Stack4Things: integrating IoT with OpenStack in a Smart City context

Giovanni Merlino*‡, Dario Bruneo*, Salvatore Distefano†§ Francesco Longo*, Antonio Puliafito*

* *Dipartimento DICIEAMA, Università di Messina,*
Contrada di Dio, 98166 Messina, Italy
Email: {dbruneo,flongo,gmerlino,apuliafito}@unime.it

† *Dipartimento DEIB, Politecnico di Milano*
Piazza L. Da Vinci 32, 20133 Milano, Italy
Email: salvatore.distefano@polimi.it

‡ *Dipartimento DIEEI, Università di Catania*
Viale Andrea Doria 6, 98166 Catania, Italy
Email: giovanni.merlino@dieei.unict.it

§ *Machine Cognition Lab, Higher Institute of Information Technologies and Information Systems*
Kazan Federal University, Kazan, Russia

***Abstract*—As the adoption of embedded systems, mobiles and other smart devices keeps rising, and the scope of their involvement broadens, for instance in the enablement of Smart City-like scenarios, a pressing need emerges to tame such complexity and reuse as much tooling as possible without resorting to vertical ad-hoc solutions, while at the same time taking into account valid options with regards to infrastructure management, and other more advanced functionalities. In this sense, a widely used and competitive framework for Infrastructure as a Service such as OpenStack, with its breadth in terms of feature coverage and expanded scope, looks like fitting the bill. This work thus describes rationale, efforts, and results so far achieved, for an integration of IoT paradigms and resource ecosystems with such a kind of Cloud-oriented environment, by focusing on a Smart City scenario, and featuring data collection and visualization as example use cases of such integration.**

***Keywords*-IoT; Smart City; Cloud; IaaS; OpenStack; Ceilometer; MOM; AMQP; CoAP; REST; CEP**

I. Introduction

In recent years, different approaches, also based on new technologies, have been adopted in order to morph cities into smart ones. The Smart City scenario is a fertile application domain for different sciences and technologies, in particular for those related to the information and communication areas. Several ongoing projects illustrate Smart City opportunities and challenges [1] in diverse application fields, e.g., networking, decision support-systems, power grids, energy-aware platforms, service-oriented architectures, highlighting the need to equip the cities of the future with a variety of urban sensors. Plenty of applications can be envisioned from traffic monitoring to energy management, from e-health to e-government, from crowd to emergency management. This all-encompassing and much ambitious scenario calls for adequate ICT technologies. In particular, solutions for managing the underlying physical sensing and actuation resources infrastructure are required.

A number of solutions of this kind, mainly at a lower (communication) layer, may be found within the Internet of Things (IoT) [2] domain, mainly aiming at interconnecting network-enabled devices and, generally, any *thing* featuring a network interface, to the Internet. However, in order to manage devices, sensors, and things building up a dynamic Smart City infrastructure, management, organization, and coordination mechanisms are also required. In particular, Cloud computing facilities, implementing a service-oriented approach in the provisioning and management of resources, may be exploited. The Cloud approach could be a good solution to address Smart City-related issues, fitting with the requirements of relevant service users and providers: on-demand, elastic and QoS-guaranteed, to name a few, all needed properties for a Smart city service platform.

Several works deal with infrastructure issues and solutions related to Smart Cities and their relationship with IoT and Cloud. Specifically, the authors of [3] propose a platform for managing urban services that include convenience, health, safety, and comfort. Cloud computing infrastructure recently [4], [5] found useful application in the context of Smart Cities. In [6], authors took a different trajectory to tackle transducers in a Cloud environment: sensing and actuation devices should be handled along the same lines as computing and storage abstractions in traditional Clouds, i.e. virtualized and multiplexed over (scarce) hardware resources. This step paves the way to an IoT/Cloud-powered Smart City framework able to support related services in the (sensing) infrastructure provisioning. Even if a lot of

978-1-4799-6447-5/14/$31.00 ©2014 IEEE

applications in the Smart City scenario have been proposed so far, there is a lack of common initiatives and strategies to address issues at an infrastructural level.

In order to fill this gap between Smart City applications and the underlying infrastructure, in this paper we propose to extend a well known framework for the management of Cloud computing resources, OpenStack, to manage sensing and actuation ones, implementing in our *Stack4Things* solution an infrastructure-oriented [6] approach, while coping with communication requirements and scalability concerns by leveraging Cloud-focused design choices and architectural patterns.

