OCCI-IOT: an API to deploy and operate an IoT infrastructure

Augusto Ciuffoletti *Member, IEEE,*

***Abstract*—The infrastructure that supports the IoT is a critical component and major ICT industries are ready to propose their solutions. This paper is an attempt to define an approach based on the principles of openness and expandability.**

Our proposal encompasses both the application level interface of the IoT infrastructure, which allows interaction with external applications, and the interface offered by the single components of the infrastructure, which allows interaction and coordination between the components. In both cases we adopt a REST approach using HTTP for communication.

Our purpose is to contribute to the production of a standard for IoT infrastructures and components, to foster an open competition in a fast growing market.

***Index Terms*—Internet of Things, OCCI, REST, WebSocket, Edge Computing.**

I. Introduction

With the advent of the Internet of Things (IoT), a number of devices that operate mainly without human intervention gain access to networking technologies and infrastructures. Such devices produce data that, broadly speaking, are used to improve the quality of life, from agriculture to traffic lights control.

This opportunity extends the Internet in a new direction, since such new devices do not have the features traditionally associated with Internet end-users. Namely, they are not associated with a human user, they generate data at a steady rate, they produce and consume asynchronous signals, they are grouped and organized to contribute to a task.

The independence from a human user induces the proliferation of networked *Things* — Gartner envisions billions of such devices in 2020 [14] — the control of which is a design challenge. In fact, such devices need to be connected to data repositories, their signals must be managed, and appropriate controls must be distributed.

This paper addresses a framework to define and control an IoT infrastructure with three fundamental targets in mind:

- **simplicity**, meaning that we minimize the number and complexity of the types of basic devices that compose the infrastructure
- **openness**, meaning that the tools used to deploy the infrastructure are standard or promote standardization
- **scalability**, meaning that the infrastructure grows linearly with the number of endpoints

The three targets are not each other in contrast, and are often found together in projects that share a similar conceptual approach. In this sense, we take advantage of a framework proposed by the NIST [19] that addresses the definition of terms and concepts that apply to IoT infrastructures. We analyze their proposal, and adhere to basic concepts and terminology.

Since the final purpose of IoT is not far from that of distributed systems monitoring, we take advantage of results in this latter field, and we also point out some issues shared with software defined networks, that introduce dynamic configuration features that are mandatory for an IoT infrastructure.

The proof-of-concept prototype described in the paper follows the same principles: it is easily reproducible and focused on the relevant issues.

Augusto Ciuffoletti is with the University of Pisa - Dept. of Computer Science

II. A framework for IoT infrastructures

The presence of a complex, dynamic infrastructure is not a property of IoT systems only.

Consider the analogies with the monitoring of distributed computing systems, for instance a datacentre with thousands of cloud servers, grouped in racks and rooms: a pervasive monitoring infrastructure is needed to collect data ranging form disk errors to room temperature. Like in an IoT infrastructure, sensing devices pervade the system, and are at the edge of the monitoring infrastructure.

Nagios [10] is a monitoring system addressing this task. One of its distinguishing feature is the partitioning of the system into smaller subsystems, managed by specialized devices that control monitoring applications installed on cloud servers. The need to co-exist with legacy installations introduces a design issue that limits Nagios scalability: we recently proposed [2] the OCCI-*Monitoring* framework, that adheres to Nagios principles without inheriting its constraints.

The front end API of the OCCI-*Monitoring* service offers the tools to define the architecture of the monitoring infrastructure. It is designed according with an open standard — the Open Cloud Computing Interface (OCCI) of the Open Grid Forum (OGF) [13] — that contains the basic tools to describe a complex infrastructure framed into a REST interface [6]. So that OCCI-*Monitoring* specifies both how to *describe* the infrastructure (schemas), and how to *upload* such information to the deployment engine (REST). The OCCI-*Monitoring* schema introduces two sub-types of the core OCCI types: one is for the objects that represent **measurement tools** that monitor the operation of a cloud resource (for instance, the number of connections on a web server), the other sub-type is for the objects that **process** the data obtained from the former (for instance, a load balancer)

name	short description
RQ1	Object coordination
RQ2	Object virtualization
RQ3	Dynamic object join/leave
RQ4	Dynamic object state change
RQ5	Coordination of external objects and services
RQ6	Object description
RQ7	Scalability
RQ8	Security

TABLE I
REQUIREMENTS ACCORDING WITH [18]

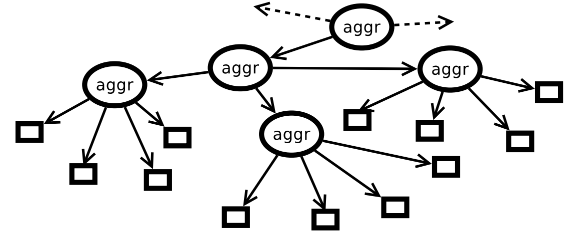

Fig. 1. A multi-layer IoT system, with 5 aggregators (ovals), 12 real objects (boxes), and 16(+2) sensors (arrows)

