Autonomous Composition and Execution of REST APIs for Smart Sensors

Daniela Ventura[1], Ruben Verborgh[2], Vincenzo Catania[1], and Erik Mannens[2]

[1] University of Catania - Dpt. of Electrical, Electronic and Computer Engineering
Viale A. Doria 6, 95125 Catania, Italy
{daniela.ventura,vincenzo.catania}@dieei.unict.it
[2] Ghent University - iMinds - Multimedia Lab
Gaston Crommenlaan 8, B-9050 Ledeberg-Ghent, Belgium
{ruben.verborgh,erik.mannens}@ugent.be

Abstract. Autonomous services discovery, composition and execution is an important problem in the Machine-to-Machine field. Achieving this objective requires addressing several issues: a) how to describe in a machine-understandable format which operations and functionalities an object is able to perform; b) how to represent the interfaces in unambiguous way and allow two or more machines to understand the data exchanged with each other; c) how to make a machine able to aggregate services in order to execute a specific task. Narrowing the domain just to REST APIs, we propose to semantically describe APIs (exposed by objects or web servers) using RESTdesc descriptions and to use JSON-LD as data exchange format. In order to illustrate the straightforward services composition and invocation process, we have implemented a smart client able to generate and execute plans (sequences of HTTP requests) that satisfy the set of operations which should be done for ensuring ideal environmental conditions to plants in a garden.

Keywords: Machine-to-Machine, Smart Client, Reasoning, Composition, REST API, Semantic Descriptions, Autonomous Execution

1 Introduction

Machine-to-Machine (M2M) interactions, the ability of devices to exchange data in order to execute a task not explicitly required by a human being, has become a promising domain for the next-generation communications, and is undergoing a rapid spread and development. M2M communications are expected to grow exponentially in the near future, aided by the large deployment of sensors, actuators, RFID/NFC tags. Ericsson's Media Vision 2020 research has predicted that by 2020 there will be 50 billion connected devices [8], and of these, 12 billion will be used only for machine-to-machine communications [15]. Therefore, M2M represents a huge potential business opportunity for companies and is expected to advance our lifes in a significant way, covering a broad range of vertical markets (e.g. smart homes, cars, smart factories).

The first issue is to describe what *functionalities* a device provides using a machine-interpretable format. In other words, each machine has to be able to offer semantic descriptions of its services. As smart objects are commonly equipped with actuators

that determine their states, semantic descriptions should also contain information about the behavior an object applies when its services are invoked. The service *composition* problem takes as input a set of service descriptions and a goal, and consists in generating a plan/workflow represented by a ordered sequence of services that have to be invoked in order to satisfy the goal. Finally, in order for two (or more) systems to communicate successfully, there has to be a well-defined and universal *semantics on the exchanged data*, and a way for knowing at runtime the used interfaces.

In this paper we present a method to enable machines to automatically compose and use services through standard semantic technologies. Despite the proliferation of protocols with which two or more machines could communicate (e.g. Bluetooth Low Energy, Zigbee, Xbee), we are witnessing the emergence of microcontrollers directly connected to the Internet and which can host small web servers and then provide REST APIs. According to the Web of Things (WoT), the physical world becomes thus "integrable" with traditionally Web services. Therefore, in our work, a *machine* is, first of all, a Web server that exposes a hypermedia API, which demands that a server supplied the possible next steps alongside each resource. That way, an agent does not need to know in advance how to use an API; instead, it can just follow the links at runtime through these supplied hypermedia controls [21].

To allow cooperation between machines, we propose to describe the functionalities of APIs through RESTdesc [22] and to use JSON-LD [13] as data exchange format. Entities with reasoning abilities will be able to produce plans that involve not only services exposed by physical objects but also Web services, generating what is commonly known as "physical mashups". Instead of specific tools or languages for services composition, this approach only requires generic Semantic Web reasoners.

In our early work [19], we have showed how the description format RESTdesc enables functionality-based compositions. Our major progress and contribution given in this paper consists in to: a) demonstrate, implementing really autonomous clients, how these services' mashups can be executed by a machine without human involvement; b) evaluate a complete cycle from the production to the execution of plan(s) in order to determine how it can impact in term of time spent and number of requests done from resource-constrained devices.