The remainder of the paper is thus organized as follows. Section II describes and investigates the Smart City scenario from an infrastructure-focused perspective. Then, Section III details the approach leading to the adaptation and the extension of OpenStack to the scenario previously described. Technical details and design decisions of the Stack4Things framework components implemented so far are discussed in Section IV. At last, a discussion and future steps about the development of the framework are provided in Section V, while conclusions are in Section VI.

II. Reference scenario

The Smart City scenario is one of the most prominent scenarios for IoT involving computer-assisted treatment of sensed data and automation of urban areas and public facilities. Smart Cities are usually characterized by field-deployed, dispersed, sensor-hosting smart platforms, usually available in several orders of magnitude bigger quantities, possibly heterogeneous along several axes (instruction set architecture, operating system, userspace, runtime, etc.). What makes a city *smart* are, apart from green and environment-friendly technologies, at least sensing and communication facilities, also embracing mobility, in order to enhance public transportation systems and monitor physical infrastructure (buildings, open spaces, etc.). When actuating subsystems enter the picture, the city as a whole may be compared to a nervous system, where reactive mechanisms are in place, either mediated centrally or even working entirely autonomously. Indeed, as hinted at in Figure 1, a middleware devoted to management of both sensor- and actuator-hosting resources may help in the establishment of higher-level services, including policies for "closing the loop", such as, e.g., configuring triggers for a range of (dispersed) actuators based on sensing activities from (geographically non-overlapping) sensing resources.

On the other end of the scale, we have this recurring paradigm of typically outsourced, centralized services over which to delegate most computation and storage needs, going under the umbrella term of Cloud Computing [7]. We believe that, yet for all the distance between the IoT and Cloud perspectives, a wide range of synergies and opportunities lies at the intersection of Cloud and IoT,

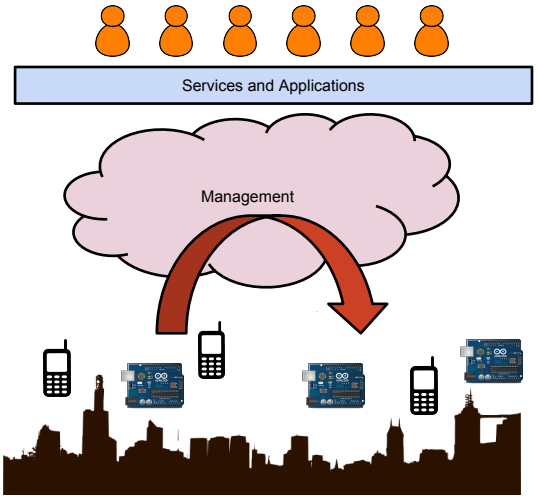

Figure 1. Smart City as closed-loop system.

especially in the contest of Smart Cities. Indeed, what is needed for a Smart City platform is at least a (Virtual) Infrastructure Manager (VIM) over which to overlay higher level services, starting from the most basic ones such as data collection, in order to extract value or possibly monetize the investments. In fact, in terms of requirements, it follows that any IT infrastructure powering Smart Cities features IoT-related devices as well as typical datacenters. In line with recent trends and latest advances, this means on one hand resorting to private (or hybrid) Cloud-enabled infrastructure, if not totally outsourcing compute and storage requirements to any of the established public IaaS providers. On the other hand, embedded systems are getting ever more powerful and flexible in terms of reprogrammable behavior and ease of use. Indeed, most devices of such a kind are gaining a "smart" labeling to indicate this evolution.

Aiding the authors in framing the requirements for such a scenario there is the early involvement in the $\#SmartME$ project, dealing with the enablement of the municipality of Messina as Smart City. Under the umbrella term of $\#SmartME$ lie both an effort to fund and procure hardware (thousands of boards and relative transducers) by a range of channels, including crowdfunding, and a plan to deploy and setup such huge IoT *dust* over a district-wide urban area.

As a very high-level description of the requirements, the ability to identify, spot (on a map) and check devices for availability and overall health status, as well as to retrieve measurements on a variety of real-world phenomena, to be stored and visualized either for real-time consumption or long-term analysis for trends and other statistics to be inferred, are all to be provided for a working Smart City implementation. Scalability is also an essential requirement, at least after early experimentation and pilot testing, when thus getting into production stages.

Taking into account other experiences, such as the earliest experiments around Smart City planning, e.g. projects like SmartSantander [8], typically most efforts revolve around managing heterogeneous devices, usually by resorting to legacy protocols and vertical solutions out of necessity, and integrating the whole ecosystem by means of an ad-hoc solution. In our vision the Cloud may play a role both as a paradigm, and as one or more ready-made solutions for a VIM, to be extended for IoT infrastructure, as detailed in the next section.