The above schema adheres to the conceptual framework proposed by NIST for IoT in [19], a summary of the principles emerging from the preceding literature on the topic (e.g. [9]). Five conceptual entities are defined — called *primitives* — that contribute to define an IoT system:

- a *Sensor* is the source of measurements and data from the environment; sensors are on the edge of the system.
- an *Aggregator* is the intermediate device that processes data coming from *sensors*: such processing usually reduces the amount of data forwarded by the *aggregator* and improves scalability
- a *Communication Channel* is the building block for the fabric that connects sensors and aggregators
- an *eUtility* is the device that provides computing capabilities or conveys data to the system without being a sensor or an aggregator
- a *Decision trigger* represents an action performed by the system together with the criteria used to decide when to perform such action

Both the OCCI-*Monitoring* schema and the NIST IoT model postulate the presence of a multi-layer architecture where the bottom layer is populated with *sensors* that perform measurements. The rest of the infrastructure is a hierarchy of *aggregators* that process and propagate data. In this paper we merge the two approaches: define an OCCI schema that complies with the NIST model for IoT.

Besides abstract principles and terminology, functional requirements are also relevant for the implementation. In paper [18] we find a precise statement that we use as a guideline. The authors introduce eight fundamental requirements for an IoT infrastructure, that are condensed in table I. In their terminology, the *object* refers to a generic entity, either a *sensor* or an *aggregator*.

III. THE ARCHITECTURAL MODEL

We want to model a complex distributed system composed of IoT devices. The system interacts with the real world on one side (the *back-end*), and provides a representation of the real-world on the other (the *front-end*). The purpose is monitoring and control of the real world, according with a closed-loop pattern. Therefore the IoT infrastructure supports bi-directional communication (*upstream* and *downstream*), although the contents transferred in the two directions are quite different: a continuous stream of measurements in the *upstream* direction, sporadic commands *downstream*.

According with NIST terminology, we introduce *Sensors* as the agents bordering the back-end of the system. They perform measurements, receive controls and act accordingly. In the simplest case, such controls are directed to the measurement function, for instance to switch off a power-consuming measurement, but they can be extended to an associated device, e.g. to control a traffic light. In the NIST model, the *Sensors* are the targets of actions triggered by *Decision Triggers*.

In our model, we qualify the *Sensors* with the following features:

- they are many, and they produce an amount of information that cannot be managed without aggregation techniques
- they are dynamic, in number and function: new sensors may dynamically enter the system, and may later change their operation
- they can be configured by an external agent, in particular by other agents in the IoT infrastructure
- they can operate on the environment both autonomously and responding to external controls

A *Sensor* is physically close to the real world entity that it measures and acts upon: for instance, a light sensor associated with a road needs to *see* it. Entities like a road or a traffic light — together with their properties — are included in our schema as *real-world entities*.

The *Sensor* delivers measurements to and receives commands from another entity that is responsible of data processing and forwarding; following NIST terminology, we call this component *Aggregator*. An *Aggregator* is associated with one or more *Sensors*.

We qualify an *Aggregator* with the following features:

- it implements *Decision triggers* that process *upstream* data to produce *downstream* controls
- it processes and forwards the *upstream* data flow
- it processes and forwards the *downstream* controls

Aggregators can form a chain, and this is fundamental for scalability. To cope with a growing numbers of Real World Objects the output from *sensors* cannot be used directly, but has to be processed using *big data* techniques. A layer of aggregators may turn multiple streams of data into sporadic notifications (see figure 1), according with *edge computing* principles [16].

Although a *hierarchy of aggregators* introduces communication delays that grow with the number of layers, it may equally reduce the response time, since intermediate *aggregators* may detect relevant events and produce an early response. Overall, we consider that layering responds to the mandatory requirement of scalability, stated as RQ7 in table I.

Aggregators that are located deep into the IoT infrastructure may be implemented as virtualized cloud components — towards an architecture based on micro-services — to enhance the flexibility of the system: this meets RQ1 in table I.

We now translate this informal insight into a formal schema, starting from its notation. This result enables the description and control of the components of an IoT infrastructure (RQ6 and RQ1 in table I) using a REST interface.

IV. An OCCI schema for IoT infrastructures

The OCCI specification is based on a *core* schema, and supports an *extensions* mechanism.

The *core* OCCI schema describes two kinds of *core* entities: the *resource* and the *link*. A *link* is associated with two *resources*: the *source* and the *target* resource. Each entity has an URI associated with it, and the user is provided with a REST-ful interface to interact with the entities on behalf of the management system. A given entity has defined *attributes*, whose values characterize a specific instance.

Another category is used to define the *action*, that stands for ability to perform an activity, often supported by the abstraction of an internal state of the entity.

