Everything as a Resource: Foundations and Illustration through Internet-of-Things

Blind Review

Abstract

This paper presents Everything-as-a-Resource (*aaR) as a paradigm for designing collaborative applications on the Web. Abstracting these applications' various physical and logical entities, resources are defined in a way that permits their discovery, composition, and participation in business scenarios. Compared to Everything-as-a-Service (*aaS), resources are categorized into computational, consumed, and produced, have trackable lifecycles as per their respective category, and are customized in order to consider the characteristics of future resource-based collaborative applications to develop. From a capacity perspective, a computational resource processes data, a produced resource abstracts data, and a consumed resource captures data. Along with their capacities, resources expose methods that other resources and/or applications' stakeholders call. The proper call of methods is ensured through restrictions like limited and non-shareable. This paper exemplifies the *aaR paradigm with a case study that revolves around the use of Internet-of-Things (IoT) in the healthcare domain. The case study is implemented in a RESTful fashion along with some standard Web technologies and protocols. The evaluation of IoTR4HealthCare system is benchmarked against two existing systems using cost and latency criteria.

Keywords: Everything-as-a-Service, Everything-as-a-Resource, Internet of Things, Healthcare, Resource, Restriction.

1. Introduction

In the Information & Communication Technologies (ICT) community, *aaS is a well-known acronym standing for Everything-as-a-Service [12]. Everything (i.e., thing as a general term) could be software, platform, infrastructure, communication, data, just name it[1]. Exposing things as services has different advantages such as abstracting the complexity of the digital and physical worlds, complying with the separation-of-concerns principle [35], and shifting the burden of managing things internally to external bodies (e.g., cloud providers) in-return

*Corresponding author; x@x.com

[1]Software, platform, and infrastructure are the essence of cloud computing.

of a fee. In this paper, we promote a broader vision of things by treating them as Resources (Everything-as-a-Resource *aaR). Contrarily to services that act as proxies over things independently of these things' characteristics like types and roles, we (*i*) make resources the pillars of an ecosystem that capture stakeholders' needs, requirements, and concerns, (*ii*) categorize (specialize) resources into computational, consumed, and produced, (*iii*) associate resources with separate lifecycles that stress out their differences from an operation perspective, and finally, (*iv*) customize the ecosystem's resources and stakeholders with respect to the characteristics of the future resource-based applications that can be developed upon this ecosystem. Resources are more than proxies but active components that can be specialized and customized. The connection between the three categories of resources is straightforward: a computational resource that is invoked at run-time, could consume (existing) resources and/or produce (not necessarily new) resources. Examples of resources include REpresentational State Transfer (RESTful) service as a computational resource, XML document as a consumed resource, and txt file as a produced resource.

The concept of resource is not new in the literature and has been used in different disciplines like distributed artificial intelligence (e.g., resource logic for multi-agent planning [10]) and service computing (e.g., RDF for interoperability [41] and REST for building applications (Web, Intranet, and Web services) [13]). While some disciplines consider that resources (whether logical or physical) are abundant, we argue the opposite. According to Dimick, all resources are expected to decline [11]. To deal with the non-abundance concern and to ensure proper use of resources, we define restrictions over resources and specialize these restrictions into limited (*versus* unlimited), non-shareable (*versus* shareable), and non-renewable (*versus* renewable). Restrictions permit to differentiate resources from services further and to have a better control over resource engagement in future resource-based applications. In this paper, our objectives are: (*i*) develop an *aaR framework that provides the necessary guidelines for setting-up and managing an ecosystem of resources, (*ii*) define both resources and restrictions regardless of the applications to deploy over this ecosystem, and (*iii*) exemplify the *aaR framework with a resource-based Internet-of-Things (IoT) case-study.

There are no doubts that Mark Weiser's statement about ubiquitous computing, "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it*" [46], has become a reality thanks to IoT and other forms of computing like ubiquitous [3]. Network high-connectivity and -bandwidth have allowed various digital devices, whether wearable or not, to form an ecosystem that includes other non-digital devices like white goods and medical equipment. According to Gartner[2], 6.4 billion connected things were in use in 2016, up 3% from 2015, and will reach 20.8 billion by 2020. In response to IoT intrinsic characteristics like ubiquitous sensing, thing diversity, dynamicity, and integration of the physical-

[2]www.gartner.com/newsroom/id/3165317.

and cyber-worlds together [2], we demonstrate how our ecosystem of resources transparently accommodates these characteristics. To this end, we propose a three-layer representation for the *aaR framework: application, resource, and infrastructure. The layers feature building blocks and operation modules that are necessary for setting-up and functioning both the ecosystem and the future IoT applications. It is worth mentioning that non-IoT applications could be considered without impact on the *aaR framework's concepts and principles.

The remainder of this paper is organized as follows. Section 2 is an overview of the fields of resource and IoT. Section 3 details the *aaR framework in terms of representation, resource categories, and restrictions over resources. Section 4 implements the *aaR framework through a healthcare-driven case study and evaluates the implemented system against two IoT healthcare systems. Finally, Section 5 concludes the paper along with discussing some future work.

2. Background

This section provides an overview of the concept of resource along with some definitions and examples of IoT uses. Then, it concludes with some highlights of the *aaR framework.

2.1. What is resource?

In ICT disciplines like computer science, information technology, and software engineering, resource means different things. It refers to hardware and software (e.g., network, server, storage, application, and service) in cloud computing [37], person/machine who execute tasks in business processes [32], Web content in the (Semantic) Web architecture [4, 13], etc. Despite the multiple uses of resources, they abstract some entities, whether physical or logical, that could be discovered, composed, and consumed so that certain business goals are achieved. It is worth noting that such entities might not be abundant and thus, restrictions on their use are deemed necessary.