III. An OpenStack-based approach

In the pursuit for integration of Cloud technologies with IoT infrastructure, we are trying to follow a bottom-up approach, consisting of a mixture of relevant, working frameworks and protocols, on one hand, and interesting use cases to be explored according to such integration effort.

Indeed, beyond concerns about the scale of the effort, other requirements such as elasticity of the sensing-based services to be provided, as well as registration and provisioning mechanisms of the underlying heterogeneous sensor-hosting platforms deserve an Infrastructure Manager (IM) anyway. For instance, scalability alone calls for a communication framework based on some kind of Message-Oriented Middleware (MOM), something fully distributed IMs usually expose as a built-in feature, as MOMs typically are a core building block of such solutions.

A Cloud-oriented one fits the scenario, meeting the aforementioned requirements by default in order to cater to the originally intended user base, while at the same time also addressing other more subtle functionalities, such as a tenant-based authorization framework, where several actors (owners, administrator, users) and their interactions with infrastructure may be fully decoupled from the workflows involved (e.g., transfer, rental, delegation). Bonus points include recycling existing (compute/storage-oriented) deployments, getting most visualization and monitoring technologies for free, as those are typically already available in such systems, possibly even enabling federation of different administrative Cloud-enabled domains.

In this sense, our choice leans towards OpenStack, as a centerpiece of infrastructure Cloud solutions for most commercial, in-house and hybrid deployments, as well as a fully OpenSource ecosystem of tools and frameworks upon which many EU projects, such as CloudWave (FP-7), are founding their Cloud strategies. Moreover, with an independent consortium featuring a diverse partnership, including world-class industrial players and smaller enterprises, and a global community of thousands developers, OpenStack has the clout to withstand most shifts in paradigms and approaches, and holds the promise to keep its relevance in time within the increasingly crowded Infrastructure as a Service space.

Delving into the details about the implementation, when talking about the IoT side of the solution, typical embedded systems, and the resource-level limitations they embody, call for ad-hoc solutions. In this sense, a natural choice among standards for communication, and in particular for application-layer protocols, is CoAP [9] (Constrained Application Protocol). However, straddling distributed things and centralized subsystems is the "bus" layer. CoAP or other RESTful end-to-end protocols just scratch the surface in terms of the possibilities being enabled.

Indeed, a bus enables both a forwarding model and a queuing system, otherwise absent in pure end-to-end communication. In terms of the former, we gain in terms of flexibility, whereas for the latter, it means avoiding to leave scalability issues to the parties involved by leveraging infrastructure in the middle instead. Moreover the delivery becomes transparent, by embodying publish-subscribe mechanisms. It is also under this perspective that OpenStack and in particular its communication backbone, the *bus*, becomes part of the solution.

Thus, shifting our analysis to the Cloud-side subsystem, the OpenStack component for metering (and billing) is Ceilometer, while Horizon is the dashboard, hence we turned first our attention on these frameworks in order to tackle data acquisition and visualization duties, for samples coming from the "things".

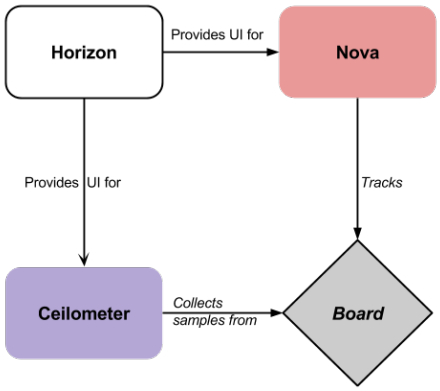

Figure 2. Stack4Things core components: conceptual architecture.

Highlighting this focus, in Figure 2 a conceptual architecture [10] of OpenStack depicting components, as boxes, and the services they provide to other components, with arrows, respectively, gets simplified in order to take into account just core components for our approach. Moreover, in place of a VM, in this case the diamond-shaped box symbolizes a (sensor-hosting) board, and corresponding interactions are described as text in italic along the arrows.

Indeed, while metering in Ceilometer is a feature needed for billing purposes, itself a core duty for any Cloud solution, in our case we reverse the perspective, as we are interested in measurements for their own sake, in order to monitor

vital urban parameters, even if we do not exclude further exploitation of the framework for billing in the future, as the Cloud paradigm may be applied to sensing resources to its fullest, as already investigated in the Sensing and Actuation as a Service [6] approach.

