A Utility Paradigm for IoT: the Sensing Cloud

Salvatore Distefano\textsuperscript{a}, Giovanni Merlino\textsuperscript{b}, Antonio Puliafito\textsuperscript{b}

\textsuperscript{a}Politecnico di Milano, Piazza L. Da Vinci 32, 20133 Milano, Italy
\textsuperscript{b}Università degli Studi di Messina, 98166 Messina, Italy

Abstract

IoT is such a key trend in ICT that it is quickly becoming one of the most influential research and development topics. This popularity is spawning also lots of laudable initiatives, one of the most prominent being carried on by the IoT-A consortium, including influential blueprints such as its Reference Architecture (RA). Their main goal is to interconnect network-enabled devices and “things” through the Internet. This bottom-up view of IoT is lacking mechanisms for aggregating, managing and administrating groups of things. Such a perspective could be reverted to provide control and management facilities through specific framework and software, in line with new trends such as software defined networking.

In this paper we propose a top-down utility paradigm for IoT starting from the IoT-A reference architecture and the Sensing and Actuation as a Service (SAaaS) approach. It aims at implementing a sensing Cloud by enrolling and aggregating sensing resources from sensor networks and personal, mobile devices. We follow a device-driven approach, as adopted in IaaS Clouds: once collected, the physical (sensing) resources are abstracted and virtualised and then provided as a service to end users. A key point of the SAaaS is the abstraction of resources, i.e. providing a uniform way to access to and interact with the underlying physical nodes in compliance with IoT goals. The main contribution of the paper is the design and development of the sensing resource abstractions for SAaaS to demonstrate the feasibility of such an approach, providing details on theoretical and design aspects as well as technical ones. In particular, a preliminary implementation for mobiles is described, delving in platform-dependent details where needed. The facilities thus developed under the Android platform have been tested through a typical IoT application, in order to gauge the validity of the approach.

Email addresses: salvatore.distefano@polimi.it (Salvatore Distefano), gmerlino@unime.it (Giovanni Merlino), apuliafito@unime.it (Antonio Puliafito)
1. Introduction and Motivations

The growing trends involving personal and mobile devices have made quite an impact on the current ICT scenario bringing about huge networks of more and more powerful smart devices. This state of affairs has spawned new ideas and paved the way for rethinking and reinterpreting foundational technologies such as Internet, driving efforts towards the Internet of Things (IoT), or service engineering at the basis of the Cloud computing and the Web 2.0.

Plenty of devices (tens of billions according to recent reports [1]), usually embedding a diverse range of sensing resources as well as extensive connectivity capabilities, populate and characterize the scenario, calling for adequate techniques and tools for their management in order to unlock their potential. IoT mainly aims at rationalising this ecosystem of (more or less) smart objects, on one hand by connecting heterogeneous devices through Internet thus enabling remote interactions. On the other hand by raising the abstraction level all the way up to, and including, very high-level representations, such as those enabled by virtue of associating the object with a semantics to identify things (cars, documents, people, etc.).

Several efforts at different levels are focused on achieving IoT aims and goals. Governments, standards bodies, research communities and companies are active on IoT from their different, complementary perspectives. These produced a wide variety of concepts, definitions and solutions respectively [2, 3, 4, 5, 6, 7]. In particular, an activity aimed at promoting and rationalizing the efforts on an IoT infrastructure is the IoT-A project [8]. It is focused on specifying an architecture designed to support and include most IoT-related functionalities and services. The IoT reference architecture identifies views, perspectives, functional modules and basic blocks required to manage IoT infrastructure. Aspects related to device connectivity and communication, as well as those pertaining to data resource management and services are addressed, covering a range of issues raised in IoT contexts. Indeed devices have to be mainly able to communicate and exchange data, while also exposing services in order to provide access to such assets.

1.1. A new perspective

In our opinion, while comprehensive, this is still a partial perspective on IoT, coming from a device-level only viewpoint, mainly aimed at interconnecting and
interoperating heterogeneous platforms. But a new perspective, from a higher level, is possible, as also demonstrated by current networking trends such as *software defined networking* (SDN) [9], aimed at managing network services by abstracting and decoupling lower level functionalities. Indeed the goal of SDN lies in splitting control and data flows, i.e. moving the former to a high, logically centralized, layer.

Following this approach, we think about feasible mechanisms and solutions to face a greater challenge across the board, namely one where such ecosystem of things geographically distributed may be discovered, selected according to the functionalities they provide, interacted with, and may even cooperate for pursuing a specific goal. A new, extended and more comprehensive interpretation of the IoT paradigm is thus achievable by thinking about a greater involvement of the devices through their sensing and actuation capabilities and/or resources. From this perspective, IoT could also allow end-users to grab handles on the sensing resources populating the involved devices, not just the mere data output they produce. To this end, a complete set of rich mechanisms for managing sensing resources would do the job.

So far, to the best of our knowledge, existing approaches mainly deal with aspects and issues related to connectivity and communication. With regard to the management and exploitation of IoT resources through applications or services, a *data-centric* approach is currently prevalent [10, 11, 12], i.e. just considering the data produced and/or consumed by the device sensing and actuation resources in place of the resources and platforms themselves. In fact, both the management and the high level facilities in current IoT infrastructure and services just take into account (sensed) data, as captured by the underlying devices. A way to leverage further the IoT paradigm is by also allowing users to get handles on sensing resources instead of just the data they produce, thus implementing the provisioning of virtual instances over actual resources through a *device-centric* approach.

Even if the data-centric approach is interesting and effective in several contexts, the device-centric one features some benefits in comparison:

- it allows to decentralize control, transferring intelligence to the resources through customization of virtual sensors and actuators, where the software/service, as required by the user, are to be deployed;

- data are pre-filtered and processed before being transferred; furthermore, just one transfer (virtual sensor/actuator ⇔ user) is required, against the two required in the data-centric approach (sensor/actuator ⇔ DB/provider ⇔ user),
thus reducing the amount of data to be transferred as well as round-trip times;

- it features higher granularity and choice, as the approach provides a degree of freedom in aggregating and composing as well as in repurposing sensors and actuators, which by design cannot be provided within data-centric solutions;

- it allows to adopt the right level of security with regard to the actual constraints for involved devices, e.g. shifting the complexity to the network in case the sensor-hosting platform is relatively simple, or leaving the burden onto the device in case there is enough (computational) capacity available;

- in terms of information dissemination, unlike centralised protocols which focus on the data to be delivered and the paths/routes required to deliver it, the device-centric approach allows to implement protocols focused on local topology information gathered by the devices themselves. This may still help in achieving optimal results in terms of data transfer, while leaving most of the choices to the infrastructure user in terms of design and tradeoffs for networking protocols, when needed.

1.2. The SAaaS vision

We can thus envision a sensing Cloud as a further development of the IoT paradigm, reverting the approach into a top-down, logically centralized framework abstracting the underlying sensing resource infrastructure from the applications. It implements an elastic, on-demand provisioning of sensing resources by adopting service oriented solutions such as Sensing and Actuation as a Service (SAaaS) [13]. One of the key features for the adoption of an IoT-based device-centric approach is the management capabilities on sensing resources. Techniques for abstracting details and mechanisms away from heterogeneous hardware solutions, and to access, interact and communicate with other IoT resources are required indeed. A few of such functionalities, mainly related to device data management, interface and networking, may be provided by current state-of-the-art IoT protocols and relevant standards (M2M [6], COAP, SWE [5], GCM, etc.).