Fielding proposes REST as a style for designing and developing distributed hypermedia systems like the Web that would comply with Resource-Oriented Architecture (ROA) principles [13]. According to Lucchim et al., resources are *directly-accessible components handled through a standard common interface* [27]. This interface is a set of stateless operations (e.g., HTTP methods). Systems that comply with Fielding's REST style are called RESTful. Resource has become prevalent when RESTful services have overtaken SOAP-based Web services [1, 36]. The former expose their operations as resources that are accessed through Uniform Resource Identifiers (URI)s and thus, are better suited for the adhoc deployment of systems over the Web. The latter suffer from the complexity of their protocol stack (UDDI/WSDL/SOAP to facilitate discovery/publication/invocation) and excessive number of standards (WS-$*$) [43]. According to Richardson and Ruby, ROA allows different representations of a resource through different URIs. This allows clients and servers to communicate through these representations [38].

Xu et al. adopt REST principles and service-oriented architecture[3] to use resources in business processes [47]. For instance, they model business processes, instances, tasks, and states as URL identifiable resources, also model control flows and state transitions as links between connected resources, and finally use micro-formats and URL templates for dynamic process coordination. For Xu et al., resources increase process visibility and interoperability, and links could represent control/data flows for possible next-step actions.

Karkkäinen et al. propose an approach for managing information on products using a peer-to-peer architecture in association with a centralized data repository [24]. Data (seen as a consumed resource) gathered from different stakeholders is processed into information on products and then, shared with the whole supply network of partners. However, building a centralized repository to integrate a wide variety of data could become costly along with using a lot of (computational) resources.

2.2. What is Internet of things?

Connecting the Web and physical objects together is not new. For instance, attaching physical tokens (e.g., bar-codes) to objects allows to redirect users to a specific page(s) that contains object-related information [45]. The pages were first available on static Web servers, then developed further to enable low-power devices to be part of wider networks, so that gateway systems can access these pages [40]. Mapping physical objects onto virtual counterparts makes these objects accessible from and controllable over the Web [17]. Främling et al. propose first, a globally unique product identifier [16] to identify product items during their lifecycles and second, a dedicated product agent [15] to manage these items. In a similar vein, managing IoT devices requires unique identification and interfacing. As stated in Section 2.1, ROA adopts objects' URIs in a bid to build a reference model that would make each resource a directly accessible distributed component via a standard uniform interface [27]. This interface facilitates interaction with physical objects, connected to resources in the Web, through four primitives - create, read, update, and delete - which are mapped onto HTTP methods: PUT to create a resource, GET to read a resource, POST to update a resource, and DELETE to delete a resource [34].

ROA principles have also been supportive of the Web-of-Things (WoT) that results from the convergence of cloud computing and IoT [20]. WoT proposes methods for accessing smart devices through existing Web-based technologies such as Web services and RESTful services. WoT offers a solution to manage and use IoT resources in a service-oriented fashion.

Mayer et al. propose AutoWoT as a toolkit for virtualizing and managing IoT devices on the Web [29]. AutoWoT defines these devices' hierarchical structure and properties and integrate them into the WoT as automatically generated RESTful services. The authors adapt hRESTS microformat of Kopecky et al. [25] to describe the RESTful services for interoperability and

[3]Micro-services could also be represented as resources.

discovery purposes. In the same vein, Mayer et al. present an open semantic framework for the Industrial IoT [30]. The objective is to support thing interaction using semantic technologies. In [34], Paganelli et al. propose a framework targeting developers who wish to integrate IoT devices into the WoT as RESTful services as well. The framework proposes tools for creating and composing these services. To represent resources, aggregation and reference relations are used. Nastic et al. encapsulate fine-grained IoT devices into software-defined APIs so that an IoT cloud-system is created [31]. The necessary software-defined APIs encapsulate IoT devices and their functionalities in the IoT cloud in order to abstract their access, configuration, provisioning, and governance in the IoT cloud systems.

Last but not least, in another initiative, the Open Group in [19] and Robert et al. in [39] present a standard communication interface, Open Messaging Interface (O-MI) [18], and a standard data format, Open Data Format (O-DF), respectively, to bridge the interoperability gap among IoT-based objects/devices and stakeholders. O-MI fulfils the same purpose of the HTTP for the Internet, but for IoT objects. Indeed, O-MI allows transporting a variety of data in almost any format (e.g., JSON, CSV, and XML) over these objects.

*2.3. Highlights of the *aaR framework*

To wrap up this section, we note as per Section 2.1 the restrictive use of resources as either a modeling concept or an operation to trigger. Contrarily, we advocate for a different role for resources by first, specializing them into computational, consumed, and produced, so that applications' needs are considered and second, allowing them to engage together in collaborative scenarios. Moreover, we note that the *aaR framework's motivations are inline with those of [5, 31, 34, 29]: connecting the Web/cloud and physical devices together. For instance, Botta et al. identify some gaps that the integration of cloud into IoT could fill out [5]. Healthcare domain has been used to illustrate the tangible benefits of such integration. Everything-as-a-resource has also been used by Hofman as a base for building seamless interoperable platforms in the world of IoT [21]. Each resource (e.g., truck and smart object) has goals and capabilities and may have an owner, user, and virtual representation.

Like [34, 29], we embrace ROA principles to virtualize and manage IoT devices on the Web through appropriate resources. This happens since the *aaR framework is flexible supporting resource specialization. Also, like [31, 34], we consider that compositions of atomic (or fine-grained) IoT devices are deemed necessary when building IoT applications and thus, composition is supported by the *aaR framework. Our main contributions are threefold: *(i)* the *aaR framework considers IoT devices as well as other resources (e.g., Web services), *(ii)* resources in the *aaR framework are categorized into computational, produced, and consumed; each exposes different capabilities that are mapped onto methods, and *(iii)* resources' methods might have restrictions that need to be satisfied prior to their use (e.g., a sensor whose data is available only at a certain time).

3. Everything as a resource framework

This section presents the *aaR framework's foundations and principles that shape the design of and define the functioning of the ecosystem and future resource-based (e.g., IoT) applications. Categorizing resources and defining their lifecycles are, also, presented in this section along with a discussion about restrictions over and descriptors[4] of resources.

3.1. Framework representation

The *aaR framework guides the architecture and management of first, the ecosystem of resources (called ecosystem perspective in the rest of the paper) and second, future resource-based applications (called application perspective in the rest of the paper). These applications could (if necessary) "adjust" the ecosystem in order to consider their structural and functional characteristics/requirements (e.g., IoT).