The present paper is structured as follows. The next Section contains a brief description on the state of art about already existing methods for describing and composing REST APIs. In Section 3, we explain the technologies we have adopted and the proposed approach. As proof of concept about the feasibility and flexibility of the method introduced in this work, a use case has been implemented on the field of smart gardens and is presented in Section 4. Section 5 contains the evaluation of our approach and Section 6 concludes the paper and provides an overview of future work.

2 Related Work

In this section, we discuss some related works about syntactic and semantic descriptions of REST APIs and present some existing approaches in services composition.

2.1 Methods to describe the syntax and semantic of REST APIs

REST-based services are still almost exclusively described by human-readable documentation, due to a lack of universal formalism. SA-REST [14], hRESTS [11] are some ways to describe REST APIs. The REST APIs' interfaces and their corresponding meaning, are typically documented in HTML pages. SA-REST leverages this common practice by annotating those pages with RDFa in order to make the information machine-interpretable. hRESTS is very similar to SA-REST. The main difference is that hRESTS uses microformats instead of RDFa. However, for a more complete overview about these and other methods, we refer to [20].

Other types of approaches to describe the functionality of Web APIs are based on the definition of specific ontologies. This is the case with the Hydra Core Vocabulary [12] for hypermedia APIs, which describes hypermedia controls (similar to HTML's `<a>`, `<link>` and `<form>` tags) in a machine-processable way. Hydra makes it possible to identify a number of concepts commonly used in Web APIs, such as links or collections of resources. By the definition of a common ontology to describe these concepts, a client can construct HTTP requests at runtime in order to manipulate the server's state. Finally, key relevant works in the service modelling prevalently for sensors are SSN model [23] and OGC's sensor Observation service [9].

2.2 Web Services Composition and Physical Mashups

The composition of Web APIs is still a task that is mainly done manually by users. Many applications allow to connect logical blocks which represent Web services or physical devices. Two of the best-known of such types of tools are IFTTT [4] and Atooma [1].

In the state of art we have found many architectures supporting creation, management and execution of service mashups directly on the Cloud. SenseStream [10] and SODIUM [17] are two examples but, unfortunately, these type of platforms are usually domain-dependent. Different solutions are presented in [6], that proposes decentralized and stateless control-flow patterns in order to support REST APIs' composition, and in [16], in which an extension of BPEL for REST services is described.

In this context, activities of standardization led by ETSI TC M2M [2] and by the Alliances (like Open Mobile Alliance [5]) are noteworthy. Although they aim is prevalently to define architectures and specifications for machine-to-machine interactions regardless of protocols, an important part under study is data exchange based on Semantic Web in order to enable machines to autonomously interpret responses received by peers and provide new APIs created combining the basic services. The biggest challenge is to reach agreements between all the member manufacturers.

3 Basic Approach and Adopted Technologies

In order to create an ecosystem of interoperable devices able to communicate each other and satisfy goals involved in an algorithm in a fully automated way, the first step is to find the technologies able to answer the following questions: a) how can a server describe the semantic meaning of its APIs and its states in machine-understandable format?; b)

Listing 1 This RESTdesc description represents a service to transit from the current on/off state to another for an irrigation pump

```
1  @prefix vocab: <http://example.org/vocab#>.
2  @prefix http: <http://www.w3.org/2011/http#>.
3  @prefix st: <http://www.mystates.org/states#>.
4  @prefix log: <http://www.w3.org/2000/10/swap/log#>.
5  @prefix bonsai: <http://lpis.csd.auth.gr/ontologies/bonsai/BOnSAI.owl#>.
6  {
7    ?actuator a vocab:IrrigationPump.
8    ?state a st:State;
9        log:includes { ?actuator vocab:hasSwitchingState ?oldValue. }.
10   ?newValue a bonsai:SwitchAction;
11       vocab:hasValue ?val.
12 }
13 =>
14 {
15   _:request http:methodName "PUT";
16             http:requestURI (?actuator ?val);
17             http:resp [http:body ?actuator].
18   [ a st:StateTransition;
19     st:typeOperation "replacement";
20     st:oldComponent { ?actuator vocab:hasSwitchingState ?oldValue. };
21     st:newComponent { ?actuator vocab:hasSwitchingState ?newValue. };
22     st:originalState ?state ].
23 }.
```

how can a client invoke an function of an API without prior knowledge about the service name and what the server expects and returns?. The answers to these questions will be described in the next subsections.