On top of such IoT-focused technological solutions, the SAaaS idea, at its core, is to gather and collect sensing and actuation resources from contributing devices and nodes among those which are IoT-enabled. In this sense, a volunteer-based approach could be adopted, in order to aid in the enrolment of both sensor
networks and personal or mobile devices, and building up of a sensing infrastructure for the provisioning of sensing resources for end users, in an Infrastructure as a Service (IaaS) fashion. Specifically, end users need to be able to handle, manage and customize (virtual) sensing resources at their will and according to their requirements, e.g. for augmenting an existing sensor network in order to cope with uneven coverage for a certain area, in a Cloudbursting fashion.

High level services could therefore either directly process sensed data retrieved by the underlying infrastructure, as in typical IoT scenarios, or enable innovative and pervasive applications by opening up physical infrastructure to end-users through their SAaaS powered devices, in an IoT-turned-utility landscape. Under these premises, on one hand the SAaaS may be considered as a new development for IoT, on the other hand, as a component of the IoT infrastructure, able to unlock novel features and boundless potential.

1.3. Contribution and contents of the paper

The main contribution of the paper is the design and implementation of an utility perspective of the IoT paradigm through a service-oriented SAaaS framework for sensing Clouds. To this purpose we focus on the basic mechanisms for enabling such utility view of IoT starting from the SAaaS concepts, artifacts and modules. Thus, we first discuss on how SAaaS goes to augment an IoT infrastructure, starting from the architecture down to the preliminary implementation of a prototype, including the basic SAaaS framework modules. We drew upon the IoT-A reference architecture, highlighting blocks, connections and interactions. Then, on top of IoT communication solutions such as the Open Geospatial Consortium (OGC) Sensor Web Enablement (SWE) [5] we design and develop the SAaaS abstraction and virtualisation functionalities hosted by the Hypervisor module [14]. A working implementation of such functionalities on Android mobiles is described in order to demonstrate the feasibility of the approach, also through preliminary results from performance evaluation efforts on an IoT application prototype.

This paper extends our previous work [13, 14] where the main idea of the SAaaS and its modular architecture is presented. Here we specifically further develop and contextualize such idea into the IoT scenario as discussed above, also providing practical details on their development and demonstrating the feasibility of the approach through an actual implementation. To the best of our knowledge, even if several architectures and infrastructures for IoT have been developed so far, as discussed above, the main contribution of this paper to the state of the art is based upon a vision for IoT, subverting typical assumptions about the paradigm by
adopting a top-down approach instead of the traditional bottom-up one. Based on this new perspective, we provide both an architecture and implement the building blocks of an IoT framework starting from the SAaaS artifacts and modules.

The remainder of the paper is thus structured as follows. Section 2 provides background concepts and an overview of the state of the art on the related work. Section 3 outlines our utility view of the IoT by merging the SAaaS framework architecture into the IoT-A reference one. Then the SAaaS building blocks and main modules are discussed in Section 4. A preliminary implementation of the SAaaS stack for Android mobiles is described in Section 5, reporting on development and testing of a use case. Final remarks and future work are discussed in Section 6.

2. Preliminary Concepts

2.1. Related Work

In this section we discuss some techniques and solutions dealing with IoT infrastructure issues, with particular reference to IoT architecture and sensing/Cloud related issues.

2.1.1. IoT Infrastructure

One of the most authoritative and enhanced work on this topic is performed by the IoT-A project [8], aiming at implementing interoperability of solutions at the communication level, as well as at the service level, across various platforms. The approach adopted, starting from existing literature, produced a reference model for the IoT domain creating a common knowledge base, and a reference architecture identifying the IoT building blocks for supporting interoperable and compliant IoT infrastructure solutions.

In a similar way the SENSEI project [2] aims at creating an open, business driven architecture that fundamentally addresses the scalability problems for a large number of globally distributed devices through the following four components: Resource Concepts, Support Services, System interactions, Resource and Information Modelling. In the CASAGRAS model [3], three layers are identified beneath the Internet communication zone: i) the physical layer, ii) the interrogator-gateway layer, and iii) the information management systems. Also the FI-WARE project [4] includes an IoT chapter which focuses on services enablement, providing a set of services to support applications consuming data from a range of different devices, based on the IoT-A reference architecture.
Several standard initiatives on IoT topics focus on the specification of an IoT architecture. One of the most authoritative is the ETSI Machine to Machine (M2M) [6], aiming at developing and maintaining an end-to-end overall high level infrastructure for M2M. The M2M architecture is specified according to a centralised RESTful approach, where applications can define resources (memory blocks that are located in the platform components deployed in the devices, gateways or central infrastructures), which can be used to exchange information between applications.

Another important attempt in such direction is the Sensor Web Enablement (SWE). SWE [5] is a standards’ suite for achieving abstraction and interoperability in sensor networks, widely used [15, 16, 17] in the implementation of sensing Web services as well as to address abstraction issues in virtual sensor networks [18]. It comprises several standards such as the Sensor Model Language for the description of sensor systems and processes associated with sensor observations; the Observations & Measurements to express observations and measurements through standard Web service interfaces; the Sensor Observations Service (SOS) to collect observations and system information; the Sensor Planning Service (SPS) to plan observations; the Sensor Alert Service (SAS) to publish and subscribe sensor alerts; the Web Notification Service (WNS) to asynchronously deliver messages and alerts from SAS and SPS.

All such initiatives adopt a layered modular approach in identifying IoT functionalities and blocks mainly at three levels: physical-node, network-infrastructure and service-application ones. But, as discussed above, the main goal of all of them is the connectivity and the interoperability of devices, as background for high level (data-centric) services and applications.

2.1.2. Sensing, Virtualisation and Cloud

IoT means meshing a networked environment for nodes out from anything, including daily life items, where semantic tagging is the missing link. The collective, distributed power pooled from resources attached to the Cloud may be useful to overcome typical constraints for smart devices in IoT scenarios, yet lack of context-awareness widens the chasm between mobile devices and elastic resources in the Cloud. A few approaches exist [19] for bridging the gap, still bindings are needed to provide mappings between physical environments as per the IoT and virtual environments in the Cloud [20], as described in [21].

Most of them just consider the Cloud as an extended application domain from which data are retrieved and pushed according to a data-centric approach, providing services for their management. Typical examples in such direction are
provided by the Sensing as a Service [10], Mobile Sensing as a Service [11],
Cloud-enabled Mobile Sensing Services [12], while the BETaaS project [22] aims
at implementing a platform for the execution of M2M applications, on top of ser-
vices deployed in a Cloud of gateways. Another interesting attempt to merge
potentials of sensing and IoT technologies with Cloud ones is the IoT6 project
mainly focused on “… research, design and develop a highly scalable IPv6-based
Service-Oriented Architecture to achieve interoperability, mobility, Cloud com-
puting integration and intelligence distribution among heterogeneous smart things
components, applications and services…” [23].