To customize a given entity instance, OCCI introduces the *mixin*. The association of one of them with an entity instance adds new *attributes* and *actions*, that complement those already present in the entity instance. From a commercial point of view the mixins have the important role of differentiating the offer of distinct providers: entity types are the common denominator that allows basic portability, while mixins are provider-specific.

Given its simplicity, the OCCI interface can be adapted to a number of distinct environments: the description of IaaS cloud resources — historically the first application of OCCI —, Service Level Agreement, PaaS resources, Cloud Monitoring infrastructures and more.

The procedure to apply the *core* model to a specific purpose consists of writing a document that defines new sub-types of the core entities with appropriate features. In OCCI terminology such document is an *extension* of the *core* schema.

A. An OCCI *extension for IoT*

According to the informal model explained in section III, a *sensor* monitors a *real world object* on behalf of an *aggregator*. In OCCI terms, both the *aggregator* and the *real world* entity are OCCI *resources*, and the *sensor* is an OCCI *link*. The definition allows *aggregator* hierarchies of unlimited depth, since the aggregator is a subtype of the core *resource* type, and a *sensor* is a link between *resources*, as in figure 1.

The class diagram is in figure 2: we do not introduce specific attributes for the new OCCI entities, but we expect that appropriate mixins are defined by the provider to allow to tailor an entity to user's needs (for instance, to measure the temperature in a greenhouse as in table II). The user obtains the description of the available mixins in response to a GET sent to the OCCI server: the available mixins are therefore discoverable. By associating a *sensor* instance with

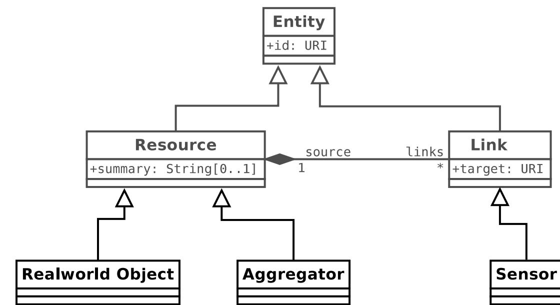

Fig. 2. The UML class diagram of the IOT extension (core model light-gray)

selected mixins the user implements a *sensor* with the required functionality.

For instance, in table IV we see a sensor rendering embedded inside an aggregator: the *link* attribute is an array with the `ntc` sensor as the unique element. The sensor has an NTC mixin associated, and inherits from it two attributes: `period` and `OUTlevel`, an *entity port*.

The NIST document postulates the existence of communication channels: in our schema they are represented by *entity ports*, like `OUTlevel` in the latter example. A *port* is a string attribute whose value is the identifier of a bi-directional communication channel: ports with matching identifiers share the same channel.

The capability of a sensor to communicate with the next upper layer is represented with a *null* mixin that connects one port of the low level aggregator with a port in the upper layer aggregator.

V. Orchestrating an IoT infrastructure

The OCCI-IOT introduced above represents an IoT according with REST principles: namely, each entity has a representation that is reachable with an URI that contains links to other URIs, the representation can be managed using REST operations, and the types of entities too are discoverable and manageable using the same paradigm. By definition this meets RQ4 of table I. The schema defines an interface, the specifications of which are an input for the design of a management application, e.g. a graphical dashboard.

Now we want to step on the other side of the interface, and see the architecture of the *engine* that deploys the IoT infrastructure using its definition. Since its interface is REST, the use of the same paradigm for the implementation of the *engine* is consequential. Although the REST paradigm is independent from the communication protocol, in practice it is strongly related to, and motivated by, the HTTP protocol and the web infrastructure.

The web infrastructure that we use to support an IoT has the well known advantages of an established technology. For one, the use of HTTPS (when the involved devices are sufficiently powerful) solves a lot of security issues [17], and meets RQ8 in I. However HTTP is unsuitable for *data streaming*, especially following the REST paradigm, since all interactions are strictly request-response between a client and a server.

Several techniques have been introduced to overcome these limits: Java RMI, SOAP, AJAX, although different in concept,

all go in that direction. The problem with these techniques is that they alter the strict REST behavior to change operation and control. Instead, the WebSocket technology is considered as REST-compliant and implements a bidirectional, unrestricted communication (in TCP Socket style, as suggested by the name) within an HTTP session. The switch to the WebSocket protocol is implemented with an HTTP *upgrade* request. See [5] and [7] for further details.

A WebSocket is an asynchronous, bidirectional channel suitable for sensor-to-aggregator communication, where the sensor produces an *upstream* flow of raw measurements for the aggregator, and control messages are delivered *downstream*. It is sufficiently lightweight to be implemented by IoT devices, as we will demonstrate in the experimental section.

The introduction of a new sensor proceeds as follows:

- physically connect the new sensor to the network
- record a new real world object (RWO) in the system with a POST to the relevant IoT engine component
- associate the new sensor device to an aggregator with another POST (as an incoming link)
- the aggregator sends a POST to the sensor, in order to configure its activity
- the aggregator turns the HTTP session with the sensor into a WebSocket
- the sensor starts feeding the aggregator with data

An aggregator-to-aggregator sensor follows the same process: specifically, when a new low-layer aggregator is introduced in the architecture, or a sensor is *handed-off* from one aggregator to another.