3.1 API Semantic Description

Candidate technologies, in order to describe the functionalities of the Web APIs, must take into account that real-world objects have properties that represent their physical characteristics and capabilities. The value for each property at a given time determines the object's state. For example, the defining *property* for a multicolor lamp is its color. Its *capability* is to change color when the value of this property is modified. Therefore, descriptions need to express the concepts of properties and interstate transitions.

We have selected RESTdesc [19, 22] as method to describe REST Web APIs. It expresses the semantics of Web services by pre- and postconditions in Notation3 (N3) [7] rules and, integrates existing standards and conventions such as Link headers and URI templates. An example of a RESTdesc description is presented in Listing 1. It describes the process to switch on or off an irrigation pump. The precondition is included in the lines 6-12, while le lines 14-23 are the postcondition. This RESTdesc description indicates that if a resource exists whose type is *IrrigationPump* and a new *SwitchingState* would be set, invoking a HTTP PUT request to the URI which identifies the resource plus the value of new state, is possible the transition from the precondition to the related postcondition. In particular, the postcondition expresses the meaning to replace the old status with the new. As RESTdesc is a technique based on N3 rules, every N3

reasoner can process RESTdesc descriptions and apply suitable inference rules. We have used inference rules exactly in the postcondition of the example in Listing 1. Finally, the fact that RESTdesc descriptions focus on resources and links between them, makes them an excellent way to describe hypermedia APIs and to accomplish services mashups. The use of N3, a superset of RDF 1.0, supports variables and quantification, is necessary to express such rules. N3 is compatible with RDF, and thus JSON-LD, which we discuss next.

3.2 Data Interchange Format and Services Interface

One of the main obstacles for more flexible clients are heterogeneous data format and services interface issues. Semantic technologies can mitigate the problem of interoperability and expressiveness in the exchanged data, but formats like RDF/XML, Turtle or N3 are widely disliked to APIs' developers and have not been optimized for Web APIs.

The power of JSON-LD has been to combine technologies from both the world of Web APIs and the Semantic Web in order to produce a data interchange format human/machine-understandable. Each JSON-LD message is a self-descriptive message, both data and their semantic meaning are transmitted at the same time in the same message. An important its feature is that each JSON-LD message can be converted in RDF format and viceversa. Furthermore, since JSON-LD is 100% compatible with traditional JSON, it is not *needed* to understand RDF to work with it.

Listing 2 JSON-LD response sent by a irrigation pump when it is switched on

```
1  { "@context" : {
2    "vocab" : "http://example.org/vocab#",
3    "schema" : "https://schema.org/",
4    "bonsai" : "http://lpis.csd.auth.gr/ontologies/bonsai/BOnSAI.owl#",
5    "actuatorState" : "vocab:hasSwitchingState",
6    "hasValue" : { "@id" : "vocab:hasValue", "@type" : "schema:Boolean" }
7    },
8    "@id" : "http://localhost:3300/actuators/1",
9    "@type" : "vocab:IrrigationPump",
10   "actuatorState" : { "@type" : "bonsai:SwitchAction", "hasValue" : 1 } }
```

JSON-LD requests and responses optimally fit with data represented in RESTdesc descriptions allowing a simple and straightforward execution of chains of RESTful APIs. The Listing 2 is the response received from a client when it does the request to switch on the irrigation pump identified by *http://localhost:3300/actuators/1* URL. This JSON-LD message shows that the resource is a *vocab:IrrigationPump* and has a *SwitchingState* whose value is the boolean data "true" (the meaning is that the actuator is switched on).