Ceilometer, like most other components of the middleware, cannot be fully analyzed on its own, as it needs to interface to, and support, the compute resource management subsystem, Nova, lying at the core of OpenStack. In particular, while both Ceilometer and Nova exploit a common bus, the latter alone dictates a hierarchy on participating devices, including their role and policies for interaction. Indeed, Nova requires a machine operating as Cloud *controller*, i.e., centrally managing one or more *Compute nodes*, which are typically expected to provide component-specific services (e.g., computing) by employing a resource-sharing / workload-multiplexing facility, such as an, e.g., *hypervisor*.

IV. Implementation

In this section we are going to describe our implementation efforts, geared toward both data collection and visualization, as foremost examples of the kind of requirements to be met in order to provide a workable Smart City-enabling solution.

A. Data collection and inference

As depicted in Figure 3, the whole data collection architecture comprises both additional modules for Ceilometer, and external components needed for higher-level functions, e.g., event processing. Framing the discussion in terms of the devices involved and of compute-based IaaS, the board may be considered an instance of a machine, e.g., a VM or even just a Cloud-provisioned physical machine.

Along the same lines, a Compute node is just any machine which, on one hand, may host standard compute VMs, on the other, may be IoT-enabled in order to supervise and track the lifecycle of one or more boards. The Controller in turn is expected to host a Collector (for Ceilometer) as a centralized component for data collection, storage or further processing if needed.

1) Board-side: s4tProbe: A probe (s4tProbe), to be hosted in an active instance of a runtime, has been implemented in Python, based on the *stevedore* library for enabling and loading a monitoring plugin. There is such a plugin for each runtime environment (depicted as *E*), such as Node.js or Python, available on the sensor board, in our case an MPU-equipped Linux-hosting Arduino embedded system, the YUN [11]. The same tools and approach are employed to load other OpenStack modules, such as the *pollster*, the *dispatcher*, etc., in particular with regard to Ceilometer, in turn made up of an *agent* and a *controller*. More in detail, an abstract class (*PluginBase*) has been designed for the implementation of new plugins, and is to be enabled by the aforementioned library for plugins.

In order to enable communication toward the Ceilometer Agent for the Compute node, the queue-based Apache Qpid messaging system has been employed, providing an implementation of AMQP (Advanced Message Queuing Protocol), an open standard application layer protocol for message-oriented middleware, a bus featuring message orientation, queueing, routing (including point-to-point and publish-subscribe), reliability and security.

In particular the circle-edged arrows represent the interaction model, where there are bus-enabled queues, one or more publishers as well as one or more subscribers, respectively depicted as pointing toward the queue and out of the queue.

For each instance a message queue gets initialized automatically (if not already available), featuring a namespace for addressing under the AMQP environment according to the following format:

```
stack4things.<COMPUTE-HOSTNAME>.<VM-IDENTIFIER>
```

As alternative to this pull-based bus-compliant design, a RESTful approach has been implemented, push-based and layered over CoAP, whose interactions are represented by arrows pointing to RESTful interfaces. This kind of interaction model has been implemented by exposing an *observations*-enabled CoAP server on the board, and letting a CoAP client (the REST Module) inside the Agent register itself for notifications, in order to get updates pushed as soon as new samples get collected by sensors of interest.

2) ComputeNode-side: s4tAgent: On OpenStack Compute nodes a Qpid server had to be installed as prerequisite with the corresponding Python modules, usually already available by default on the Controller, as there Qpid is needed for the communication between Ceilometer agent and collector.

In order not to twist the OpenStack architecture we opted to implement a pollster, e.g., a polling-based sample dequeuing object, which is one of the officially sanctioned methods typically suggested to extend Ceilometer for the monitoring of extra metrics, i.e., not measurable through Libvirt such as the ones exposed by default. This component[1] is to be executed after a fixed amount of time by OpenStack, according to the configuration[2] of "pipelines". A connection would be established to the AMQP queue of each Stack4Things instance running on that Compute node inside this new pollster by using Qpid methods. The pollster would then go on to extract from queues the messages bearing the samples sent by Probe plugins. The information of interest thus obtained would be used to build