The adoption of the WebSocket technology makes the deployment flexible, as in RQ3 in table I. Using POST verbs the user agent configures new entities, that the *engine* deploys and connects with dynamic WebSockets.

VI. A PROTOTYPE: TEMPERATURE MONITORING

The purpose of this section is twofold. On one side, we want to demonstrate that the formal schema is sufficiently expressive to describe and guide the deployment of a simple use case. On the other, we want to compare the performance of a REST-based deployment with the deployment found in [18], in a way that is easily reproducible.

For the task we need a prototype that focuses on the critical aspects of our proposal:

- the deployment is guided by OCCI-IOT data structures
- data transport inside the infrastructure uses WebSockets

We deliberately simplify other aspects — notably, networking — to avoid the introduction of spurious variables.

The problem we want to solve with our prototype is a typical *smart agriculture* use case stated as follows:
We want deploy a IoT that gathers ambient statistics, that controls air conditioning and watering inside greenhouses. The system is distributed across a number of possibly distant plants.

The IoT system is organized on three layers. The *Real Word Objects* layer contains the greenhouses and it is linked to the upper layer by *sensors* that send raw data to the devices on the intermediate layer. This is populated with *aggregators* that convert raw data into temperature and apply a filter to obtain a robust sample. The output is passed to a ThingSpeak server — an external service, meeting RQ5 for [18] — that gathers and displays the data from all aggregators, and may compute alarms and schedule events. It is an *aggregator* in the OCCI-IOT schema.

To represent the above system we need six mixin types:

- three *sensor* mixins, **ntc** to describe the temperature meter, **null** to connect the intermediate aggregator to a centralized server, **dummy** that is not attached to a real measurement device and is used to stress the aggregator;
- two *aggregator* mixins, **NTC2Degrees** for the intermediate aggregator, **ThingSpeak** for the ThingSpeak server;
- a *real world object* mixin for the **greenhouse**.

We concentrate on two of them: the *temperature meter*, and the *intermediate* aggregator.

The *temperature meter* mixin is represented in table II. The `term` attribute reveals the use of a NTC thermistor — a cheap temperature sensor — as the input device. The mixin type definition is part of the sensor scheme defined by the provider (see the `scheme` attribute), it depends from a *sensormixins* subtype defined in a OCCI extension for IoT (`depends` attribute), and it can be associated only with a sensor link (`applies` attribute) defined in the same IoT document. The mixin attributes are prefixed with the provider domain: `com.example`.

- `period` configures the time interval between successive measurements in milliseconds
- `level_out` indicates the id of the *output channel* associated with the `OUTlevel` port: the value comes from the output of a A/D converter connected to the NTC device
- `uri` is the URI of the remote measurement device hosting a WebSocket server.


```
{"term":"NTC",
 "scheme":"http://example.com/sensor#",
 "depends":"http://schemas.ogf.org/occi/iot#sensormixins",
 "applies":"http://schemas.ogf.org/occi/iot#sensor",
 "attributes":{
  "com.example.NTC.period":{ "type":"number" },
  "com.example.NTC.level_out":{ "type":"string" },
  "com.example.NTC.uri":{ "type":"string" }
 },
 "title":"NTC",
 "location":"/sensor/NTC"}
```


TABLE II
THE *json* RENDERING OF THE TEMPERATURE SENSOR

The aggregator mixin that computes the temperature and collapses successive values is represented in table III. The r aw data received from the input channel `ntc-in` and converted into degrees using a logarithmic interpolation of the characteristic function of the NTC. Data in the history are combined using an exponentially weighted moving average and, at regular intervals of `period` seconds, the current value is forwarded across the `degrees_out` channel.

Using the above mixins it is possible to define an infrastructure with an arbitrary number of sensors: the system smoothly scale since, when the number of aggregators in the

```
{"term":"NTCtoDegrees",
 "scheme":"http://example.com/aggregator#",
 "depends":"http://schemas.ogf.org/occi/iot#
      aggregatormixins",
 "applies":"http://schemas.ogf.org/occi/iot#aggregator",
 "attributes":{
  "com.example.NTCtoDegrees.ntc_in":{"type":"string"},
  "com.example.NTCtoDegrees.degrees_out":{"type":"string"},
  "com.example.NTCtoDegrees.gain":{"type":"number"},
  "com.example.NTCtoDegrees.period":{"type":"number"}
 },
 "title":"NTCtoDegrees",
 "location":"/aggregator/NTC"}
```

TABLE III
THE *json* RENDERING OF THE AGGREGATOR MIXIN

intermediate layer exceeds the capacity of the ThingSpeak aggregator, it is possible to introduce a further layer of intermediate aggregators.