The conversion of this JSON-LD in to RDF will generate the Listing 3. Comparing the resulted RDF with the RESTdesc description of the Listing 1, it is possible to see the high degree of correlation and correspondence in them. In fact, RESTdesc description not only suggests that the response to the PUT HTTP request (line 17) will be an actuator resource but also contains already the information decoded in RDF (the type of resource

Listing 3 JSON-LD of Listing 2 converted in to RDF (without prefixes for brevity)

```
1 <http://localhost:3300/actuators/1> a vocab:IrrigationPump.
2 <http://localhost:3300/actuators/1> vocab:hasSwitchingState _:b0.
3 _:b0 a bonsai:SwitchAction; vocab:hasValue "1"^^<https://schema.org/Boolean>.
```

and properties). In other words, if we interpret a JSON-LD as triples, we can see it actually realizes the N3 description we have.

4 Autonomous Composition and Invocation of Hypermedia APIs

In order to illustrate the theoretical services composition and invocation process, we have implemented a use case on the field of smart gardens.

4.1 Use case

The objective of our use case is to create a smart garden system that autonomously monitors environmental conditions and decides how and when to act in order to ensure plants thrive. Real-time data about the current environment conditions can be got through sensors (light, temperature and soil moisture) directly located in the garden and associated to each plant or using the information produced by weather stations. In order to reach the ideal environment conditions, each plant is associated to one or more actuators (irrigation pumps, lamps, heaters and automated windows for greenhouses) whose status changes during the execution of the decision-algorithm. The entities involved in our use case are:

- one Internet-connected microcontroller, i.e. a **embedded board**, that runs as server and implements the Garden API[3]; sensors and actuators are directly connected to the board (that could be an Arduino Yun, STM32 Nucleo, Intel Edison, and so on). Moreover, the board stores plants' information like type, species and ideal light, temperature and soil moisture values during the different parts of day (morning, afternoon, night). Finally, we assume that the embedded board has a GPS module in order to know its geographical coordinates.
- **Weather web server** which implements the Weather Forecast API[4]; it returns the current and forecast weather conditions for the required location (expressed in geographical coordinates or city name).
- **Smart client** which is implemented as a software module[5]; it has the task to execute a decision-algorithm in order to ensure the ideal environment conditions for each plant. No pre-knowledge about what the APIs do and how invoking their services is needed, because the smart client is able to understand the APIs' functionalities and, generate and execute plans that meet the goals whose the algorithm is composed.

The client can execute four types of algorithms progressively more complete:

[3] Available: https://github.com/dventura3/irrigation-api
[4] Available: https://github.com/dventura3/weather-forecast-api
[5] Available: https://github.com/dventura3/plants-planning-agent

1. In the first algorithm, the smart client uses only the **real-time information** given by sensors to determine what status the actuators should have.
2. In case that a sensor is temporarily out of service (e.g. it is not connected to the Network) or a plant has not been associated to a certain type of sensor, the use of the **current weather conditions**, given by a weather API, can overcome the lack of sensors' data. Therefore, the second algorithm combines the current weather conditions and sensors' values to determine the actuators' states.
3. Using the forecast weather condition is a good practice to decide when to **switch on/off the irrigation system** in a garden or how to **increase or decrease temperature/light** in a greenhouse. For example, if a plant needs to be watered but it is expected that soon it will rain, it is possible to avoid turning on the irrigation pump and simply wait. Therefore, in this third algorithm, the client generates plans to know the weather forecast for the next three hours before to set the actuators' states.
4. It is the same of the third algorithm, except for the fact that the smart client uses the **forecast weather conditions for a longer time,** the next three days.

4.2 Smart Client's Architecture

Since the generation and the execution of APIs compositions is done by the smart client, we describe its architecture, highlighting its main features in the next subsections.

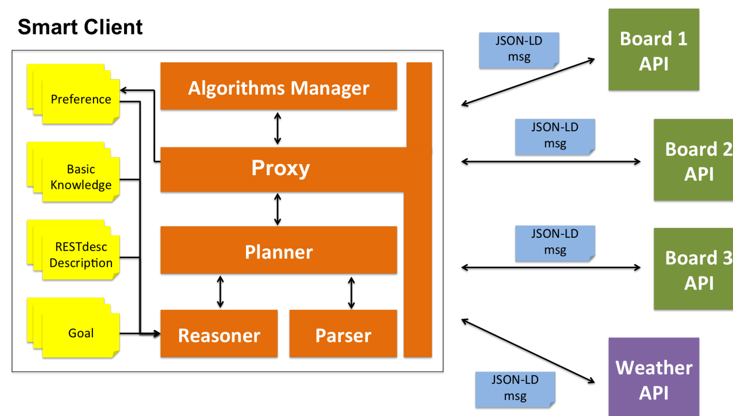

Fig. 1. The Smart Client's Architecture. Plans are generated by Reasoner and parsed by Parser. The Planner selects only one of them and manages its execution. The services invocation is done by Proxy under the supervision of Planner