In fact, in heterogeneous sensing environments, to address interoperability and
communication issues calls for specific software abstraction layers [24, 25] as well
as to enable the dynamic reconfiguration of sensor nodes [26, 27], and to manipu-
late sensor data [28, 29]. Virtualisation has been proposed to enable seamless scal-
ability and interoperability among sensor node platforms from different vendors
through uniform management tools, by means of an abstraction layer between the
application logic and the sensor-driver [30], and a similar approach has been put
forward in the IoT domain [21]. In [25], a framework to enable management of
physical sensors on IT infrastructure, abstracting and virtualising them into virtual
sensors, in a Cloud-fashion, is provided.

Some solutions are specifically conceived for building up networks of mo-
biles’ sensors. The Mobile phone Sensor Network [15] allows to collect obser-
vations from Bluetooth-enabled sensors on mobiles and send them to a database
through an OGC SWE SOS. Similarly, [16] allows to perform the injected mea-
surements and to express them in SWE-compliant format, also resorting on the
mobile computing and storage resources. An infrastructure for connecting sensor
networks and applications is proposed in [31] allowing to select physical sensors
in wide-geographic sensor networks that can be implemented as suggested in [32].
A Semantic Web Architecture for Sensor Networks is proposed in [33], specifi-
cally dealing with inference on the sensor data collected by several heterogeneous
SNs. In [17] a service-oriented framework to integrate heterogeneous sensors and
virtual ones is described, starting from the SWE SPS, WNS, and SOS. In [34]
Android mobile sensors are categorized into two different classes depending on
the functionality provided (common an complimentary) and are considered as po-
tential providers of raw data. To recognize user context information the mobile
phone infrastructure of [35] can be used, e.g. for monitoring physical actions per-
formed by users such as walking and running. Similarly, [36] allows to recognize
physical activities of mobile phone owners-users.

The main difference between this work and the proposed approach is on the
provisioning model adopted. Here, a device-centric on-demand model is implemented, aiming at providing actual, customizable sensing resources after abstraction and virtualization of physical resources gathered by enrolling mobile and static nodes, according to a service oriented paradigm. Some basic functionalities such as those related to abstraction and virtualisation can be taken from existing work as discussed above, revised, adapted and extended taking into account our goals in relation to IoT domains. A contribution of this work therefore lies in rethinking and adapting existing solutions, mainly at lower layer, to demonstrate the feasibility of a utility paradigm for IoT.

2.2. Background

Here we provide some background concepts on which this work is based, describing the IoT-A reference architecture and introducing the SAaaS framework one.

2.2.1. IoT-A Reference Architecture

![Figure 1: The IoT-A Reference Architecture.](image)

The functional decomposition of the IoT-A reference architecture (RA) into seven functionality groups of components is depicted in Figure 1 and detailed in the following.
i. **IoT Process Management**: provides functionalities concerning the integration of (business) process management systems with the IoT infrastructure.

ii. **Service Organisation**: acts as a communication hub among the other functionality groups.

iii. **Virtual entity**: maintains and organises information related to physical entities, enabling discovery of services exposing resources associated with virtual and physical entities.

iv. **IoT service**: provides the functionalities required by services for processing information and for notifying application software and services about events related to resources and corresponding physical entities.

v. **Communication**: provides the set of primitives for device connectivity and communication as well as for content-based routing, providing a common interface to IoT services.

vi. **Management**: manages resources efficiently in terms of costs, unexpected events, fault handling and flexibility.

vii. **Security**: is in charge of ensuring the security and privacy of IoT-A-compliant systems.

The last two groups implement functionalities covering cross-cutting concerns and therefore are vertical, to interface with all the other layers.

### 2.2.2. The SAaaS Framework

The SAaaS reference architecture [13], shown in Figure 2, is composed of...
three modules: Hypervisor, Autonomic Enforcer and VolunteerCloud Manager. The Hypervisor allows to manage, abstract, virtualise and customise sensing and actuation resources that could be provided by enrolling either mobile device or SN nodes. The VolunteerCloud Manager (VCM) and the Autonomic Enforcer (AE) deal with issues related to the interaction among nodes in the sensing Cloud. The former is in charge of exposing the virtual sensing resources via Web service interfaces, indexing the resources, monitoring Quality of Service (QoS) metrics and adherence to Service Level Agreements (SLAs). The latter implements the enforcement of local and global Cloud policies, the subscription management, and provides facilities for the cooperation on overlay instantiation.

3. IoT-SAaaS Architecture

In this section we are going to describe how SAaaS may build an utility scenario for IoT, showing how it can be also considered as an extension to current IoT solutions, with specific regard to the IoT-A reference architecture described in Section 2.2.1. As highlighted in Figure 3, the high-level architecture of our SAaaS framework sits well within the comprehensive IoT-A reference one, a complete depiction of which is in Figure 1.

Before delving into the details about layering and interplay among blocks, it is important to stress the purpose of extending IoT-A with SAaaS, as can be seen upon closer inspection of the aforementioned diagram: the introduction of a brand new level between the communication one below and the service/resource-oriented one above. This allows to enrich the bouquet of functionalities provided by an IoT infrastructure based on and implementing the IoT-A reference architecture by also covering device management and on-demand, elastic provisioning of sensing and actuation resources, i.e. essential mechanisms for building up a sensing Cloud infrastructure.

This way, much beyond augmenting the existing levels with SAaaS-specific blocks, this approach goes to expand an already quite rich reference architecture with one of the few missing features to be expected from fully feedback-enabled sensing-oriented IoT implementations, namely exposing sensing and actuation resources in terms of infrastructure, available according to a Cloud-like provisioning model. Approaching services based on sensing from an IaaS angle pays off in many ways, such as the customisation and virtualisation possibilities it discloses, while multiplexing requests over available endpoints and dealing with hardware resource scarcity elastically as in typical Cloud fashion.
Figure 3: The SAaaS framework inside the IoT-A reference architecture.

Even more key is the benefit of enabling new business models by introducing brokers and other players that would base their services on disintermediating access to resources, typically for higher level service developers and providers to consume, in accordance with recent approaches and trends moving towards the software defined networking paradigm [9]. This model of interaction and management of resources does not interfere with the other layers from the service/resource one upwards, per the IoT-A RA, as the main advantage is in terms of abstracting away heterogeneous platforms and exposing data-driven services and resources while resorting to uniform access patterns to the underlying hardware.

This means that implementers of frameworks modelled after IoT-A RA would
benefit from such an addition too, yet for all the implied benefits nothing mandates its inclusion. The same applies for the blocks higher up in the SAaaS, as the presence of Hypervisor-enabled devices does not call for either Autonomic Enforcer or VCM-like facilities, especially if ease of development and abstracted mechanisms of interaction with heterogeneous platforms is what implementers mainly are after. Still, as soon as those components kick in, resources get to be exposed fully as an infrastructure Cloud, while still preserving the other mechanisms of interaction, such as pure data-driven and mining-related services, which simply run in parallel and keep open one more avenue for exploitation.

Moreover, as an alternative to side-by-side execution of IaaS and SOA stacks, the latter may be deployed over the former, and operate while backed by the underlying resources, according to the tried and true PaaS/SaaS formula. This is valid for any of the higher levels of the original IoT-A diagram, thus the ‘‘staircase’’ contour of the SAaaS section which features in the modified one. The aforementioned considerations translate into a not so strict positioning of the AE or VCM blocks, as those can be placed anywhere spanning the staircase, under any of the three levels featuring in the unmodified IoT-A depiction.