A. Implementation of a proof-of-concept prototype

The devices that implement the prototype — the *eUtilities*, according with NIST terminology — are an Arduino Duemilanove for the measurement device and a Raspberry PI for the aggregator. They are two low cost COTS devices that encourage the reproduction of our experiments. The Arduino wears an Ethernet shield.

We used a switched Ethernet network because of its minimal interference on the kind of measurements we want to perform. Wireless devices are also commonly adopted, but they would barely introduce uncertainty into experimental, without technical added value.

The Arduino implements a WebSocket server that delivers the measurements of the temperature at a fixed rate. Since timing is critical for our experiments we use low level interrupts to trigger data production. The temperature sensor is a voltage divider made of one fixed resistor and an NTC (negative temperature coefficient) resistor.

The aggregator device is hosted by a Raspberry Pi 3, a credit card-sized single-board computer based on a powerful ARM (quad core at 1.2 GHz), powered by a Linux Operating System. It is connected to the public Internet across a NAT router in order to reach the public ThingSpeak server.

The implementation is available through two distinct GitHub software repositories. The code for the Raspberry (orchestrator and aggregator) are at [3], while the Arduino sketch and electronic layout is at [4].

According to CAIDA the dashboard ThingSpeak server is hosted in the AWS autonomous system: IP location services reveal that it is in the United States. A REST API allows the user to upload numerical data; these can be displayed and used to trigger a number of actions.

Finally, in our intranet a PC is equipped with a daemon that, upon request, spawns dummy measurement devices that send data to the aggregator.

The prototype is a pipe composed of a sensor link (`ntc`), an aggregator (`temp`), another sensor link (`s1`), and another aggregator (`TS`). In figure 3 for each component we indicate the instance identifier, its OCCI-IOT type, and the associated mixins: the syntax is outlined in the caption. Each component

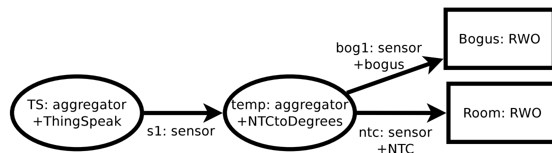

Fig. 3. Graphical representation of the system deployed in the prototype. Syntax: $< id >:< type > [+ < mixin >]*$

```
{
"kind":"http://schemas.ogf.org/occi/iot#aggregator",
"mixins": [
 "http://example.com/aggregator#NTCtoDegrees"],
 "attributes": {
  "com.example.sensor.NTCtoDegrees": {
   "ntc_in": "channel1",
   "degrees_out": "channel2",
   "gain": 16,
   "period": 30 } },
  "id": "temp",
  "links": [
  { "kind":"http://schemas.ogf.org/occi/iot#sensor",
   "mixins":["http://example.com/sensor#NTC"],
   "attributes":{
    "id":"ntc",
    "target": {
     "location":"/room/MyRoom",
     "kind":"http://schemas.ogf.org/occi/iot#rwo" },
    "source": {
     "location":"/aggregator/temp1",
     "kind":"http://schemas.ogf.org/occi/iot#aggregator"
    },
    "com.example.sensor.NTC": {
     "OUTlevel": "channel1",
     "period":1,
     "uri": "ws://192.168.113.177"}
}}]}
```

TABLE IV
OCCI-*json* RENDERING OF THE *temp* AGGREGATOR WITH THE OUTGOING *ntc* SENSOR

has a OCCI-IOT representation in JSON. The representation of the `temp` aggregator is shown in table IV. According to OCCI it embeds the outgoing `ntc` sensor link.

Such data structure is received by a *deployment engine* component that in our setup is a Sinatra/Ruby web server on the Raspberry. The server should receive the JSON in table IV as the payload of a PUT from the infrastructure orchestrator; in our setup the descriptor is preloaded as a Ruby hash (see table V). The web server spawns a new thread to implement the *temp* aggregator.

The new aggregator thread dynamically loads the library containing the requested mixin (*NTCtoDegrees*) and spawns another thread that implements the mixin functionalities, configured according to the content of the description. The mixin thread receives raw values from the input Queue, converts the values into degrees and periodically delivers a new value to the output channel. Many such threads, possibly of different kind, can run concurrently within the same host and each of them originates mixin threads.

To enforce a precise frequency in the feed, input and output operations are decoupled: to this end the aggregator spawns a thread that has access to the last computed result. Such thread periodically reads that value and sends it to the output channel.

The aggregator spawns another thread for the sensor. The functional parameters of the Arduino measurement device are

```
'NTCtoDegrees' => {
  ntc_in: channel["channel1"],
  degrees_out: channel["channel2"],
  gain: 16,
  period: 30
}
```

TABLE V
THE RENDERING OF THE *NTC2Degrees* SENSOR MIXIN AS A RUBY HASH

found in the `links` attribute of the JSON description of the aggregator (see table IV), and translated into a Ruby hash (see table V). To configure the measurement device with its functional parameters, the sensor thread opens a WebSocket to the remote measurement device, the server being hosted on the Arduino. Next the measurement device starts feeding the sensor with data, that is forwarded to the aggregator across `channel1`.