[1] `/usr/lib/python2.6/site-packages/ceilometer/compute/pollsters/s4tpoll.py`

[2] `/etc/ceilometer/pipeline.yaml`

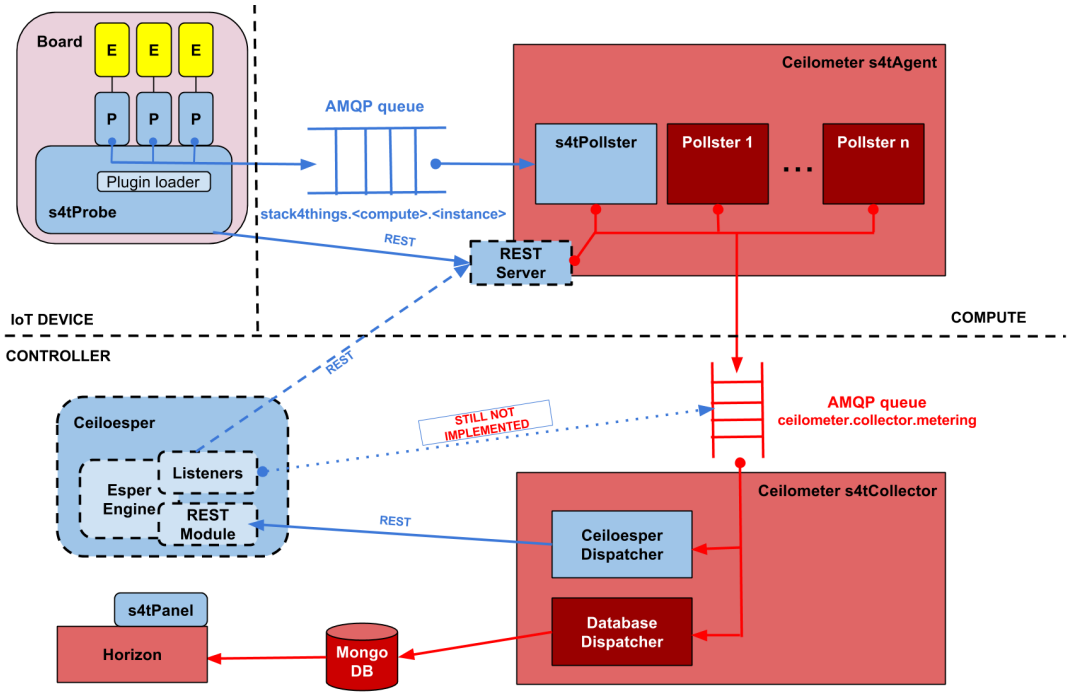

Figure 3. Data collection and inference subsystem: architecture.

a Ceilometer-compliant "sample", by means of the function `util.make_sample_from_instance`.

Each sample would then be processed by the Ceilometer Agent to be sent in turn to the Ceilometer Collector hosted on the Controller, by means of the ad-hoc queue `ceilometer.collector.metering`. In order to enable the new pollster at boot time for the Ceilometer Agent, the following line has to be added into section `[ceilometer.poll.compute]`:

```
s4tpoll=
ceilometer.compute.pollsters.s4tpoll:S4tPollster
```

inside the corresponding file[3].

3) Controller-side: s4tCollector: In order to send Stack4Things measurements, as received from the Collector, to an inference engine for the infrastructure, called Ceiloesper, a new dispatcher (CeiloesperDispatcher) for the Collector had to be implemented. The latter features by default two kinds of dispatchers for writing Ceilometer measurements either on file or in a DB (MongoDB by default), respectively. Even in this case, the addition of our new dispatcher did not lead to any modification to the core setup of Ceilometer, or OpenStack overall. This new dispatcher would analyze only measurements coming from probes and encoded as proper Samples by the Ceilometer Agent. More in detail, upon creation of the sample at the Ceilometer Agent the field `meter_dest:ceiloesper` would be added into section `resource_metadata` of the corresponding JSON. Such field would aid in distinguishing Ceilometer samples from those generated through Stack4Things additions and meant to be consumed by Ceiloesper. The forwarding of measurements to Ceiloesper would work over REST by sending a JSON message such as the following:

```
'{''probe_inst'':''$+$meter_ist_name$+$''',
''volume'':''$+$meter_volume$+$''',
''timestamp'':''$+$meter_timestamp$+$''',
''name'':''$+$meter_name$+$''',
''unit'':''num'',
''ev_type'':''measure'',
''source_type'':''board'',
''type'':''gauge'',
''additional_metadata'': \
'$+$additional_metadata$+$'}'
```

where, apart from self-describing parameters, there is a UUID as *probe instance* identifier, the sample value under *volume*, any descriptive name under *name* as well as any other (nested) JSON under *additional metadata*.