More specifically, as shown in Figure 3, the SAaaS Framework could be considered as the eighth functionality group of the IoT-a RA, interacting with the others in order to implement the sensing Cloud as a utility specialisation (a commoditisation) of the IoT paradigm. In detail, bottom-up, we have the Communication functionality group exposing communication facilities to either the IoT Service (ITS) one directly or through the SAaaS. The latter exploits ready-made protocol endpoints and offers handles on virtualized resources, for the ITS, the Virtual Entity (VE), the IoT (Business) Process Management (BPM) and the Service Organization (SO) functionality groups to consume. The ITS is meant for (historical) tracking and resolution of resource-based services, either exposed as physical ones (by the Communication layer) or as virtualized ones (by the SAaaS).

The VE functionality group is tasked with similar duties as the ITS, but relative to Virtual Entities and their establishment (through definition of semantic relations and physical world to sensing/actuation resource mapping), in this sense operating with help of either the ITS facilities, or directly leveraging the SAaaS VCM to discover and keep track of (virtualized) resources. As well, the BPM may extract models from running workflows and execute business processes based on those, after finding, binding, and invoking of services that are used for each process step, where it relies on SO to map the abstract process definitions to more concrete service invocations. In the end the binding may involve either VE-provided endpoints or the ones exposed from the SAaaS VCM, or even combine elements from
both sets if needed.

The SO instead may leverage resources across the board, either to provide a mapping service, as detailed above, or to compose and orchestrate services itself.

4. Building Blocks

Aim of this section is to provide details on the basic functionalities and modules that are key enablers for the IoT infrastructure utility paradigm. To this purpose we mainly focus on the low level of the SAaaS stack, the Hypervisor. It can be viewed as the crucial component of our device-driven approach to infrastructure-focused Clouds of sensors in an IoT utility perspective: it manages the resources related to sensing and actuation, introducing layers of abstraction and mechanisms for virtualisation.

It works at the node level, which could be either a whole sensor network, under a unique administrative domain, or a set of sensors, as built-in to a standalone device. In other words, referring to Figure 2, an SAaaS node could be either a whole sensor network, composed of several sensor boards or motes managed by a sink and/or a gateway, or a personal device that could be more or less smart and thus can be equipped with one or more sensors. The Hypervisor functionalities should fill this gap, dealing with such heterogeneity, hiding it to the above modules of the architecture. In the SN case, it is therefore necessary to split the Hypervisor architecture between the centralized element of an SN, the “node”, and the leaves of the topology, e.g. the motes. This way, all the motes composing an SN should have installed a specific Hypervisor module locally managing

Figure 4: The SAaaS Hypervisor modular architecture.
them, coordinated by the high level modules of the Hypervisor deployed on the node/SN gateway. This kind of two-level separation of concerns and assignment of operations descends also from the need for certain duties to be (self-)managed through autonomic approaches, typical of distributed entities.

A high-level, modular view of the Hypervisor architecture comprises four main building blocks: Adapter, Node Manager, Abstraction Unit and Virtualization Unit as shown in Figure 4.

4.1. Virtualization Unit

At the top of the layered model of the Hypervisor we have the Virtualization Unit, whose main task is slicing, i.e. generating possible partitioning schemes for the cluster of resources exposed by the Abstraction Unit, or the Node Manager underneath, if there is no need for aggregation of resources, e.g. when slicing degenerates to mere mapping. These partitioning schemes can be subsequently ranked according to a number of criteria including sensor provenance, proximity, QoS, security and so on.

4.2. Abstraction Unit

Below the Virtualization Unit is the Abstraction Unit. As can be seen comparing Figure 5a with Figure 5b, it replicates planning and observation facilities, modeled after those featured in the Adapter, but on a node-wide scale, combining the pool of resources of the whole SN. In particular, the Observation Aggregator exposes all resources from the nodes and the Planning Aggregator manages this set, sending combos, i.e. combination of commands, and tracking exit codes,
eventually reacting to (partial) failures by triggering apt adjustments. The Resource Discovery module, which offers an interface to the motes, actively gathering descriptions of underlying resources and forwarding the results to the Aggregator modules. The Customization Manager acts as an orchestrator for customization engines located on the motes.

4.3. Node Manager

Virtualization and Abstraction Units work on top of the Node Manager. It acts at node level only and is in charge of sensing resources operations and mandating policies, in cooperation with the Mote Manager inside the Adapter, which replicates its functionalities at mote level. In a standalone device the roles and functionalities of both modules are collapsed into the Mote Manager. It is important to remark that the depiction (Figure 4) of the Virtualization Unit is L-shaped because it can work directly over the Agent-hosting Adapter in selected cases, e.g. when dealing with a degenerate SN made up of a single mote.

4.4. Adapter

The lowest component of the Hypervisor, the Adapter, shown in Figure 5b, plays several distinct roles: first, only when deployed over motes, exposes a standards-compliant customer-friendly Interface to on-board resources.

It is also in charge of requesting, retrieving, and eventually pre-processing measurements, through the Observation Agent. As we are going to describe in the following sections, in terms of both behaviour and implementation details, the Planning Agent (PA) pushes requests for actions (tasks) to the device. The two agents rely on the presence of a platform-specific driver, the Translation Engine, responsible for converting the high-level directives in native commands. It is essential to remark that there is no overlap of duties here with the IoT-A, where the Communication components are tasked with abstracting heterogeneous devices and topologies only in terms of networking, leaving to the Translation Engine these same goals with regard to platform-level interaction and other low-level device- or operating system-specific details.

The Hypervisor, through the Adapter, is also in charge of processing requests to (re)configure the resource, using the (optional) Customization Engine, an interpreter able to execute on the sensing device the code needed to tailor the sensing activities to customer-mandated requirements. Finally, an autonomic approach is adopted delegating some management tasks of the Adapter to the Mote Manager running on the mote-side, performing specific operations such as power-driven self-optimization with the Node Manager.
Yet, the most important duty this layer-spanning module has to cope with consists in providing mechanisms for the customer to establish an out-of-band (i.e. not Interface- or Agent-mediated) channel to the system, for direct interaction with either the resources (e.g. for Agent-agnostic collection of observations) or low-level modules (e.g. the Customization Engine), thus pinning it as a mandatory component of the architecture.

4.4.1. Planning Agent

One of the core elements of the above system is the Planning Agent, providing very basic facilities for sensing resource management such as reservation of functionalities, tuning of parameters, scheduling of observations. These allow to manage the sensing resource operating parameters such as duty cycle, sampling frequency, etc.

The PA works side-by-side with the Observation Agent, complementing its features. Unlike the latter, engaged in providing upper layers with XML-encoded measurements (observations), sampled while driving the sensing resources, the former is mainly devoted to tune sampling parameters according to user-defined preferences, still to be interfaced with by means of extensible standard-compliant encoding of requests for tasks, and corresponding responses. Other than tuning, tasks for scheduling of observations can be consumed by the PA: it may be following a predefined schedule, or upon the occurrence of a particular event, or simply a request from a client. The main aim of this effort is exposing all underlying knobs to make them available for customers to operate on transparently. Although providing useful and standardized mechanisms, the Observation Agent is not strictly needed to let customers exploit sensing infrastructure. Indeed the PA is enough to handle the physical (or virtualized) resources as long as a working bidirectional communication channel is established between the client and the mote or mobile hosting the sensing device. Such a facility would then be enough to let the customer do what is needed for getting and storing observations, synchronously working over the channel if required.