The framework offers providers and engineers building blocks (like repositories) to architect the ecosystem and applications, and operation modules (like discovery) for managing the ecosystem and applications. By block we mean anything that structures the ecosystem and/or an application. And, by module we mean anything that acts upon certain building blocks and/or interacts with certain modules to ensure the functioning of the ecosystem and/or an application. It is worth noting that the adjustment of the ecosystem of resources results into dedicated blocks and modules that respond to the applications' specific needs.

To represent the *aaR framework, we use three layers that permit to separate the concerns between the different blocks and modules and to confine these blocks/modules' particular duties into each layer. The layers are application, resource, and infrastructure (Fig. 1).

- The application layer is at the top of the *aaR framework acting as an interface between all resources and any potential stakeholder who will set-up the ecosystem and/or configure (i.e., adjusting if necessary) the ecosystem when she is about to develop a resource-based application. Two categories of stakeholders are identified in the framework: resource providers who are linked to the ecosystem and application engineers who are linked to (IoT) applications.
    - From an ecosystem perspective, there are not any building blocks to report in the application layer. Contrarily, there is one operation module that is *description*. This module assists both providers and engineers define resources for the ecosystem and applications, respectively.
    - From an application perspective, there is one building block that is *repository of business scenarios*. These scenarios guide the development of resource-based applications in terms of what resources are

[4]Descriptors are documents describing resources (Appendix 1).

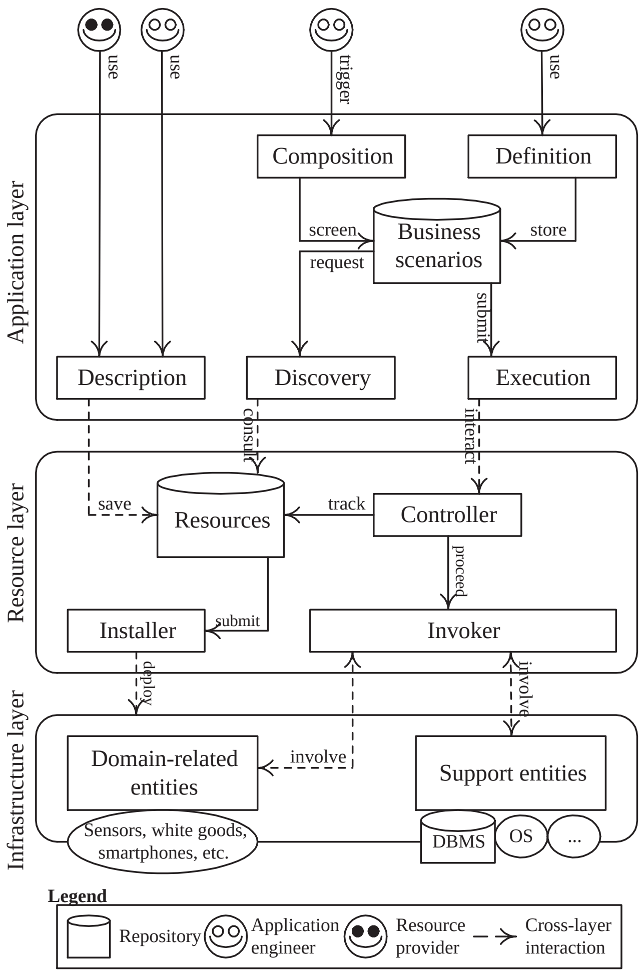


Figure 1: Representation of the *aaR framework

needed and how to assemble the necessary resources together. The operation modules include *definition*, *composition*, *discovery*, and *execution*. The *definition* module supports engineers specify business scenarios[5] that are stored in the appropriate repository. The *composition* module[6] identifies (one or many) resources according to business scenarios and with the assistance of the *discovery* module[7] that consults the resource layer's *repository of resources*. Last but not least, the *execution* module implements the business scenarios as per the request of the *composition* module and once the resources are confirmed for involvement in these scenarios. Confirmation means satisfying restrictions (Section 3.3).

- The resource layer sits between the infrastructure and application layers assisting stakeholders either in deploying resources on the infrastructure (for the benefit of providers and engineers) as per the request of the application layer's *description* module or in confirming resources for business scenarios (for the benefit of engineers) as per the request of the application layer's *discovery* module.

  – From an ecosystem perspective, there is one building block that is *repository of resources*. It contains the description of the ecosystem's and applications' resources along with their lifecycles and restrictions. There is one operation module that is *installer*. It deploys resources on the infrastructure layer by mapping them onto specific concrete *entities* in the resource layer.

  – From an application perspective, there are not any building blocks to report in this layer. Contrarily, the operation modules include *controller* and *invoker*. The *controller* module tracks a resource's lifecycle according to changes/events that happen/arise in the ecosystem and/or applications. This module also allows the *aaR framework's stakeholders (i.e., providers and engineers) to enforce restrictions over resources at run-time. The *invoker* module involves entities in the execution of business scenarios upon the approval of the *controller* module.

- The infrastructure layer is at the bottom of the *aaR framework hosting two types of entities: *support* and *domain-related*.

  – From an ecosystem perspective, *support* entities act on behalf of hardware (e.g., servers) and software (e.g., OS) components that are as-

[5]Existing techniques like BPMN could be adopted (and/or adapted) but this does not fall into the scope of this work.

[6]Existing composition techniques like BPEL [8] could be adopted (and/or adapted) but this does not fall into the scope of this work.

[7]Existing discovery techniques based on broker/matchmaker [44] and meeting infrastructure [28] could be adopted but this does not fall into the scope of this work.

sociated with the ecosystem's/applications' building blocks and operation modules.

– From an application perspective, *domain-related* entities act on behalf of specialized hardware (e.g., sensors) and software (e.g., temperature controller) components associated with future resource-based applications.