The client has a list containing IP and Host for each server (board or Web service) and, during the initialization phase, uses it to get all the RESTdesc descriptions invoking HTTP OPTIONS requests. After that, the Algorithms Manager can execute one of four algorithms every 5 minutes. Whatever algorithm is chosen, it consists in a sequence of instructions that involve remote services. Therefore, the Proxy is the block that has

Listing 4 Part of output generated by Reasoner. For the sake of brevity, we report only the plan that will be really executed in order to get the sensors values for one board

```
1  p:lemma27 a c:ServiceCall.
2  p:lemma28 a c:ServiceCall.
3  p:lemma29 a c:ServiceCall.
4  p:lemma28 c:hasDependency p:lemma27.
5  p:lemma29 c:hasDependency p:lemma28.
6  p:lemma27 c:details {_:sk3_1 http:methodName "GET".
7    _:sk3_1 http:requestURI <http://127.0.0.1:3300/plants>; http:resp _:sk4_1.
8    _:sk4_1 http:body <http://127.0.0.1:3300/plants>.
9    <http://127.0.0.1:3300/plants> hydra:member _:sk5_1. _:sk5_1 a vocab:Plant}.
10 p:lemma28 c:details {_:sk36_1 http:methodName "GET".
11   _:sk36_1 http:requestURI _:sk5_2; http:resp _:sk37_1.
12   _:sk37_1 http:body _:sk5_2.
13   _:sk5_2 vocab:hasAssociatedSensors _:sk40_1; vocab:hasAssociatedActuators _:sk41_1.
14   _:sk5_2 vocab:hasIdealTemperature _:sk42_1; vocab:hasIdealMoisture _:sk43_1.
15   _:sk5_2 vocab:hasIdealLight _:sk44_1.
16   _:sk40_1 a vocab:SensorsPlantCollection; hydra:member _:sk45_3. _:sk45_3 a vocab:Sensor.
17   _:sk41_1 a vocab:ActuatorsPlantCollection; hydra:member _:sk46_1.
18   _:sk46_1 a vocab:Actuator; vocab:hasSwitchingState _:sk47_1.
19   _:sk42_1 a <http://dbpedia.org/resource/Temperature>.
20   _:sk43_1 a <http://dbpedia.org/resource/Moisture>.
21   _:sk44_1 a <http://dbpedia.org/resource/Light>}.
22 p:lemma29 c:details {_:sk104_1 http:methodName "GET".
23   _:sk104_1 http:requestURI _:sk45_4; http:resp _:sk105_1.
24   _:sk105_1 http:body _:sk45_4. _:sk45_4 vocab:madeObservation _:sk108_1.
25   _:sk108_1 vocab:hasTimestamp _:sk109_1; vocab:outputObservation _:sk110_1}.
```

to select/generate the goal associated to each specific service required by Algorithms Manager and initialize the Planner. The latter executes the Reasoner which produces all the possible plans that achieve the goal. These plans are transformed in a machine-readable format by the Parser so that the Planner can determine that one to execute. The plan execution consists in to invoke the services in order and use the JSON-LD data of X response as input of X+1 request until the end of the plan. When the whole plan is terminated, the final result is processed by the Proxy and communicated to the Algorithms Manager, so that the execution of the algorithm can progress.