In order to meet the aforementioned requirements, an architecture comprising the six modules shown in Figure 6 has been designed: a Request Dispatcher, a Sensors Prober, a Task Explorer, a Task Manager, an Observation Access Provider and an Interface. The Request Dispatcher has to identify and demultiplex a request to the modules underneath. The Interface has to interact with the Adapter services, i.e. the Customization Engine, the Translation Engine and the Node Manager.

The Sensor Prober is in charge of enumerating all the sensors and actuators within a sensors platform, however rich and complex, by low-level platform-
specific system probing. These sensors are then identified according to their types, supported observation facilities and sampling specs, overall (nominal) features and manufacturing details (brand, model, etc.). A further characterization of the Sensor Prober, in terms of approaches and platform-specific technologies, follows in Section 5.1.

The Task Explorer is responsible for enumeration of available tasks, to be provided by probing sensors as listed by the aforementioned Prober. In terms of tasks, those related to parameter tuning for sensing resources differ logically, according to sensor type and technology, so it is possible to e.g. plan retrieval of temperature samples from a thermometer, once a certain threshold has been exceeded, change the relative position and the focal length of a camera, or simply schedule reading of sensor observations at fixed intervals, etc. Moreover, in order to assess feasibility of a certain task, among the ones enumerated for selection, the sensor has to be queried and provide (runtime) confirmation, or else denial, of availability for servicing (or reservation thereof). It is then up to the querying party to decide what to do after feasibility assessment for the task under consideration.

The Task Manager controls tasks’ lifecycle, since feasibility assessment through reservation/submission stages, then following up, and acting upon, running task progress. Due to the number of, and dependencies for, the operations involved, as described in greater detail in Section 5.1, the Task Manager duties have been assigned to six modules as shown in the architecture of Figure 7. Two of them are mandatory, the rest are optional.

Task Submitter and State Controller implement mandatory functionalities. Their
roles are, respectively, to enable users to set all (mandatory) parameters for a specific task before submission to a sensor, and submit it when ready, and to follow up the processing of the task, alerting any agent, subsequently querying about availability for task execution, about its (busy) status until completion. The optional modules are instead: Reservation Manager, Feasibility Controller, Task Updater and Task Canceller. These modules provide additional facilities for control on running (or yet to be scheduled) tasks to process.

If needed, a user may reserve a task for a period of time, during which he/she gets exclusive access to the underlying resource, as no other user can submit or reserve it. The task will then be executed as soon as the user confirms for the real processing stages to commence. The Reservation Manager is responsible for both reservation of tasks, and its confirmation. The Feasibility Controller has to check if a task is feasible, as detailed above. The feasibility of a task depends on the availability of any resource essential for task servicing, e.g. if not still allocated due to a previous request.

Then, the Task Updater is in charge of updating configuration parameters of a task, if some modifications have to be pushed after tasks enter into processing stages. Lastly, the Task Canceller empowers users to stop and therefore retire a task, when already submitted or under reservation.

Finally, once a task has been serviced, the resulting observation gets stored. Any observation will be accessible through the Observation Agent only. In terms of observations, the sole duty up to the PA lies in the Observation Access Provider ability to provide endpoints to access measurements. Being dependent on the Observation Agent, it is an optional component, required only if the latter is implemented in the Abstraction Unit.

5. Implementation and Testing

In this section we deal with the implementation of a prototype for basic functionalities at the core of the IoT-SaaS stack. We first discuss some technical issues to be tackled in order to enable a platform implementation, which has been developed and tested as a prototype on Android mobiles. Details on technical issues related to the implementation and some preliminary results from testing are described in the following.

5.1. Platform-specific Enablement

As outlined above, the Node Manager of Figure 5b is mandatory exclusively for its role as an out-of-band conduit, e.g. to the Customization Engine, where
otherwise the only way to interact with sensing resources is mediated by the PA, leaving no path to mould the platform itself to the sensing requirements at hand. In terms of the Node Manager, a way to establish such an out-of-band channel may be either under the guise of a Communication layer-provided implementation based on IoT-A RA, or through platform-provided facilities, i.e. Google Cloud Messaging under Android.

Besides obvious considerations of widespread availability and abundance of hardware diversity, our choice to focus on Android first was also due to the kind of work needed for platform enablement with regard to hypervisor-related core features (abstraction/virtualization). Indeed what sets such a kind of platform apart from those (Contiki, TinyOS) typical of embedded environments, in use for WSN-based scenarios, is the availability by default of uniform APIs specifically devoted to sensing, coupled with restricted access to underlying devices, in order to funnel all interactions with those through the aforementioned interfaces.

Anyway, most efforts in literature about abstracted access to resources and uniform approaches to coding for mote-class environments are based on some kind of constrained runtime environment. This usually takes the form of a minimal Java-based VM. Future work may tackle such device families as well, by leveraging these kind of enablement technologies.

This difference can play out both as an asset and as liability, but when certain usage scenarios, e.g., low-level access to resources, turn out to be unanticipated, or undesirable by default, and thus not provided for by the platform itself, beating the environment into shape is not trivial. Indeed it gets harder than simply coding from scratch to provide required APIs, as changes need to be kept to an absolute minimum in order to avoid fragmentation and introduce further hurdles to a proper setup and wide adoption.

With regard to the Translation Engine, any sensing-related APIs provided by the node should be used when available, since developing the Agents against those frees us from the need to implement a layer for command translation. The same considerations apply for the Translation Engine interactions, yet this component cannot be fully disposed of, even when in presence of a platform-specific, but general-purpose, API for sensing, such as the Sensing APIs under Android. This requirement stems from the fact that not all onboard devices, which could be leveraged for sensing, are exposed as such, i.e., under a uniform, platform-level collection of APIs. This way the Translation Engine exploits the platform-standard (e.g. Android and/or Linux) low-level tools and mechanisms in order to export these resources under an extended set of devices the sensing APIs know about. This abstraction eases the job of the Sensor Prober, e.g., just letting it enumerate
resources through the Sensing APIs.

Moreover, when we discuss about getting low-level access to a sensing resource, we mean achieving the highest level of control the infrastructure machinery enables us to exert on the device itself. This implies at the very least engagement of interfaces to OS-level facilities. Usually this may be realized either directly interacting with the driver (e.g. by device-specific system calls or IOCTLs), or mediated by standard library routines or even, as a last resort, just leveraging ready-made low-level platform-specific tools for the device under consideration. As is the case for Android, these may be generic Linux-based utilities, readily available when under a platform running over a compatible kernel.

For instance, under Android a few (even native) tools, either underlying the SDK APIs, or easy to compile in with the NDK, are really straight ports of their desktop counterparts, such as

- wireless-tools, for userspace Wifi management, not included by default; and/or
- the BlueZ stack, comprising kernel modules and userspace libraries for the whole family of Bluetooth protocols, leveraged by the SDK.

Let us now make an example, based on BlueZ, aiming at

- i. directly programming against kernel-side BlueZ-provided IOCTLs;
- ii. resorting to basic POSIX-compliant functions and data structures, as exposed by the available standard library (e.g. Bionic, under Android);
- iii. just developing against the BlueZ library.