Different cross-layer interactions are represented in Fig. 1 using dashed lines and are summarized below:

1. The application layer's *description* module interacts with the resource layer's *repository of resources* in order to save resources' descriptors, lifecycles, and restrictions.
2. The application layer's *discovery* module interacts with the resource layer's *repository of resources* in order to consult the necessary resources with respect to the needs of business scenarios.
3. The application layer's *execution* module interacts with the resource layer's *controller* module to ensure that resources' restrictions are satisfied prior to using these resources during business-scenario execution.
4. The resource layer's *installer* module interacts with the infrastructure layer in order to deploy *support/domain-related* entities associated with the ecosystem's/applications' hardware and software components.
5. The resource layer's *invoker* module interacts with the infrastructure layer in order to involve appropriate hardware and software components, through their respective entities, when operating the ecosystem and/or applications.

3.2. Categories of resources

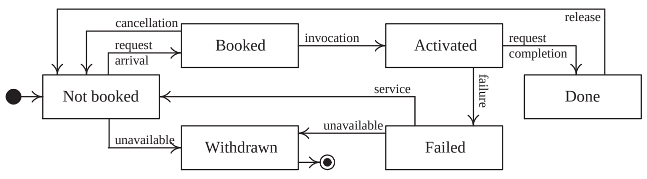
We categorize resources into computational, consumed, and produced. We define a computational resource as a software program that processes inputs and produces outputs, although both are not compulsory (e.g., a regulator receiving a room's temperature and adjusting the heater accordingly). We also define a consumed/produced resource as an input(s)/output(s) that could be linked to a computational resource at run-time (e.g., room's temperature that a sensor produces is consumed by a regulator).

For a better understanding of how resources are used and operate in both the ecosystem and the resource-based applications, we define their behaviors using statecharts (Fig. 2). Behavior is a lifecycle that indicates the permissible states that a resource takes along with the possible overlaps between resources (e.g., a produced resource could become a consumed one).

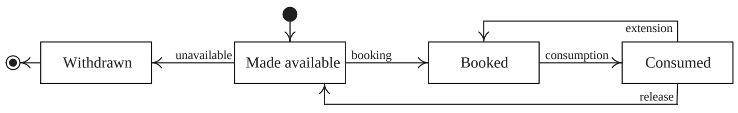
- States of a computational resource include not-booked (i.e., idle), booked (i.e., confirmed for activation), activated (i.e., under activation), done (i.e., successful activation), failed (i.e., unsuccessful activation), and withdrawn (i.e., no-longer available). Some transitions include booked $\xrightarrow{invocation}$ activated and failed $\xrightarrow{unavailable}$ withdrawn.

- States of a consumed resource include made-available (i.e., announced for possible consumption), booked (i.e., confirmed for consumption by a computational resource), consumed (i.e., under the consumption of a computational resource), and withdrawn (i.e., no-longer available). Some transitions include consumed $\xrightarrow{extension}$ booked and booked $\xrightarrow{consumption}$ consumed.

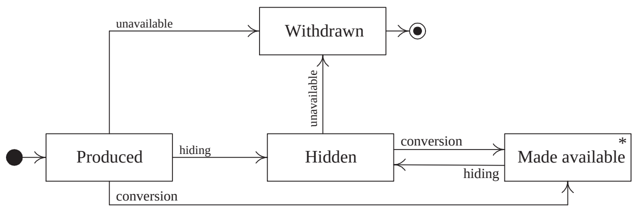
- States of a produced resource include produced (i.e., resulted from the activation of a computational resource), hidden (i.e., made unavailable temporarily), made-available[8] (i.e., announced for possible consumption), and withdrawn (i.e., no-longer available). Some transitions include produced $\xrightarrow{hiding}$ hidden and hidden $\xrightarrow{unavailable}$ withdrawn.



(a) Computational



(b) Consumed



(c) Produced

Figure 2: Resources' lifecycles represented as statecharts

We rely on transitions between states to define the necessary methods for operating resources (i.e., to ensure resource inclusion in business scenarios). In the following, we list these methods and mention the transition's name next to each method's name (Fig. 2, trs stands for transition).

[8]Asterisk in Fig. 2c indicates that a produced resource could become a consumed one.

Computational resource - we list the following methods (Fig. 2a):

- Book(trs:request_arrival) allows to schedule the invocation of a computational resource.
- Activate(trs:invocation) allows to invoke (bind) a computational resource by either another computational resource or a third-party application.
- Release(trs:release) allows to make a computational resource available for forthcoming activations.
- Cancel(trs:cancellation) allows to drop the booking of a computational resource after booking confirmation.
- Withdraw(trs:unavailable) allows to make a computational resource unavailable for invocation.
- Service(trs:service) allows to maintain and/or fix a computational resource as part of preventive and corrective strategies. Maintenance could also concern revising methods and/or restrictions in response to certain changes (e.g., new activation fees).

Consumed resource - we list the following methods (Fig. 2b):

- Offer(trs:available) allows to make a consumed resource available to computational resources.
- Consume(trs:consumption) allows a computational resource to use a consumed resource.
- Book(trs:booking) allows a computational resource to request the use of a consumed resource. This helps the consumed resource respond to other computational resources' booking requests.
- Cancel(trs:cancellation) allows a computational resource to drop the request of using a consumed resource. This helps the consumed resource respond to other computational resources' booking requests.
- Release(trs: release) allows a computational resource to make a consumed resource available after use for other computational resources. This helps the resource respond to other computational resources' booking requests.
- Extend(trs:extension) allows a computational resource to adjust the booking of a consumed resource due to some time and/or computing constraints (e.g., expiry date).
- Withdraw(trs:unavailable) allows to make a consumed resource unavailable to computational resources. This unavailability could be used for revising methods and/or restrictions in response to certain changes.

Produced resource - we list the following methods (Fig. 2c):

- Convert(trs:conversion) allows to make a produced resource available to other computational resources (asterisk in Fig. 2c). The produced resource will be treated as a consumed resource.
- Hide(trs:hidding) allows to suspend the availability of a produced resource temporality prior to making this resource totally either unavailable or available as a consumed resource.
- Withdraw(trs:unavailable) allows to make a produced resource unavailable (e.g., retire).