Ceiloesper is a service built upon Esper, a Java-based CEP (Complex Event Processor) for the detection and management of complex events. This is built by implementing an Esper *listener* for each event of interest. Event triggering

[3] `/usr/lib/python2.6/site-packages/ceilometer-2013.2.3-py2.6.egg-info/entry_points.txt`

may be based on thresholds, or pattern matching for events of higher complexity; such rules are described in the EPL language, similar in form to SQL queries. At the moment two listeners relative to the same kind of measurement (in our case, temperature and brightness, i.e. illuminance) have been implemented already:

- detecting levels in excess of a certain threshold
- detecting levels getting again below threshold, by pattern matching

Ceiloesper implements a built-in REST server listening for measurements sent by dispatcher we wrote for the Collector (CeiloesperDispatcher). Upon arrival of each measurement an "event object" (called *bean*), relevant to the metric to be monitored, would be instantiated, whose fields are to be populated with values extracted from the incoming JSON message. The relevant listener to such object class would detect the construction of the new object and check whether the EPL rule has been met or else. Indeed, if the rule is respected, an event trigger would fire and a signal (in JSON) would be sent to the Ceilometer Collector, which would manage it as a Ceilometer sample and write it into MongoDB, as such from this point onwards available to be rendered in the dashboard, as any other kind of measurement.

B. Visualization: dashboard

Apart from Web UI-based administration tasks, the OpenStack Dashboard, codenamed Horizon, is employed also for data visualization duties.

In order to add a realtime graph and a Google Maps-powered sensors geolocation system to Horizon, an IoT-enabled panel, the s4tPanel, has been implemented, included inside the "admin" project label group and labeled "IoT".

The latter requires dropping in a predefined directory[4] a folder (`iot`) which contains all the essential files such as:

- `panel.py` to set the name of the panel and relative access permissions;
- `tabs.py` to create one or more tabs inside the panel where all graphs are located (in our case only a "Stats" tab has been created);
- `templates/iot/stats.html` to add Javascript code related to real-time graphs and Google Maps diagrams but also to specify the position of the relative div sections inside the html page.

The `dashboard.py` file, available in the admin folder described before, needs to be modified, in order to enable the Dashboard to show panels on the left sidebar.

Other files involved in this scenario are the ones used to retrieve information from the database by sending a cURL request and then to parse the response to return data to the Javascript code in order to plot graphs:

[4] `/usr/share/openstack-dashboard/openstack_dashboard/dashboards/admin`

- `retrieve.sh` to get the admin token which will be used to retrieve metrics of the relative sensors (temperature, brightness, number of sensors and their Google Maps coordinates);
- `chart.php` and `gmap.php` to respectively parse data related to real-time and gmap metrics.

Talking about the *Real-time* section, in our test we collected measurements related to brightness and temperature from two peripheral sensors connected to the "main sensor node". Periodically, with a user-selectable frequency, a request to the MongoDB is sent to retrieve information on the last sample available for each of the aforementioned measurements.

In Figure 4 there is a screenshot that shows the panel layout and how metrics are plotted in real-time. As long as there are new incoming values to be plotted, lines are updated accordingly. In this case, the upper line represents the brightness, the lower one the temperature and the one in the middle the number of peripheral sensors, the latter a constant in one of our tests as depicted in figure.

Turning our attention to the "Map of Sensors" section we used the Google Maps API to geolocate sensors, whose coordinates are sent inside the `resource_metadata` JSON field inside Ceilometer metrics, and also to visualize the last sample value for each sensor (master and peripherals).

In the screenshot in Figure 5 there is a short description explaining how many sensors are connected to the master node, e.g., a single board hosting however many local (or proximal) sensors.

With regard to the metrics under consideration the screenshot in Figure 6 shows information related to a sampled value of one such metric, and to the relative timestamp at which this measurement was sent to the database from the sensor.

V. Lessons learned: limitations and future work

A first evident limitation we can immediately highlight is related to coupling devices with measurement activities, by latching those to already available instances of running compute VMs. In order to support IoT dynamically, devices are to be treated as first-class objects of the infrastructure, thus we hope to soon have a working framework for tagging devices as PMs (with an identification system, e.g., UUID, to be provided by agents).

Another fundamental requirement we have only started addressing is that of scalability. In this sense, under the currently prevalent approach in our implementation, i.e., polling, load balancing may be achieved by multiplying Ceilometer Agents. Each Ceilometer Agent can poll a "cloud" of smart things probably deployed on the same physical area, as can be seen in Figure 7.