There are a variety of needs driving the adoption of each of the aforementioned approaches:

- leveraging IOCTLs for rewriting, or expanding, parts of the Bionic and/or the (userspace) BlueZ stack;
- as an aid, at least letting standard C library functions wrap syscalls and take care of low-level routines for the developer to focus on customization of BlueZ itself;
- just leaving the stack alone, simply exporting outright new, higher-level facilities, e.g. an in-house developed communication protocol geared towards niche use cases.
As well, there are several reasons to avoid reliance on higher level APIs like e.g. SDK-provided ones under Android. One could be the (oversimplified) assumption that the user is just interested in data, possibly not even raw but only undergoing preprocessing and filtering, to the point of losing any info on the originating device and its precision, i.e. Wifi and/or cell-assisted geopositioning APIs. Another one could be the (implicit) ruling out of feasible exploitation of hardware for unplanned activities, thus leaving out customization of low-level functionalities, possibly with no need for user-unfriendly procedures like substitution of kernel modules, e.g. converting the BT or Wifi module in radio/protocol scanners and penetration testing devices, with Linux-derived tools such as bluediving or aircrack. Or also, in general, the possibility to expose, under a common API (e.g. sensing subsystem), hardware otherwise available under a totally unrelated system category, e.g. communication-oriented devices such as Wifi or Bluetooth modules.

We have already highlighted the advantages of a device-centric approach to provide infrastructure under the guise of sensor Clouds, unfeasible when tackled under classic data-oriented schemes, while still keeping with typical IaaS requirements, e.g. (physical) resource sharing and usage maximization on one hand, isolation and SLA-mandated QoS on the other. Just to recap:

- (direct or mediated) handling of device knobs and tunables;
- virtualizing (e.g. augmenting) resources;
- repurposing, i.e. shifting device usage towards patterns unaccounted for by the original equipment manufacturer;
- (ultimately even) customizing (parts of) the platform itself (e.g. kernel modules), by leaving an out-of-band channel open for deployment.

All of the above translated in a pressing need for a module able to probe the system for low-level facilities, e.g. kernelspace componentry, C-library device-specific routines and stack tooling, to be then exposed for device-oriented consumption by SAaaS customers.

Even more convenient is the adoption of the SAaaS approach if considered as a utility projection of the IoT paradigm. For example, even beyond the aforementioned out-of-band channels, certain in-band IoT-related M2M protocols could be injected through platform customization to augment sensing resources, e.g. publish location status to an MQTT broker [37] as a way to decentralize access to
data previously available through a Google-provided service (Latitude) now de-
commissioned. An example on how to exploit the IoT-SAaaS stack in the mobile
crowdsensing [38] application domain is documented in [39].

5.2. Prototype

A first prototype implementation of the IoT-SAaaS basic functionalities has
been targeted to mobiles equipped by Android OS 4.0, using the NDK developer
libraries and API provided by the Android developer community [40]. The core of
this effort is based on the SWE Sensor Planning Service (SPS) 2.0 standard [41]. It
enables the interaction among user clients and sensor and actuator services using
XML schemas to submit requests and to allow the service to reply. Modeling
behavior after the SPS standard, the functionalities of the Sensor Prober, Task
Explorer, Task Manager and Observation Access Provider modules described in
Section 4.4.1 have been developed.

The Sensor Prober has to retrieve information regarding: i) the contributor, if
available (the extent of such information disclosure is totally up to the contribu-
tor); ii) the node sensors and their descriptions, also including the measured phe-
nomenon and corresponding metrics; and iii) the geographic area (range) inside
which observations are significant. This feature is implemented by the SPS Get-
Capabilities primitive. A GetCapabilities request is composed of four sections.
The first one is ServiceIdentification containing the contributing node metadata,
i.e. generic info on the type of the node, brand, model and similar. Then the
ServiceProvider section provides information on the contributor, if available and
public. The third section is the OperationsMetadata one, with metadata about the
operations specified by the service and implemented by the node. The last is the
Content section, containing metadata about the sensors provided by the smart de-
vice through the PA and the communication mechanisms supported (XML, SOAP,
etc.).

The Task Explorer retrieves the list of tasks that can be performed on a sensor
through specific SPS DescribeTasking requests. A description of the available
configuration operations for the sensor is thus obtained and provided to the Task
Manager. The request just contains a Procedure element to enquiry a sensor in the
list about the tasks that can be performed. The tasks are identified by the name,
the description, and the capabilities’ configuration information.

The Task Manager implements a set of SPS requests. The Submit one allows
the user to launch the execution of a configured task. Eventually, before to submit
a job request, it is possible to enquire about its feasibility through the GetFeasi-
bility primitive. The reply can be “Feasible” or “Not Feasible” and, optionally, it
may contain a list of alternative sets of tasking parameters that might help to the reformulation of a request. The user can also reserve the resources required to perform a specific task and then launch the task through the Reserve and Confirm requests. In a Reserve request an expiration time has to be specified. At expiration time, all the reserved resource are released if the task has not been confirmed. It is possible to check the status of a task using the Status request. A task can be in six different states: “In Execution” if the service is executing it, “Completed” if it was completed as planned, “Reserved” if it has been reserved, “Failed” if execution fails, “Expired” when the task reservation expires and “Cancelled” if the task was cancelled. The framework can eventually update or cancel a task, with the Update and Cancel requests respectively.

Finally, the Observation Access Provider in the PA aims at providing the framework with mechanisms, if needed, and endpoints to access the observations and measurements obtained during execution. It implements processing of SPS DescribeTaskingResult requests to interact with a specific sensor or a specific task.

5.3. Use Case

To test the SAaaS basic modules and functionalities described in Section 4 and developed as discussed above we implemented a use case in a typical IoT scenario, the Smart City. This use case concerns a smart surveillance application involving smartphones in a specific area. It is necessary to provide the user with facilities for manipulating the camera, in particular to focus on a specific point by zooming in/out the camera. We have therefore implemented a mock SAaaS provider that enrolled just one contributing node. This is an LG Optimus L9 P760 smartphone with Android 4.0 with a dual-core processor TI OMAP 4430 clocked at 1Ghz with GPU PowerVR SGX540, 1GB RAM, 4GB intern memory, a 5Mpx APN, a visio VGA camera and a 2150mAh battery. The user also interacts with the SAaaS system trough a mobile on which a specific SAaaS Framework client has been deployed. It has been implemented and deployed on a native instance of the Android platform running on an x86 PC.

In the following we show the interactions between the user and the contributor through the SAaaS stack in a smart surveillance application, mainly focusing on the behavior of the core components of the implementation, by investigating a simple user-SAaaS acquisition-interaction-observation access workflow.

5.3.1. Acquisition

At the beginning, the user has to select the kind of sensor required, e.g. a camera, also specifying the area of interest and the specific operations/tasks to
perform (zoom-in/out). The request is submitted to the SAaaS that performs the
device, sensor, task discovery according to the user requirements. At this point
we need to step back a moment and illustrate another advantage of the approach.
When developing this kind of app (e.g. surveillance) and looking for infrastruc-
ture, more than just an enrolled device equipped with zoomable camera, among
requirements we should consider that it has to be one actually ready to capture,
and the end user should be able to express this constraint accordingly. In practical
terms this means that devices which are not pointing anywhere, e.g. put away in
a pocket or bag, or whose stream is otherwise totally unusable, e.g. extreme dark
surroundings and no lighting aids, should be filtered out by the framework itself,
in presence of such requirement. Abstraction plus virtualisation are there to cope
with this kind of issues, by exposing useful virtual capabilities e.g. “surroundings
capture readiness”, even only for the resource acquisition phases.