In the *aaR framework, resources' lifecycles are tracked so that the *controller* module enforces restrictions over these resources (i.e., only the allowed transitions according to statecharts). The definition of restrictions is presented in the next section.

3.3. Restrictions over resources' methods

A reader could raise the question of why we apply restrictions to methods of a resource and not the entire resource. We respond as follows:

1. Restrictions over methods permit a better fine-tuning of how to manage and operate a resource in a specific context. For instance, a method is disabled in a non-secure environment but enabled in a secure one. Otherwise, the entire resource is either enabled or disabled.
2. Restrictions over methods are revisited throughout the lifecycle of a resource without impacting all restrictions nor all methods. For instance, only certain restrictions are revisited when a resource is in a specific state.

We propose the following restrictions over the methods of a resource with the assumption that, unless stated, a method is by default unlimited (ul) and/or shareable (s):

- limited (l): when the calls to a method are subject to a threshold (e.g., maximum number) and/or a specific time frame (e.g., before June 2017). The main motive of limited is to ensure resource performance and/or availability. Limited could be relaxed a bit through limited-but-renewable (lr), which permits to extend the calls to a method in return of a fee, for example.

- non-shareable (ns): when the concurrent calls to a method are not allowed so coordinating these calls becomes necessary. The main motive of non-shareable is to ensure resource consistency.

The satisfaction of restrictions over methods happens as follows. Before the *execution* module calls any method of a resource, it requests the assistance of the *controller* module that proceeds with analyzing this resource's descriptor stored in the *repository of resources* (Section 3.4). If there are restrictions (needless to discuss the opposite), the *controller* module analyzes the nature of these restrictions. In the case of limited, the *controller* module evaluates whether the call respects a specific threshold or falls into a specific time frame, for example. Otherwise (i.e., non-shareable), the *controller* module checks if there are not any ongoing calls associated with the method.

Table 1 lists computational and consumed resources' methods along with restrictions. Produced resources are dropped from the table since they do not get involved in any business scenarios unless they become consumed. Because some methods (e.g., hide and service) are reserved for internal use by providers and/or engineers, only, they are free of restrictions and hence, not reported in Table 1.

Table 1: Application of restrictions to methods of resources

Resource category	Resource method	Restrictions (l,ns.lr)
Computational	activate	l:applicable
		ns:applicable
		lr:applicable
	release	l:not-applicable
		ns:not-applicable
		lr:not-applicable
	book	l:applicable
		ns:not-applicable
		lr:applicable
	cancel	l:not-applicable
		ns:not-applicable
		lr:not-applicable
Consumed	book	l:applicable
		ns:applicable
		lr:applicable
	cancel	l:not-applicable
		ns:not-applicable
		lr:not-applicable
	release	l:not-applicable
		ns:not-applicable
		lr:not-applicable
	extend	l:applicable
		ns:applicable
		lr:applicable
	consume	l:applicable
		ns:applicable
		lr:applicable

3.4. Resource descriptor

In Section 3.1, we mention that providers and application engineers describe resources. To support both, we propose a resource's descriptor model (non-UML model) and use some of XML terminologies when discussing this model. The use of these terminologies also helps in converting the model into an XML document. In Fig. 3, a resource consists of a unique identifier (e.g., URI), name, category, and two recursive relations (i.e., "consumes" and "produces") that refer to the potential connections between resources. In addition, a resource has two elements that are capacity and method. Each element consists of a name and description. The multiple descriptions offer a human-readable content that permits to ease the search of the necessary resources over the *repository of re-*

sources. In the descriptor model, some methods are associated with restrictions that consist of a name, type, and value.

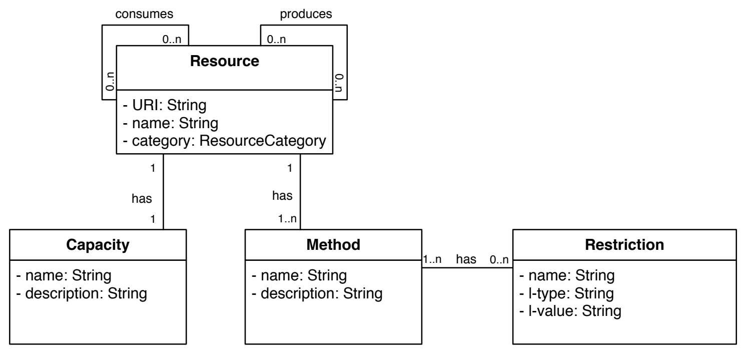


Figure 3: Descriptor model of a resource

In Fig. 3, capacity is a way of describing the role of a resource in either the ecosystem or an application. Capacity is to *process* (e.g., analyzing a blood sample) for a computational resource, to *abstract* (e.g., storing a sensor's data) for a produced resource, and to *capture* (e.g., converting a text file into an XML document) for a consumed resource. Examples of resource descriptors are given in Appendix 1.

4. Implementation

Section 1 suggests IoT as a potential application domain that would call for adjusting the *aaR framework. This adjustment means including extra building blocks and operation modules in the framework to meet IoT-specific requirements. In this part of the paper, we present an IoT-driven case-study along with its implementation and evaluation.

4.1. Case study

The market for health monitoring systems is currently over stuffed by application-specific solutions and tools that are disconnected from each other (i.e., silos) as they are made up of diverse architectures and technologies. Hence, achieving a cost-effective way that supports the cooperation/interoperability of these systems and tools is a challenge along with ensuring efficient data exchange using standards, for example [23]. For illustration, we study Blue Cross Blue Shield Association (BCBSA), a US national healthcare federation of thirty six independent and community-based companies [6]. BCBSA needs to adapt to the US rapidly changing healthcare business by creating a more dynamic and responsive system. This system will help quickly analyze the large volume of data collected/generated from patients (e.g., sensors in rooms), and make it accessible in different formats when deemed necessary. BCBSA would like to ensure that its affiliates (e.g., clinics, practitioners, physicians, and hospitals) provide safe and efficient care while keeping healthcare cost affordable to the US population, hence enabling healthier living and communities. In this regard, IoT is a good candidate for supporting

1. Develop programs, in the form of computational resources, that might process inputs and produces outputs, for allowing BCBSA system to connect to patients' data sources.
2. Allow BCBSA to use consumed resources to capture the data obtained from (*i*) internal systems' affiliates, (*ii*) patients' sensors, (*iii*) ambient sensors, and (*iv*) any other local and national agencies, where potential patients live.
3. Generate comprehensive patients' reports/files in the form of produced resources. These reports/files can be self-updated with the latest data and news collected from the aforementioned data sources, and made available in different file formats (e.g., flat files), that other affiliates will use as consumed resources for further processing.