4.3 Reasoning and Parsing

All the four algoritmhs include the following goal: "*Get sensors values associated with each plant monitored by each embedded board known by the client*". In order to solve this statement, the first operation to do is to find all the possible plans that fulfil the goal. This is the Reasoner's task. As defined in a previous work [19], a plan is a chain of implications that starts from an Initial state and leads to entail the Goal state:

$$I \Rightarrow S_1 \Rightarrow S_2 \Rightarrow ... \Rightarrow S_x \Rightarrow G$$

Of course, the way to meet the goal may not be unique, so many chains of dependency (i.e. plans) can be found. As the RESTdesc descriptions are N3 descriptions, any N3 reasoner can generate these plans. We have used EYE [18]. The input information, used

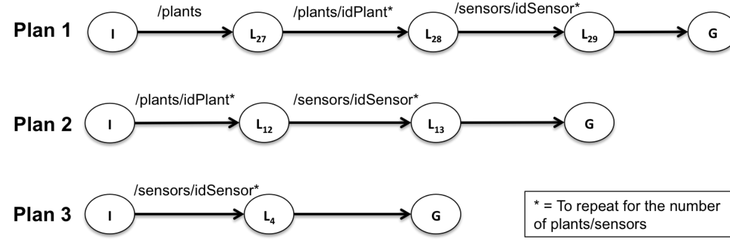

Fig. 2. All the three plans that could satisfy the goal about getting sensors values for each plant of each embedded board. The selected plan is the longest, i.e. the first

by Reasoner, is: 1) the RESTdesc descriptions associated to one embedded board at time and all remote web servers; 2) basic knowledge, that are RDF metadata about the resources exposed by each server or embedded board and, is used to infer knowledge; 3) preference, usually is something that we want to set. For example, if we want to switch on/off an actuator, this information has to be converted in N3 format as input for the reasoning process (in the above example we don't need any preference); 4) the goal expressed as N3 rule. After a pre-parsing operation, part of output generated by Reasoner, in order to solve the goal of the above example, is showed in the Listing 4. The lines 1-5 contain the dependency among the different lemmas. The details for each lemma are described in the remaining lines. The *lemma27* is the initial lemma and we already know its URI (it is a basic knowledge). The URIs for the other lemmas (lines 11 and 23) will be found only at plan's execution time. Therefore, at the moment, we know only the order in which the lemmas have to be run, namely the following plan:

$$I \Rightarrow lemma27 \Rightarrow lemma28 \Rightarrow lemma29 \Rightarrow G$$

In order to select one plan and execute the composition, the plans must be understood by client. The N3 format presented in the Listing 4 is not easy to use for a machine/software. Therefore, the Parser converts the output generated by Reasoner in JSON object properly formatted.

4.4 Plan Selection and Execution

The Planner has the task to select and manage the plan execution. Using the example in the previous subsection, the Planner should decide one of the three plans shown in the Figure 2. As previously said, only the first column of URIs, used to pass from Initial state to the next state, are known to the Parser, while we reveal the URIs of inner states just for descriptive purpose. Lacking a mechanism of cache server (we have not implemented yet) and taking into account the third plan does not have any reference to the associated plant(s), we have decided to always select the longest plan, i.e. the first. Although more complex, it is more reliable.

In order to execute the plan, the Planner needs to know all details of what each HTTP request to those APIs should like. Fortunately, this information is contained in RESTdesc descriptions and is kept by Parser. The execution starts from the first lemma and the

result is used to find the URI(s) in order to execute the next lemma. For this scope, a bottom-up algorithm was implemented. To understand how it works, look at the Listing 4. Suppose we have to find the *requestURI* of *lemma28*. The blank node associated to the *requestURI* is *_sk5_2*. The first part, i.e. “sk5”, identifies the blank node in the *lemma27* from which extracting the URI that we are searching. This blank node is *_sk5_1*, a Plant. We can’t stop here, but we have to go up until the URI of *lemma27* because we have to know if *_sk5_1* is only one or are many plants. In other words, we have to understand if we are searching just one or many URIs, because one (or many) of visited nodes could be a collection. This is exactly what happens in our example. Infact *_sk5_1* is member (the semantic property is *hydra:member*) of the collection *http://127.0.0.1:3300/plants*. As demostrate in the Section 3, RESTdesc descriptions are strictly correlated with the JSON-LD responses returned by server. In fact, the *http://127.0.0.1:3300/plants* call returns a JSON-LD with a *vocab:PlantsCollection* whose members (the property is *hydra:member*) are the URIs (of type *vocab:Plant*) we are searching. We use them to advance in the execution of the plan. Similar considerations can be done to find the *requestURI* of *lemma29* and complete the plan.