A second stab at scalability may thus involve reassessing the overall approach toward a push agent, exposing a CoAP/REST server, as also represented in Figure 7. In fact,

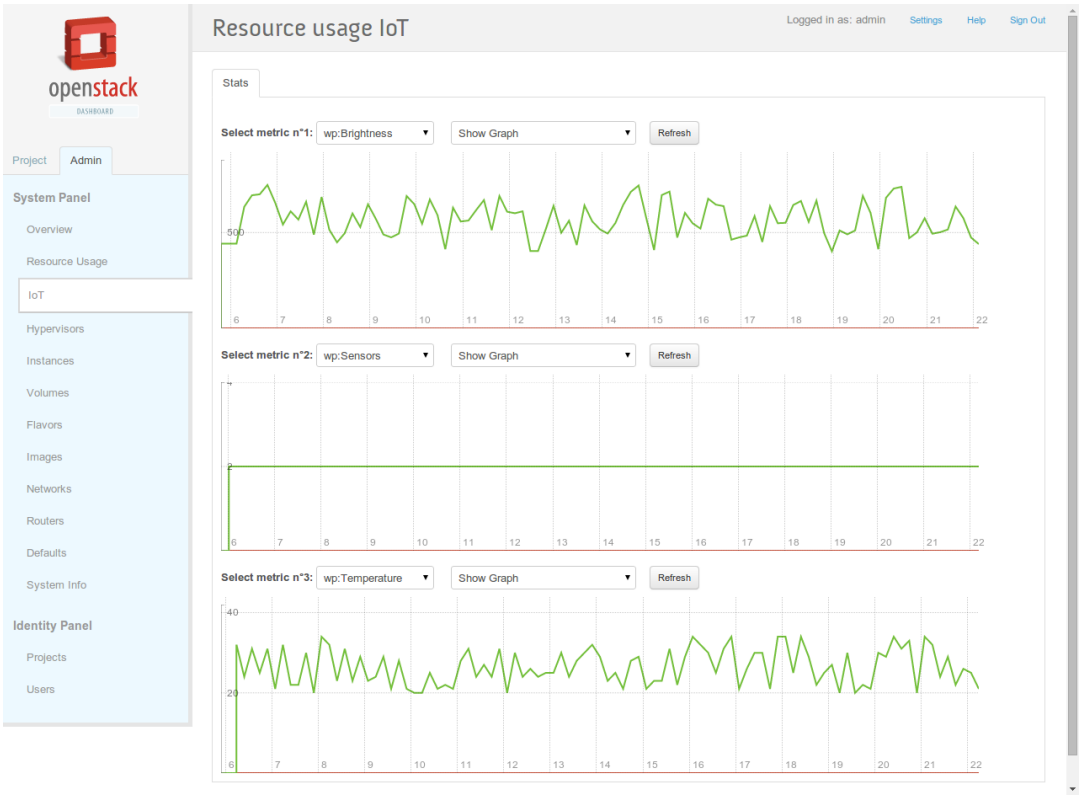

Figure 4. Horizon - IoT panel: real-time graphs.

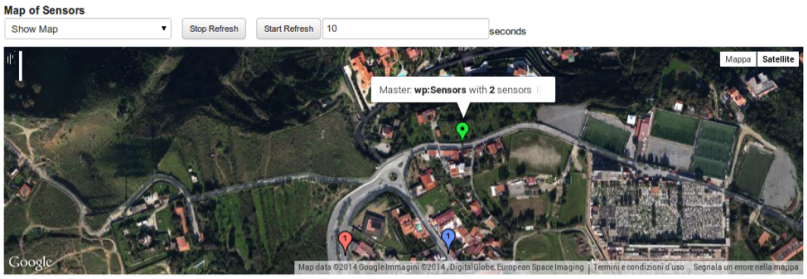

Figure 5. Map screenshot: node resources.