In terms of concurrent requests and multiple access, even if not dealt with
in this work, the idea is to let the user select whichever subset of choice of the
resource under consideration, e.g., in this case a camera-provided video stream
and actionable parameters if available. Therefore a user may choose the whole
video stream, including all knobs of the camera subsystem, or she can opt for just
a sliced subset, e.g. a smaller tile over the whole framebuffer. In the latter case,
the framework would mediate access to both the data and the device parameters.
For instance, if a user is the first to ask for camera-related resources, by opting for
a tile of the whole stream, she may still obtain full control of zooming capabilities,
if allowed per provider policy. She may even opt for not being interested in these
capabilities at all. An option may be about flagging zooming as a “don’t care”
where it applies, i.e., any possible zooming actions are fine for that user, because
those would not interfere anyway with the expected kind of usage. Yet, subsequent
users would not be able to access to that specific tile, and may only obtain tiles
with either full zooming capabilities, or reduced ones, where not absent, or even
subject to third parties, depending on previous bookings.

The resource acquisition is triggered by the end user by sending a request spec-
ifying the required sensing resource and a list of parameters and tasks this should
be able to implement/provide. In the smart surveillance use case the end user
submits to the SAaaS Framework a request for a camera with zoom configuration
capabilities. The SAaaS Framework thus sends a resource discovery task to iden-
tify the nodes with camera and zoom facilities on-board. If the discovery detects
the presence of a camera on a contributing node, the framework asks that node
information on the tasks provided by the camera. Once the task list is received,
the SAaaS Framework probes the list to find the tasks required by the end user.
In our scenario, through the sensor probing the system finds a device equipped by a camera that matches the requirements, the LG Optimus L9 smartphone as shown in Figure 8a, and thus forwards the full list of tasks it can perform on the camera to the framework for probing on task capabilities. Otherwise, in case of no requirement matching, the system provides no results as depicted in Figure 8b.

5.3.2. Interaction

Once the sensing resource, the LG Optimus L9 smartphone camera in the smart surveillance use case, is acquired, the direct end user-sensing resource (camera) interaction is triggered. It is composed of two steps: submission and management. The submission step is performed in the case a result has matched the request sent by the user during the Acquisition phase. We can see on the two screenshots of Figure 9 the list of tasks matching the user request provided by the L9 smartphone camera. The tasks thus identified are grouped into three sets: parametrization tasks (set the flash mode, set the effect on the image, set the anti-bandning, set ...), activation tasks (enable or disable the default shutter sound when taking a picture, start and stop the autofocus, ...) and observations & measurements tasks (take a picture, record a video).

In the first case (parametrization), we can imagine the user wants to change
one of the camera parameters. He/she can select among the “Auto” flash mode, the “Sepia” effect, the “Infinity” focus mode, the “Night” scene mode, etc. In the activation case, the user would like to start/stop, enable/disable one of the tasks in the list. This request is submitted to the sensor which is in charge to activate/deactivate, enable/disble the targeted feature in the peripheral. In the example, the user could take a picture or record a video on the camera. The main difference with the activation tasks is the presence of a result or observation to be made available for access by the user, as soon as the corresponding sampling operations described in the task have taken place. Figure 10 shows the configuration and the submission of a video recording task scheduled at a specific hour.

Once the task is submitted to the contributing node, and thus forwarded to the camera for processing, the user receives a notification specifying that the submission was successful, or that an error occurred. If the task was successfully submitted the interaction management step starts, and the user can ask for the current status of the task. If the task is already in processing stages, the user is still able to act on it. For example, he/she can update or cancel the task by sending either a new configuration or a cancellation request, respectively.
5.3.3. Observation Access

At this point, the task was fully executed by the sensing resource camera. In the case of a parametrization task or an activation task no interesting values are returned and thus have to be retrieved by the user. On the other hand, the results of an observations & measurements task, the sensed data, are usually required by the user and thus retrieval is supported by the framework. Therefore, the user may request a pathway to the video just recorded, receiving the endpoints, needed to access the data, from the Observation Agent. Figure 11 shows the screenshots of the smart surveillance app recorded video request-reply through our framework.

5.4. Evaluation

It is not trivial to settle on the kind of meaningful non-functional properties to be assessed for such a system, where there are no tools or frameworks implementing the same kind of functionalities, to the best of our knowledge. In this section we evaluated the impact and performance of the SAaaS-enabled smart surveillance app from two different perspectives: the contributor and user ones. Since the resources on a mobile are usually limited in terms of computing power, battery power and network bandwidth, from the contributor viewpoint the interest is mainly on evaluating the impact of the SAaaS Framework on the node, in
order to quantify the cost of its modules on the smartphone. On the other hand, the user is interested mostly in evaluating the performance of a request, roughly quantified by the time to submit, process and return the results of its processing. In order to do that we evaluated all these parameters from the two different perspectives by considering a whole request-reply, which, according to the above description, implements the user acquisition-interaction-observation access workflow. Referring to the SAaaS Framework implementation, it implies a sequence of GetCapabilities, DescribeTasking, Submit and Observation Access requests as discussed above. To obtain significant values reducing the error probability we repeated each test 1000 times, collecting the corresponding results to obtain the parameter $x$ mean time $\mu$ and the 95% confidence interval offset $\gamma$, i.e. with confidence level of 95% $x$ lies in the confidence interval $\mu - \gamma \leq x \leq \mu + \gamma$, for each measurement as described in the following.

In any case it is important to remark that all evaluation efforts are mostly agnostic about the state of the network, on purpose. Indeed, Android is an advanced enough platform to expect the availability of powerful abstractions for the networking and transport layers, and any disruption or dropout due to handoff/over mechanisms would have the same kind of effect (e.g. short bursts of delays or
timeouts) and impact as any other generic Internet-aware app, considering there is no strong coupling between the behavior of the framework and the level of connectivity.

5.4.1. Contribution Overhead

In this section the overhead of the operations involving the SAaaS modules are evaluated with regard to computing, network (bandwidth) and power resources, which are usually scarce when dealing with mobiles. This kind of evaluation is quite common in literature [42, 43, 44, 45], helping as a way to assess overhead of a framework against baseline resource usage profiles, including those of a generic app. All the parameters thus obtained by the evaluation are reported in Table 1 and discussed in the following paragraphs, in particular in comparison to the ones obtained by an Android virtualization framework in [45]. Even if the scope and functionalities of our work do not fully overlap with those of the cited framework, the same evaluation approach may apply in our case, just as a basis for comparison.

<table>
<thead>
<tr>
<th>Parameter/Statistic</th>
<th>Battery Depletion</th>
<th>Network I/O</th>
<th>CPU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Energy (J)</td>
<td>Power (W)</td>
<td>Data (KB)</td>
<td>Bandwidth (KB/s)</td>
</tr>
<tr>
<td>μ</td>
<td>2.837</td>
<td>0.089</td>
<td>63.18/60.95</td>
<td>2.046/1.97</td>
</tr>
<tr>
<td>γ</td>
<td>0.087</td>
<td>0.00013</td>
<td>0.092/0.088</td>
<td>0.0032/0.0030</td>
</tr>
</tbody>
</table>

Table 1: Contributing node framework parameters, as obtained through the experiments.