5 Evaluation

In order to evaluate our approach, we have measured the average time spent by the Smart Client’s blocks to generate and execute each of the four algorithms. The simulations have been conducted using a Raspberry Pi Model B+ with processor Broadcom SoC at 700 MHz and 512 MB of RAM. The results are shown in the Table 5. The measurements have been split in two operations: a) reasoning and parsing (to know the time used to produce all the possible plans that achieve the correspondent goal); b) selection and execution (involving the only plan that is really executed). The simulations have been conducted with two different configurations: a) one plant, one sensor and one actuator; b) three plants, each of them has associated the same three sensors and actuators.

Table 1. Time (expressed in milliseconds) spent by EYE reasoner to generate and parse all the plans and to select and execute the only final plan that accomplish each goal for each algorithm considering two configurations: A (smaller dataset) and B (bigger dataset).

Tasks		**Algorithm 1**		**Algorithm 2**		**Algorithm 3**		**Algorithm 4**	
Goal	**Operation**	**A (ms)**	**B (ms)**	**A (ms)**	**B (ms)**	**A (ms)**	**B (ms)**	**A (ms)**	**B (ms)**
Get Plants	Reas. + Pars.	361	366	426	417	403	408	394	399
	Selec. + Exec.	33	79	46	96	50	85	47	83
Get Sensors	Reas. + Pars.	380	387	465	459	401	420	413	420
	Selec. + Exec.	35	80	49	104	42	93	50	94
Get Actuators	Reas. + Pars.	–	–	572	560	489	504	482	498
	Selec. + Exec.	–	–	35	98	38	87	39	98
Get Current Weather	Reas. + Pars.	–	–	464	470	552	514	467	501
	Selec. + Exec.	–	–	657	702	776	785	837	799
Get Current Weather	Reas. + Pars.	–	–	–	–	451	456	500	459
	Selec. + Exec.	–	–	–	–	835	850	805	865

The number of plants, sensors or actuators for each board does not influence the Reasoner and Parser's tasks. This happens because their work is strictly correlated with the number of APIs known to system and not with the data received when the APIs are invoked. In other words if the number of APIs involved in to achieve a goal increases, also the time spent to generate all the possible plans will increase linearly, as demonstrated in our previous work [19]. In contrast, the number of results returned by each HTTP request, done at each step of selected plan, influences greatly the time spent to complete the plan execution. While the operations to read the sensors values on the board are simulated (that means the time measured is not affected by the time required to physically reading each sensor), our implementation of Weather API represents a JSON-LD wrapper for the forecast.io [3] web service and therefore is affected by network delays. This shows how, in a real-world setting, the time spent to generate the plan will be less than the time used to execute the plan. As expected the first algorithm is the fastest (less than one second) because it is the easiest but less accurate. On the contrary, the time spent to complete the execution of more sophisticated algorithms (as 3-4) is approximately five seconds.

An other important aspect is about the number of requests necessary to accomplish a plan. Note that selecting the longer plan has often as consequence to replicate HTTP requests already executed. For example, all the three plans used to get information about the ideal conditions of the plants or sensor values or actuators states associated with each plant, need to invoke: 1) */plants* and then 2) */plants/idPlant*. These two requests are done three times for the three different plans. A more accurate implementation could take account of historical requests. The alternative, as already suggested, is to use a cache server mechanism and choose not to perform the longer plan. Moreover, in other use cases, information as energy consumption could be used to choose the best plan.

6 Conclusion

In this paper, we presented an approach to enable machines to compose and execute REST Web APIs in a fully autonomous way. Our proposed is built on using RESTdesc as method to describe the functionalities of services exposed by physical objects or web servers, and JSON-LD as data exchange format. A machine or software agent can generate plans involving REST Web APIs in order to achieve a goal simply using standard Semantic Web reasoners. Implementing a smart client able to run algorithms that ensure ideal environmental conditions to the plants in a garden, we have demonstrated that these two technologies are deeply correlated and permit a straightforward plans execution. Furthermore, the approach we followed has threefold benefits. First, it achieves an improved decoupling between client and server. If the server's API is updated and the name or the order of parameters for invoking the API change, the code on the client side is not affected by these changes. Second, if a server becomes (temporarily) unavailable, the client may perform an alternative plan for getting to accomplish the goals. Third, the same algorithm could be run differently at different times by implementing policies to choose the more convenient plan depending on the context or the purpose of the use case. As future work, we would like to generalize the implementation associated with the use case, so as to be reusable in other Machine-to-Machine scenarios.