To stress the aggregator and obtain results comparable with those in [18], we introduce a dummy sensor. Here the measurement device is a thread implemented on a PC that behaves like the Arduino measurement device: a WebSocket server sends one piece of data every second. In one of our experiments we activate 19 dummy sensors, so to have 20 sensors connected to the aggregator, as in paper [18].

B. Experimental results

Just using the prototype we found an answer to the following questions:

a) is it feasible to have a web server with WebSocket capabilities on a measurement device?
b) what is its timing accuracy?
c) what is its reliability?
d) is it feasible to have a multi-threaded software organization on intermediate devices?
e) what is the impact of a single aggregator, how many of them are allowed?
f) what is the impact of virtualization?
g) how is the performance compared with [18]?

Regarding question a), we showed that it is possible to implement a web server with WebSocket capabilities on an Arduino, despite its limited capabilities. We used an open source library that implements the WebSocket server with some limitations: according with the authors, memory restrictions prevent the implementation HTTPS WebSockets, and we have noticed problems with TCP recovery after packet loss, as discussed below.

To evaluate the timing performance and have a quantitative answer to question b) we measured the timing for each of the communication links.

For the link between the NTC measurement device (Arduino) and the aggregator (Raspberry) we measured the round-trip time (RTT) and the difference between send and receive timestamps (here we call it *apparent one way delay*). In a sample of 13657 RTTs (nearly four hours of continuous operation), 13457 of them (98.5%) are between 6 and 8 msecs, and the remaining 200 are in a *long tail* that reaches the 16 msecs (see figure 4). The delay introduced by the measurement device to prepare and send the packet has been also measured, and it is significantly lower ($\leq 3msecs$) than the round-trip time, that includes Ethernet delay: even more so, we guess, in slower networks, like wireless ones.

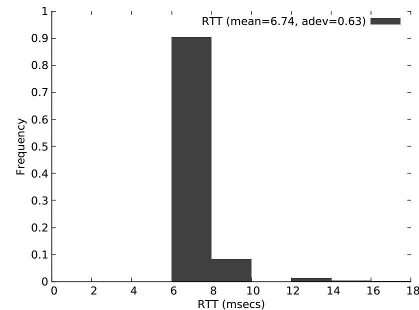

Fig. 4. WebSocket round-trip time measured from the sensor (resolution=1 msecs)

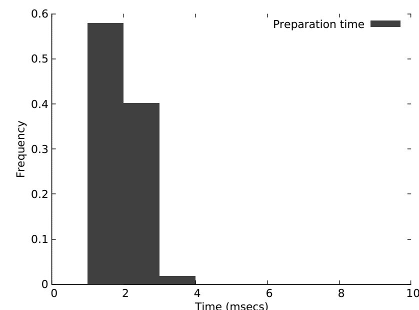

Fig. 5. Preparation for sending a WebSocket frame: includes the preparation of the digest in the header

When we display the *apparent one way delay* (in figure 6), we observe a significant clock drift ($\rho = 13 * 10^{-6}$), which might be effectively compensated exploiting this communication pattern. This result quantifies the performance of the WebSocket based communication between the two devices, but anticipates problems with a shared time reference.

The link between our aggregator and the public ThingSpeak server in the US can't be evaluated with a RTT: in figure 7 we see the *apparent one way delay* on that link. The graph shows that clocks are compensated, but the drift is nonetheless evident. Given that at time 450 they are synchronized, we argue a delay around 0.1 secs. The jitter (with a resolution limited to that of the ThingSpeak timestamp) in figure 8 is

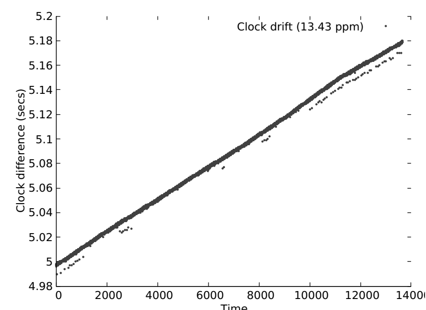

Fig. 6. Clock drift between aggregator and sensor

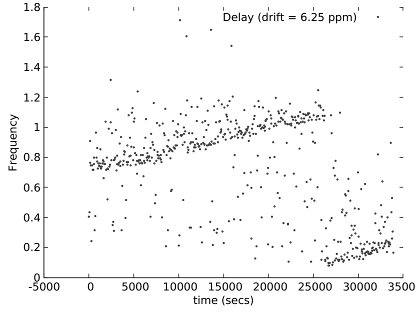

Fig. 7. Apparent delay between the aggregator and the ThingSpeak server

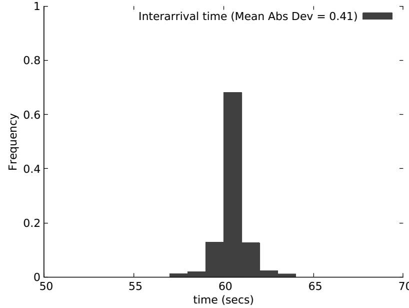

Fig. 8. Jitter of events on the ThingSpeak server (resolution 1 sec)

