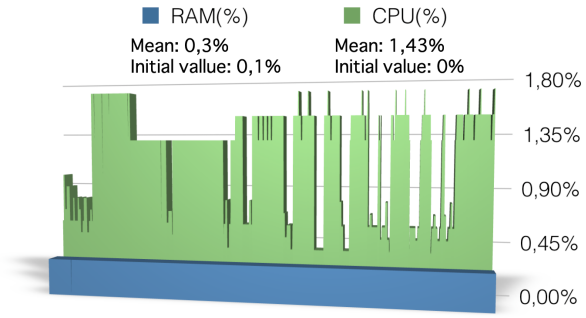

Figure 24. Apache resource usage during tests

action in devices). All internal parts are private and talk only to themselves.

SM also provides optional ACL and oAuth[16] methods like proposed in [17].

IV. Experiments and Evaluation

To test the IoT Service approach proposed by RAISe, a prototype has been developed and deployed over the UIoT Middleware. Specifically on it's Service Layer. For more information about UIoT, like how to download and install it, please visit [6].

A. The prototype

The prototype was developed using PHP programming language, memcache extension to hold Session Objects and Apache as web server. The used machine had operational system Ubuntu 12.04.5 LTS, processor quad core Intel i7 2.3GHz, 8GB of DDR3 RAM 1.600MHz and a Solid State Drive(SSD) of 256GB. UIoT Middleware setup is the same prototype described in details on [18], with Master Controller been the same PC used for RAISe and one Slave Controller being an Arduino Mega connected to a bulb light wire through a relay with circuit proposed in [19] and configuration proposed in [20] . Figure 22 shows the final setup of UIoT to receive RAISe. The software created to test RAISe is available for download, as open source code, at UIoT Middleware Website [6].

B. Experiment and Collected Results

The experiment consisted of one application making consecutive requests to turn the light on and off. In other words, the application requests the light to be turned on, when it receives the response, it request that the light get turned off. 250 consecutive requests have been monitored during 30 seconds. Figure 23 shows the time it took for a request to get completed. Figures 25 and 24 shows the amount of RAM and CPU used during tests by MySQL Database and Apache, respectively. Apache is the web server that runs RAISe's codes, representing like that the amount of resources used by proposed REST API.

It can be seen that Raise consumed around 0,2% of RAM and 1,43% of CPU, which are considerably low amounts. When the quantity of logical devices were raised from 1 to 100, it was not possible to see differences on resource usage because it have not reached the error margin to raise resource usage value in 0,01%.

V. Conclusions and Future Work

RAISe has proven to be an efficient Approach on REST API for IoT Services. It is well defined and modularized, follows success case of UPnP and allows discovery, controlling and eventing with the ease of REST APIs. It consumes low computational resources and it can be deployed as module of existent middleware and smart devices. Yet, further study must be done on an algorithm to manage amount of session objects to be kept in order to have the best performance with intelligent use of computacional resources. Besides that, it would be interesting to add an additional ontology module

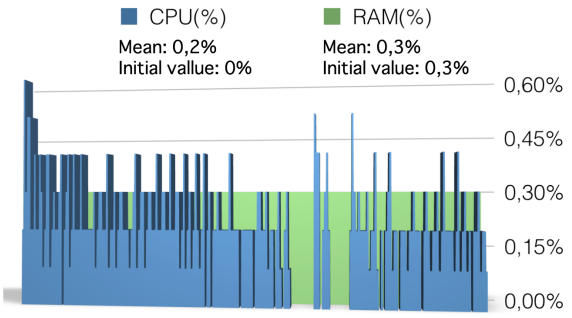

Figure 25. MySQL database resource usage during tests

on presentation layer to allow better and simpler control and discovery. Also, studies on manners to correlate actions which perform the same result to find the best device to actually perform it and balance work load to be done by each one, would be a great achievement. About tests, put RAISe to work with plenty real devices, multiple applications and multiple requests from different computers, might bring interesting new results and open doors to possible points of improvement.

VI. ACKNOLEDGMENT

The authors wish to thank CAPES Agency and the Brazilian Ministry of Planning, Budget and Management for their support to this work.

REFERENCES

[1] R. Want, "Remembering mark weiser: Chief technologist, xerox parc," *IEEE personal communications*, vol. 7, no. 1, pp. 8–10, 2000.

[2] M. Weiser, "The computer for the 21 st century," *ACM SIGMOBILE mobile computing and communications review*, vol. 3, no. 3, pp. 3–11, 1999.

[3] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, 2011.

[4] K. Ashton, "That 'internet of things' thing," *RFiD Journal*, vol. 22, no. 7, pp. 97–114, 2009.

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[6] U. Team. (2015) Uiot middleware and communication model. [Online]. Available: http://uiot.org/

[7] U. Plug and P. Forum. (2000) Understanding universal plug and play. [Online]. Available: http://www.upnp.org/download/UPNP_understandingUPNP.doc

[8] R. Fielding, "Fielding dissertation: Chapter 5: Representational state transfer (rest)," 2000.

[9] U. Plug and P. Forum. (2004) Upnp device architecture v1.0. [Online]. Available: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

[10] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[11] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] U. Plug, "Play forum," in *About the UPnP Plug adn Play Forum," in http://www. upnp. org*, 1999.

[13] J. Axelson, *USB complete: the developer's guide*. Lakeview Research, 2009.

[14] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath, *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] G. Serme, A. S. de Oliveira, J. Massiera, and Y. Roudier, "Enabling message security for restful services," in *Web Services (ICWS), 2012 IEEE 19th International Conference on*. IEEE, 2012, pp. 114–121.

[16] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[17] H. G. C. Ferreira, R. T. de Sousa, F. E. G. de Deus, and E. D. Canedo, "Proposal of a secure, deployable and transparent middleware for internet of things," in *Information Systems and Technologies (CISTI), 2014 9th Iberian Conference on*. IEEE, 2014, pp. 1–4.

[18] H. G. C. Ferreira, "Arquitetura de middleware para internet das coisas," 2014.

[19] H. Cerqueira Ferreira, E. Dias Canedo, and R. de Sousa, "Iot architecture to enable intercommunication through rest api and upnp using ip, zigbee and arduino," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*. IEEE, 2013, pp. 53–60.

[20] H. G. C. Ferreira, E. D. Canedo, and R. T. de Sousa, "A ubiquitous communication architecture integrating transparent upnp and rest apis," *International Journal of Embedded Systems*, vol. 6, no. 2, pp. 188–197, 2014.