4.2. System development

Fig. 4 illustrates the architecture of our IoTR4HealthCare system that demonstrates how the *aaR framework is used for developing a resource-based application. IoTR4HealthCare is implemented in a RESTful fashion and uses standard Web technologies (e.g., HTML5, PHP, and JavaScript) and HTTPS protocol[9]. The latter secures communications between IoTR4HealthCare and any real information-systems like those mentioned in the case study. Additional components of the system include a secure gateway that wraps sensors and a Google cloud-based platform upon which some modules of the framework are deployed. Two short videos of IoTR4HealthCare in action are available at `social.connect.rs/resources/admin.mp4` to create gateways and bind them to the system's main dashboard and `social.connect.rs/resources/doctor.mp4` to show how a doctor monitors patients.

In the application layer, engineers (i.e., resource providers) access IoTR4HealthCare through the *description* module in order to specify their resources along with possible restrictions over the methods of these resources (Table 1). This module is under-development and will be a Web application that generates resource descriptors to store in the resource layer's *repository of resources*. For the sake of demonstration, we manually define some necessary resource descriptors (Appendix 1). Still in the application layer, the *composition* module allows the medical staff (e.g., doctors and nurses) and system administrators to trigger business scenarios (e.g., HeartDiseaseAnalysis and PatientRecordBackup) stored in the *repository of business scenarios* with respect to their needs. The triggering happens through dedicated PHP-based dashboards and/or RESTful APIs. When a trigger happens, the *composition* module seeks

[9]WebSockets, MQTT, or COAP could also be used instead of HTTP protocol.

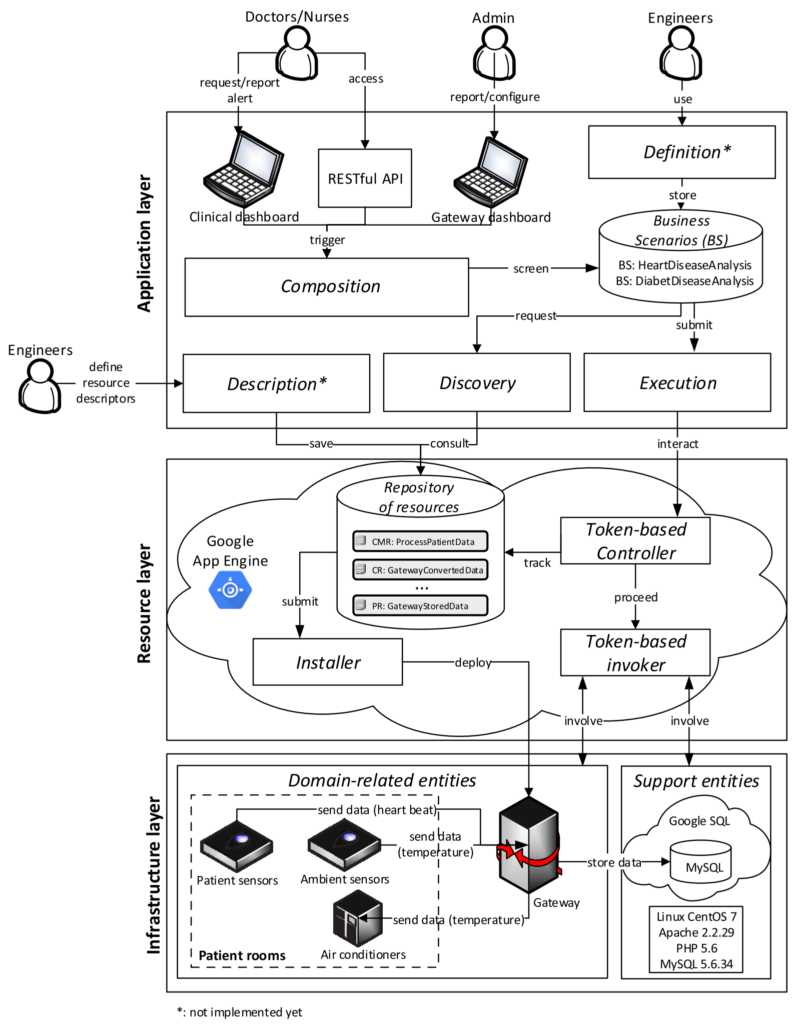


Figure 4: Deployment architecture of the IoTR4HealthCare system

the assistance of the *discovery* module, developed as PHP scripts, to look for the necessary resources. To this end, the *discovery* module uses HTTPS requests and RESTful get to consult the resource layer's *repository of resources* that is implemented as a Google Cloud SQL database. When a resource is identified, the *execution* module carries out the business scenarios under the *controller* module's supervision. For the sake of demonstration too, a business scenario is described in a simple XML-based data-oriented workflow language proposed by Sellami et al. [42] and refers in a static way to the necessary resources.

In the resource layer, a token-based strategy is adopted to implement and validate restrictions over resources' methods. We assign a token per user who will involve resources in her business scenario. To this end, we instantiate[10] the *aaR framework's *controller* and *invoker* modules into *token-based controller* (authorizes connections to the infrastructure layer's entities) and *token-based invoker* (involves the infrastructure layer's entities in business scenarios upon the *token-based controller*'s approval), respectively. In IoTR4HealthCare, the following are examples of implemented resources:

- ProcessAmbientData and ProcessPatientData as computational resources that are automatically triggered to check sensors' data prior to taking actions like adjusting the room temperature or calling a nurse.
- ReportPatientHealth as a computational resource that doctors use to produce and/or consult patients' medical reports.
- ProcessAmbientData as a computational resource that consumes GatewayConvertedAmbientData like room temperature and produces GatewayStoredPatientData like new temperature.
- ProcessPatientData as a computational resource that consumes GatewayConvertedPatientData like heart beat and produces GatewayStoredPatientData like drug levels to patient's health.