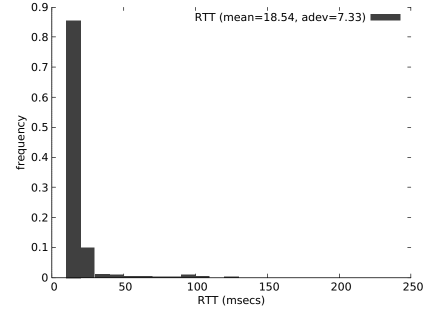

Fig. 9. Round-trip frequency distribution with twenty sensors

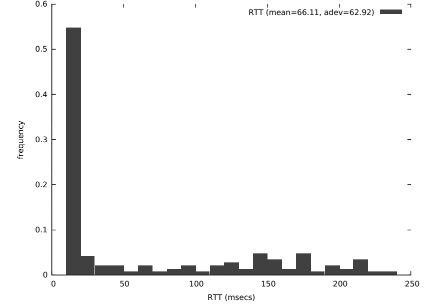

Fig. 10. Round-trip frequency distribution with twenty sensors and Dockerized aggregator

also representative of the precision of the whole pipe.

The reliability of the measurement device, question c), is an issue: after a variable number of correct operation rounds, consisting of sending the data across the WebSocket and receiving the acknowledgment, the measurement device stops sending data. By packet inspection we found that the Arduino breaks the TCP connection when a packet is lost, causing the WebSocket to hang: there is no way to compensate such problems, since the TCP/IP stack is flashed in the Ethernet shield firmware, and on this respect we obtained different results with different Ethernet shields.

Question d) is motivated by the limited processing power of the the *eUtility* we adopted for the task, a Raspberry Pi 3. With one aggregator running, we observed a footprint of 0.5 percent of memory capacity, and of the 1.5 percent on computing capacity. We conclude that the device might indeed support tens of aggregators similar to the simple one implemented for the prototype.

To precisely quantify the effects of the presence of several sensors attached to the same aggregator, we performed an experiment with 20 sensors, and measured the round-trip time: we recall that each aggregator contributes with two threads, one shared channel, and a WebSocket client connection. The round-trip frequency distribution is shown in figure 9, and can be compared with figure 4. With respect to the case of a single sensor we observe an average round-trip time three times larger, and an average deviation that is ten times larger. The difference is evident, but there is no sign of collapse.

This is an answer to question e), and shows that the communication infrastructure (WebSockets, and Queue-based channels) has a limited footprint and scales well.

To evaluate question f) we ported the aggregator from the Raspberry inside a Docker hosted by a PC in our lab, and performed the experiment with 20 sensors. The results are in 10, and show a clear degradation compared to the Raspberry implementation (figure 9). We conclude that virtualization adds flexibility but comes at a price.

VII. Discussion and related works

We share with [18] the functional requirements — in our opinion a major contribution of that paper — that we summarize in table I; along the paper we pointed out and discussed how our approach meets them. Their *Application Execution Platform* is based on Virtual Blocks that can be dynamically instantiated and may be assimilated to our Aggregators. The communication pattern addresses a Request/Done protocol, that is probably responsible for the poor performance. In a chain of 5 Virtual Blocks, with 20 requests per seconds, they report service times of the order of at least 200 msecs/stage. This result is comparable with our 18 msecs average round-trip time in the NTC link with 19 dummy sensors, and it is an answer for question g). In our opinion, the reasons for the remarkable difference are found in the adoption of the WebSocket protocol, and in the use of dedicated hardware for the devices.

In addition, the authors do not address the problem of aggregating data from many sources, which has an impact on system scalability. On the contrary, they concentrate on smart objects that access a public database. In our proposal, we indicate how real-time requirements are met moving some computation *to the edge*, which motivates a hierarchical architecture.

Another advancement of our paper with respect to [18] is in the specification of the OCCI-IOT schema to define the

infrastructure and to guide its deployment. The topic, tightly bound to RQ6, is explicitly omitted in the referenced paper.

Finally, our paper offers quantitative results obtained with open source code and a prototype implementation based on low cost COTS devices. Such factors together allow us to claim that our results are reproducible, which is mandatory for a scientific result.

We share many of the architectural principles, REST-fulness included, with [12]. The authors address a specific use case with a sensor polling mechanism, that, as we demonstrate, degrades the response time and hinders scalability. Our results have a wider applicability, and evaluate response time and scalability.