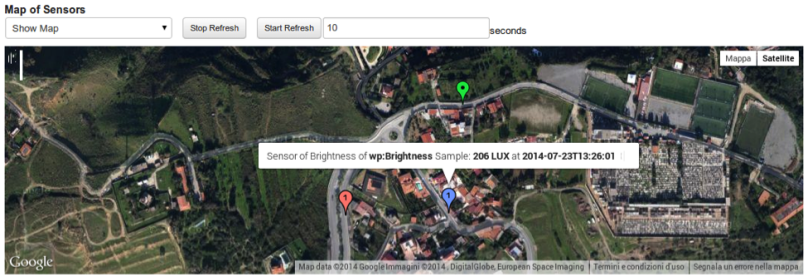

Figure 6. Map screenshot: brightness sample visualization.

as a protocol geared towards constrained devices, CoAP works over a transport protocol with the least overhead and most simplified transmission model, i.e., plain UDP, and the message format is binary to cut on bandwidth requirements, but for all its simplicity, the semantics are very similar to HTTP, and in some specific cases even

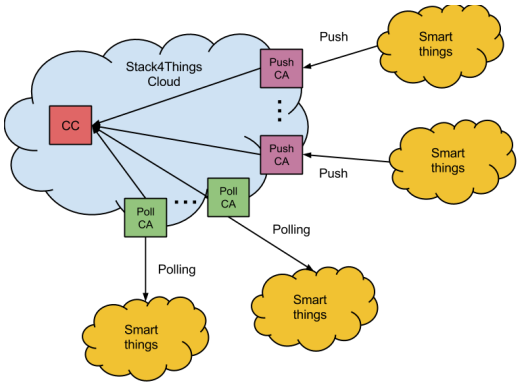

Figure 7. Scalability achieved by multiplying CAs each of which polls a "Cloud" of smart things and by exploiting push CAs.

extended to accomodate the kind of envisaged use cases, e.g., pub/sub-based interaction patterns. Moreover, featuring a broad compatibility for mapping with HTTP, CoAP fits by design with HTTP-based paradigms, such as RESTful services. Support for REST in turn means being able to exploit a de-facto standard for the engineering of distributed systems, making issues of interoperability and integration more straightforward to address.

More in general, we need a more all-round architecture, also addressing other issues (provisioning, security, real-time visualization), possibly by looking deeper into current and forthcoming directions of the OpenStack project, as exemplified in project *blueprints*.

VI. Conclusions

The growing attention devoted to Smart Cities calls for effective ICT solutions able to cope with huge quantities of data and devices that need to be managed in order to provide smart services to citizens. In this paper, we presented Stack4Things: a first attempt in using OpenStack as the underlying technology for the management of IoT devices according to a Cloud-oriented approach. We described the architecture of the system by focusing on data management and visualization aspects, while also presenting some implementation details. Future work will include the implementation of the whole architecture and its exploitation in a real-world scenario.

Acknowledgment

This work is partially funded by the European Union 7th Framework Programme under grant Agreement 610802 for the "CloudWave" project, by a PhD programme under grant PON R&C 2007/2013 "Smart Cities", by the "SIMONE" project under grant POR FESR Sicilia 2007/2013 n. 179, and by the Integrated Cloud-Sensor System for Advanced Multirisk Management (SIGMA) Project, Italian National Operative Program (PON) 2007-2013.

References

[1] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter cities and their innovation challenges," *Computer*, vol. 44, no. 6, pp. 32–39, Jun. 2011.

[2] 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.

[3] J. Lee, S. Baik, and C. Choonhwa Lee, "Building an integrated service management platform for ubiquitous cities," *Computer*, vol. 44, no. 6, pp. 56–63, Jun. 2011.

[4] Z. Li, C. Chen, and K. Wang, "Cloud computing for agent-based urban transportation systems," *IEEE Intelligent Systems*, vol. 26, no. 1, pp. 73–79, Jan. 2011.

[5] N. Mitton, S. Papavassiliou, A. Puliafito, and K. S. Trivedi, "Combining cloud and sensors in a smart city environment," *EURASIP J. Wireless Comm. and Networking*, vol. 2012, p. 247, 2012.

[6] S. Distefano, G. Merlino, and A. Puliafito, "Sensing and actuation as a service: A new development for clouds," in *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, Aug 2012, pp. 272–275.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[8] L. Sanchez, J. Galache, V. Gutierrez, J. Hernandez, J. Bernat, A. Gluhak, and T. Garcia, "Smartsantander: The meeting point between future internet research and experimentation and the smart cities," in *Future Network Mobile Summit (FutureNetw), 2011*, June 2011, pp. 1–8.

[9] C. Bormann, A. Castellani, and Z. Shelby, "Coap: An application protocol for billions of tiny internet nodes," *Internet Computing, IEEE*, vol. 16, no. 2, pp. 62–67, March 2012.

[10] "OpenStack conceptual architecture [URL]," http://docs.openstack.org/admin-guide-cloud/content/conceptual-architecture.html.

[11] "Arduino YUN [URL]," http://www.linino.org/modules/yun.