The infrastructure layer includes different IoT-related entities and support entities. In IoTR4HealthCare, some IoT-related entities include patient simulated sensors for heart beats and ambient sensors for room temperatures (Fig. 5), actuators like air conditioners that are started/triggered according to ambient temperatures, and gateway that parses collected data from sensors into JSON files. Some support entities include Linux CentOS7, Apache 2.2.29, and PHP 5.6. Acting as a proxy over the IoT-related entities, a *gateway* receives the data as consumed resources, labeled as GatewayStoredData, and generates produced resources, labeled as GatewayConvertedData. Data are stored in a Google Cloud SQL database.

[10]Instantiation is an example of adjusting the *aaR framework in response to certain needs and requirements.

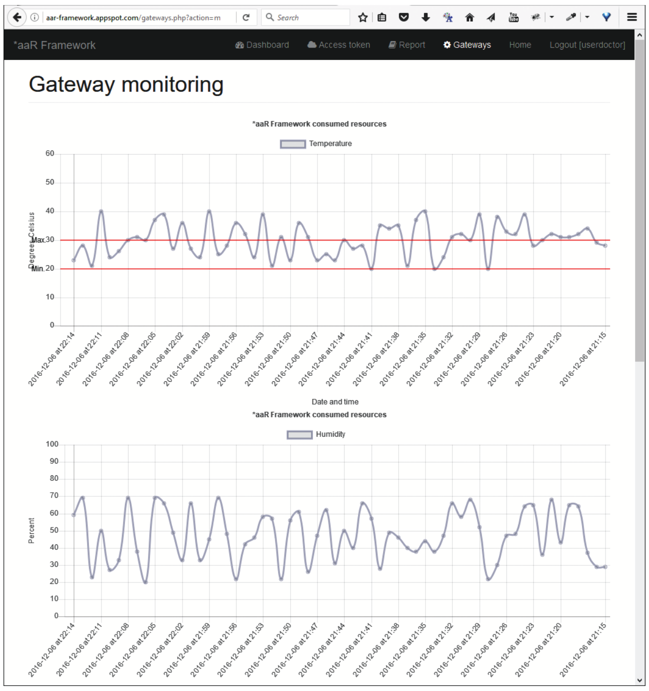


Figure 5: Collected temperature and humidity data in IoTR4HealthCare using sensors

4.3. System evaluation

To evaluate IoTR4HealthCare's *cost* and *latency* (i.e., end-to-end delay) metrics, we deemed necessary benchmarking them against two well-established solutions namely OASIS Devices Profile for Web Services-Enabled dEvices Platform (DPWS-DEEP) [7, 33] and ROA Event Manager (ROAEM) [9].

- DEEP allows the management of multiple DPWS-described IoT devices and supports the development of IoT applications by connecting these devices together via a Raspberry Pi gateway. This gateway transforms the messages/data received from sensors to DPWS-compliant services (built over a TCP/IP stack) so that devices are represented as services.
- ROAEM is deployed on top of Publish/Subscribe Applied to Distributed Resource Scheduling middleware (PADRES) [22] so that it connects event publishers and subscribers together.

For experimentation purposes, we connect IoTR4HealthCare gateway to a set of simulated sensors so that it receives their generated data as flat files (txt). These files are treated as produced resources to become consumed resources at a later stage. In the following we discuss *cost* and *latency* in the context of IoTR4HealthCare, DEEP, and ROAEM.

- *Cost*(IoTR4HealthCare,DEEP): in DEEP, the cost of read/write operations (Table 2) is based on Google resource billing rates available at `cloud.google.com/appengine/pricing`. We adopt the same rates for IoTR4HealthCare in terms of number of sensors, 300 listed patients per day, and 100 patients with connected sensors who were monitored for 15s per day.

Table 2: Cost estimation of operations performed in DEEP [7]

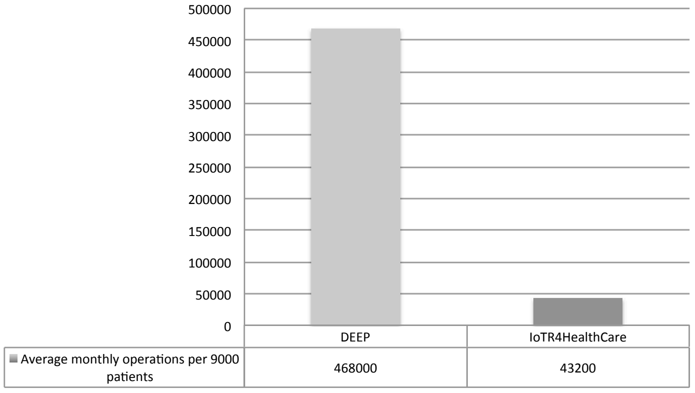
Operation	**Read**	**Write**	**Cost per operation ($)**	**Cost per each 100k ($)**
Register patient	1	2	2.50000E-06	0.25
Remove patient	0	2	1.80000E-06	0.18
List patients	1	0	1.20000E-06	0.12
List sensors	1	0	6.00000E-07	0.06
Register sensor	1	2	2.50000E-06	0.25
Remove sensor	1	2	2.40000E-06	0.24
Recover patient data	1	0	7.00000E-07	0.07
Update value of a sensor	0	1	1.90000E-06	0.19

The cost of the IoTR4HealthCare gateway operations (Table 3) indicates that it successfully surpassed DEEP in maintaining lower cost per operation across the entire simulation period. This is simply due to the way that IoTR4HealthCare gateway writes collected data sensors (serialized as a single JSON file) to the Google cloud-based MySQL database, which charges per number of write, read, and delete operations. For example, if there are 100 sensors sending data formatted into one JSON file via the gateway, then Google will charge one write operation on the database. Contrarily, DEEP sends 100 data sensors in any format separately, hence it is charged for 100 write operations.