In [20] the authors introduce a three layers service oriented architecture using the front-end/back-end paradigm. They introduce different protocols for each layer: one for describing sensor devices (DDLS), one for the intermediate layer (OSGi) in a SOA. Our paper simplifies the scenario, introducing a single protocol, and preserves expandability and scalability.

Fundamental aspects of our work are dealt with managing complex systems. One that is presently receiving attention is the software description of network infrastructures, associated with the availability of network virtualization techniques (SDN/NFV) [11]. In that domain there is a strong concern for security issues [1]. Some of the techniques used for SDN systems (e.g. trusted configuration [8]) are applicable in our scenario.

Finally, we observe that there are a number of IoT infrastructure management tools on the market: a recent survey is in [15], with more than 60 IoT middle-wares, 14 of which follow a Service Oriented approach. Our intent is not to introduce a new tool of the same kind, but to contribute to aggregate experience around shared concepts and eventually standards.

VIII. Conclusion

Complex IoT infrastructures are emerging, for the management of which appropriate tools are required to abstract from details and concentrate on system organization. We need to identify the requirements of such infrastructures, in order to define a formalism to describe them, and to implement the applications that deploy a software description as a functional infrastructure. It is also important that tools are open, expandable, and interoperable, to allow the widest outreach, that facilitates a fast progress.

To this end we define an approach that, starting from the requirements defined in [18], introduces a schema derived from OGF OCCI that decomposes the infrastructure into entities and defines their semantics and interconnection. OCCI is simple, open, and expandable, and our schema inherits such features. It enforces a REST approach for the interaction with *engine* components that are in charge of deploying the infrastructure. So we introduce the last tile in the design: HTTP as the communication protocol, which aims at interoperability.

All together, these concepts and tools make a scalable solution for managing complex IoT infrastructures. To demonstrate that they fit together and that the solution is appropriate for the task, we have implemented a proof-of-concept prototype of an *engine* component that matches our proposal: the sources and the hardware layout are available in a public repository to allow verification and improvement. The *engine* is scalable by design, since it is made of components that can be easily replicated and connected.

The next step on the main stream is to improve the definition of the interface, to make it practically adoptable, and define the software structure of the engine to give the guidelines for future products.

But we have also noticed how the basic building blocks are fragile: the *things* that populate the back-end of our system are equipped with firmware that is a bottleneck for security (no secure WebSockets), reliability (broken connections), and performance (drifting clocks). It is therefore advisable to define *down-graded* standards for IoT , and give the developers the guidelines to design smarter *things*.

References

[1] Min Chen, Yongfeng Qian, Shiwen Mao, Wan Tang, and Ximin Yang. Software-defined mobile networks security. *Mobile Networks and Applications*, 21(5):729–743, 2016.

[2] Augusto Ciuffoletti. Application level interface for a Cloud Monitoring service. *Computer Standards & Interfaces*, 46, May 2016.

[3] Augusto Ciuffoletti. Occi4iot - occi-compliant orchestrator for iot systems. https://github.com/mastrogeppetto/OCCI4IOT, 2016.

[4] Augusto Ciuffoletti. A websocket based temperature sensor. https://github.com/mastrogeppetto/arduino_NTCwebsocket, 2016.

[5] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[6] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[7] Ian Hickson. The websocket api. Technical report, W3C, 2011.

[8] L. Jacquin, A. L. Shaw, and C. Dalton. Towards trusted software-defined networks using a hardware-based integrity measurement architecture. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, April 2015.

[9] Yaser Jararweh, Mahmoud Al-Ayyoub, Ala' Darabseh, Elhadj Benkhelifa, Mladen Vouk, and Andy Rindos. Sdiot: a software defined based internet of things framework. *Journal of Ambient Intelligence and Humanized Computing*, 6(4):453–461, Aug 2015.

[10] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[11] Y. Li and M. Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[12] L. Mainetti, V. Mighali, and L. Patrono. A software architecture enabling the web of things. *IEEE Internet of Things Journal*, 2(6):445–454, Dec 2015.

[13] OGF. *Open Cloud Computing Interface - Core*. Open Grid Forum, June 2011. Available from www.ogf.org. A revised version dated 2013 is available in the project repository.

[14] Daryl C. et al. Plummer. Top strategic predictions for 2016 and beyond: The future is a digital thing. Technical report, Gartner, 2015.

[15] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95, Feb 2016.

[16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[17] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers. Twenty security considerations for cloud-supported internet of things. *IEEE Internet of Things Journal*, 3(3):269–284, June 2016.

[18] M. Stecca, C. Moiso, M. Fornasa, P. Baglietto, and M. Maresca. A platform for smart object virtualization and composition. *IEEE Internet of Things Journal*, 2(6):604–613, Dec 2015.

[19] Jeffrey Voas. Networks of "Things". Special Publication 800-183, National Institute of Standards and Technology, July 2016.

[20] Y. Xu and A. Helal. Scalable cloud-sensor architecture for the Internet of Things. *IEEE Internet of Things Journal*, 3(3):285–298, June 2016.