References

1. Atooma: A touch of magic, http://www.atooma.com/
2. ETCI-m2m, http://www.etsi.org/technologies-clusters/technologies/m2m
3. Forececast.io, http://forecast.io
4. IFTTT: Connect the apps you love, https://ifttt.com/
5. Open mobile alliance, http://openmobilealliance.org
6. Bellido, J., Alarcón, R., Pautasso, C.: Control-flow patterns for decentralized restful service composition. ACM Transactions on the Web (TWEB) (2013)
7. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax, http://www.w3.org/TeamSubmission/n3/
8. Ericsson, w.p.: Media Vision 2020: A Vision of the Television Future, http://www.csimagazine.com/csi/whitepapers/MediaVision-Brochure-RevA.pdf, (2014)
9. Janowicz, K., Bröring, A., Stasch, C., Everding, T.: Towards Meaningful URIs for Linked Sensor Data. In: Towards Digital Earth: Search, Discover and Share Geospatial Data. Workshop at Future Internet Symposium 2010 (2010)
10. Kenda, K., Fortuna, C., Moraru, A., Mladenić, D., Fortuna, B., Grobelnik, M.: Mashups for the web of things. In: Semantic Mashups, pp. 145–169 (2013)
11. Kopecky, J., Gomadam, K., Vitvar, T.: hRESTS: An HTML Microformat for Describing RESTful Web Services. In: Web Intelligence and Intelligent Agent Technology (2008)
12. Lanthaler, M., Guetl, C.: Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In: Linked Data on the Web Workshop (2013)
13. Lanthaler, M., Gütl, C.: On Using JSON-LD to Create Evolvable RESTful Services. In: Proceedings of the Third International Workshop on RESTful Design (2012)
14. Lathem, J., Gomadam, K., Sheth, A.: SA-REST and (S)mashups : Adding Semantics to RESTful Services. In: International Conference on Semantic Computing (2007)
15. MachinaResearch: M2M connections to hit 18 billion in 2022, generating USD1.3 trillion revenue, https://machinaresearch.com/static/media/uploads/machina_research_press_release_-_m2m_global_forecast_&_analysis_2012-22_dec13.pdf, (2013)
16. Pautasso, C.: {RESTful} web service composition with {BPEL} for {REST}. Data and Knowledge Engineering (2009)
17. Tsalgatidou, A., Athanasopoulos, G., Pantazoglou, M., Pautasso, C., Heinis, T., Grønmo, R., Hoff, H., Berre, A.J., Glittum, M., Topouzidou, S.: Developing scientific workflows from heterogeneous services. SIGMOD Rec. 35 (2006)
18. Verborgh, R., De Roo, J.: Drawing conclusions from linked data on the web. IEEE Software 32(5), 23–27 (May 2015)
19. Verborgh, R., Haerinck, V., Steiner, T., Van Deursen, D., Van Hoecke, S., De Roo, J., Van de Walle, R., Gabarró Vallés, J.: Functional composition of sensor Web APIs. In: Proceedings of the 5th International Workshop on Semantic Sensor Networks (Nov 2012)
20. Verborgh, R., Harth, A., Maleshkova, M., Stadtmüller, S., Steiner, T., Taheriyan, M., Van de Walle, R.: Survey of Semantic Description of REST APIs. In: REST: Advanced Research Topics and Practical Applications, pp. 69–89. Springer New York (2014)
21. Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró Vallés, J., Van de Walle, R.: Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In: Proceedings of the Third International Workshop on RESTful Design. pp. 33–40 (Apr 2012)
22. Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., Gabarró Vallés, J.: Capturing the functionality of Web services with functional descriptions. Multimedia Tools and Applications 64(2), 365–387 (May 2013)
23. Wang, W., De, S., Cassar, G., Moessner, K.: Knowledge representation in the internet of things: Semantic modelling and its applications. Automatika - Journal for Control, Measurement, Electronics, Computing and Communications (2013)