**Battery Depletion.** The battery depletion due to the smart surveillance application has been measured through the PowerTutor diagnostic tool for analyzing the system and the running app power consumption. The results thus obtained are shown in the first column of Table 1. Both the energy and the power consumed during the whole smart surveillance app-SAaaS Framework interaction on the contributing node are evaluated. The energy footprint of the SAaaS Framework implementation is 2.837 J or, in terms of power, 89 mW. This means that the impact of the SAaaS Framework on the contributing node/LG9 smartphone in terms of battery depletion can be quantified in about 1%, against 2.4-3.8 % obtained in [45] through different benchmarks.

**Network.** The values in the second column of Table 1 reports the network statistics for the framework related to the smart surveillance app workflow, i.e. the amount of data received and transmitted during the whole interaction. To get this
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCapabilities</td>
<td>µ 383.3</td>
<td>294.13</td>
<td>89.17</td>
</tr>
<tr>
<td></td>
<td>γ 18.67</td>
<td>14.32</td>
<td>3.33</td>
</tr>
<tr>
<td>DescribeTasking</td>
<td>µ 381.5</td>
<td>296.97</td>
<td>84.53</td>
</tr>
<tr>
<td></td>
<td>γ 17.84</td>
<td>15.35</td>
<td>2.38</td>
</tr>
<tr>
<td>Submit</td>
<td>µ 586.3</td>
<td>229.64</td>
<td>356.66</td>
</tr>
<tr>
<td></td>
<td>γ 29.13</td>
<td>12.56</td>
<td>16.93</td>
</tr>
</tbody>
</table>

Table 2: Performance parameters, as obtained through the experiments.

information, we used the DDMS tool provided by the Eclipse ADT Plugin, using the Network Statistics service to obtain the data received and transmitted by the contributing node. In the table, the amounts of data received/transmitted in terms of Kbytes are shown as well as the bandwidth. We can observe that the traffic generated by the app is very low, and it is well balanced between in- and out-bound (about 60 KB each). It is even more clear when considering the bandwidth, which is about 2KB/s.

CPU. The third column of Table 1 contains the amount of CPU time, in ms, spent by the SAaaS Framework running on the contributing smartphone and subject to the smart surveillance app workflow. To get these value we used the Android debug class adding specific debug instructions in in the code. The log thus obtained have been collected into a (binary) trace file to be read through specific tools such as Traceview. This way we have quantified the impact of our framework in managing the smart surveillance app in less than 3% (2.66%) on the CPU utilization, against 1.8-3.9 % obtained in [45] through different benchmarks.

Memory. The last column of Table 1 reports the values of the memory tests performed on the contributing node. In the tests we used the Eclipse ADT Plugin DDMS tool. It implements the System Information module, providing information about CPU workload, the memory usage and frame render time. Through the memory usage, we evaluated the impact of the processing of requests for the SAaaS Framework due to the smart surveillance app. As above, in this case the app impact is very low, since the app requires an average amount of memory of 1.54 MB. A whole VM in [45] requires 64 MB.

5.4.2. Performance

From a different perspective, the end user and the SAaaS provider are mainly interested in obtaining information on the behaviour of the overall system, in order to know if and when it provides a response, i.e. the time spent in a request-reply.
Since in our experiments we implemented a toy SAaaS infrastructure composed of just one contributing node (the LG 9 smartphone), we are not able to provide useful information on reliability, which is out of the scope of this work. In this paragraph we therefore focus, on one hand and from a provider viewpoint, on investigating and characterizing the SAaaS Framework overhead with specific regard to the node-side abstraction layers, on the other hand, from an user point of view, on profiling app performance.

To this end we evaluated the smart surveillance app performance measuring the time to send a specific request to the contributing node. Table 2 shows the values thus obtained. More specifically, the response times of GetCapabilities, DescribeTasking and Submit requests have been evaluated at both the user side, in relation to request submission, and the contributing node side, for request processing. To obtain those measurements we added timestamping to the operations under evaluation, calculating the delay between transmission of requests and corresponding reception of replies.

From such results we can observe that, in general, the user requests get always serviced in much less than 1 s. The highest latencies have been experienced for the processing of a submission (586.3 ms), while those related to retrieval of capabilities and tasks almost match (∼380 ms). Looking at the contributing node service time values the condition is reversed: the submission processing is faster (229.64 ms) than the GetCapabilities and DescribeTasking ones (∼290 ms). As shown by the third column values of Table 2 this means that the SAaaS Framework overhead on task submission requests is higher than the other. This difference lies mainly in submission being really a three-fold operation, comprising the GetFeasibility, the Reservation, and the Submission proper, stages, by themselves much simpler in terms of (contributor-side) processing and exchanged data; moreover, these requests require further (framework-side) processing (mapping, configuration, ...) with respect to the other operations where the abstraction modules are involved. From the measurements we performed it is not possible to quantify such overhead, since the values also include network delays, yet these measurements still provide insight for qualitative considerations as the one above.

6. Conclusions

IoT is a paradigm attracting increasing attention and accumulating efforts in several ICT scenarios. Among those, many are focused on enabling flexible, ubiquitous, mostly unassisted interaction among devices over Internet, implementing advanced management of data flows upon which to build (web) services. Com-
plementary to such data-oriented perspective on IoT, in this paper we propose a utility/device-centric take on IoT, where sensing and actuation resources, exposed in compliance with IoT-based standards, are to be provided elastically and on-demand to end users, according to a service oriented provisioning model. This approach draws the IoT paradigm towards the Cloud, thus envisioning a sensing Cloud as a further step in the development of IoT.

In this paper, we explain how to enable such vision, e.g. for IoT as utility, by adopting the SAaaS approach. The SAaaS framework architecture gets here integrated into the IoT-A reference architecture, to demonstrate that a sensing Cloud can be considered as an extension to the kind of services the IoT paradigm enables. A high-level outline of the modules, required for a basic implementation of the SAaaS, follows, considering the interactions with IoT modules, technologies and solutions, such as SWE. Accordingly, such modules have been therefore implemented on Android-based mobiles in order to assess the feasibility of the approach. The results obtained by testing this preliminary implementation are then discussed to evaluate the suitability of the solution for actual, device-centric, applications and services.

The development of enhanced functionalities such as virtualisation and customisation, exploiting features provided by IoT-related techniques, including standards such as COAP, as well as umbrella initiatives such as those pertaining to M2M, are under investigation and thus stuff for ongoing and future work.

Acknowledgement

This work is partially funded by PhD programme under grant PON R&C 2007/2013 “Smart Cities” and by Simone project under grant POR FESR Sicilia 2007/2013 n. 179.

References

URL http://www.gartner.com/technology/research/top-10-technology-trends/


[19] T. F. Bissyande, L. Réveillère, Y.-D. Bromberg, J. L. Lawall, G. Muller, Bridging the gap between legacy services and web services, in: Proceedings


URL http://dx.doi.org/10.1007/978-3-642-21535-3_14

URL http://mqttitude.org/