Table 3: Cost of operations performed in IoTR4HealthCare

Operation	**Read**	**Write**	**Cost per operation ($)**	**Cost per each 100k ($)**
Register patient	0	1	1.80000E-06	0.18
Remove patient	0	0	0.20000E-06	0.02
List patients	1	0	0.60000E-06	0.06
List sensor	1	0	0.60000E-06	0.06
Register sensor	1	0	0.60000E-06	0.06
Remove sensor	1	0	0.60000E-06	0.06
Recover Patients details	1	0	0.60000E-06	0.06
Update patient's values (all sensors)	1	1	2.40000E-06	0.24

Fig. 6 illustrates the monthly average number of read/write per operation in both DEEP and IoTR4HealthCare. Thanks to 45000 monitoring operations, 9000 values have been calculated based on (100 patients × 15 seconds monitoring per day × 1 data refresh per second × 30 days). We notice that the number of read/write operations of the same number of sensors via IoTR4HealthCare gateway is exactly twenty times less than the number of operations made by DEEP Raspberry Pi gateway.



	DEEP	IoTR4HealthCare
▨ Average monthly operations per 9000 patients	468000	43200

Figure 6: Average number of monthly operations for 9000 listed patients

- *Latency* (IoTR4HealthCare,ROAEM): measuring latency depends on both the number of connected sensors and the size of the created JSON file to be sent to the Google cloud-based platform located in the resource layer. In ROAEM, experiments were performed using both real and simulated devices. For the sake of consistency, only the simulated results of

IoTR4HealthCare and ROAEM are compared. In both cases, 30 sensors were considered using cooja simulator [9].

Fig. 7 illustrates how the latency in ROAEM and IoTR4HealthCare's gateway increases in a linear way with the increase of sensors. However, the plain- and dashed-trend lines show that latency in ROAEM increased much faster than in IoTR4HealthCare. This is due to the fact IoTR4HealthCare's gateway sends one JSON file to the cloud despite of the number of connected sensors, as such the delay is completely reliant on the size of the JSON file that is influenced directly by the number of sensors. Contrarily, data in ROAEM is sent separately, per each sensor, by the event manager that in turn needs a lot more time to process the collected data individually. Latency has helped demonstrate IoTR4HealthCare scalability when the number of sensors increases, which as mentioned before, was limited to 30 sensors in this experiment to compare to ROAEM.

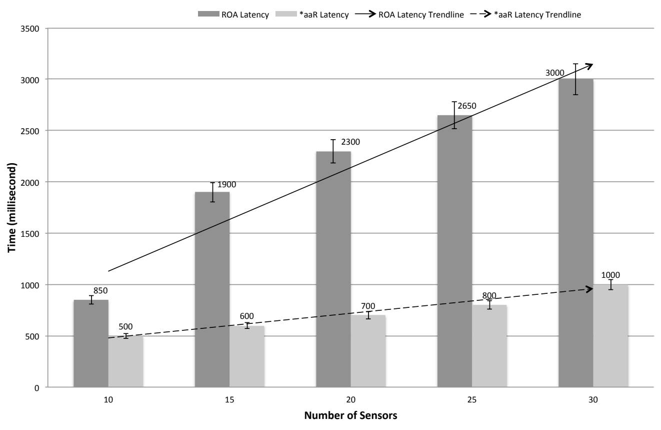


Figure 7: Latency analysis in in IoTR4HealthCare *versus* ROAEM

5. Conclusion

In this paper, we discussed Everything-as-as-Resource (*aaR) as a paradigm for designing collaborative systems on the Web. In this paradigm, resources are part of an ecosystem that features different stakeholders, are categorized into computational, consumed, and produced, are associated with trackable lifecycles that stress out their differences from an operation perspective, and finally, are customized so they accommodate the intrinsic characteristics of future resource-based applications. The connection between the three categories of resources is straightforward: a computational resource that is invoked at run-time, could consume (existing) resources and/or produce (not necessarily new) resources. Although resources have been widely used in different ICT-related domains, we analyzed resources from two perspectives, *capacity* so that computational resources process data, produced resources abstract data, and consumed resources capture data, and *restriction* so that access to resources is controlled in terms of limitedness, shareability, and renewability. We exemplified *aaR with an Internet-of-Things (IoT)-based healthcare case-study along with an online application referred to as IoTR4HealthCare that uses different technologies such as JSON and Google cloud. IoTR4HealthCare has been compared to two existing systems, DEEP and ROAEM, in terms of cost and latency. The obtained results show that IoTR4HealthCare is less costly and faster than DEEP and ROAEM, respectively. In term of future work, we will develop an automatic resource discovery mechanism, complete the implementation of certain modules like *description* and *definition*, and consider the adoption of smart fabrics and networked clothing as discussed in [14]. We will also explore the opportunities of applying *aaR to other application domains such as sport events [26].

References

[1] P. Adamczyk, P.H. Smith, R.E. Johnson, and M. Hafiz. REST and Web Services: In theory and In Practice. In E. Wilde and C. Pautasso, editors, *REST From Research to Practice*. Springer, 2011.

[2] P.M. Barnaghi and A.P. Sheth. On Searching the Internet of Things: Requirements and Challenges. *IEEE Intelligent Systems*, 31(6), 2016.

[3] P. Bellavista, A. Corradi, and C. Stefanelli. The Ubiquitous Provisioning of Internet Services to Portable Devices. *IEEE Pervasive Computing*, 1(3), July/September 2002.

[4] T. Berners-Lee, J. Handler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), May 2001.

[5] A. Botta, W. de Donato, V. Persico, and A. Pescapé. Integration of Cloud Computing and Internet of Things: a Survey. *Future Generation Computer Systems*, 56, 2016.

[6] Intel Corp. Health Insurance Association Launches a Security and Integration Cloud Service Brokerage, 2014 (visited in June 2016). http://www.intel.co.uk/content/dam/www/public/us/en/documents/case-studies/enterprise-security-intel-esg-blue-cross-brokerage-study.pdf.

[7] J. Cubo, A. Nieto, and E. Pimentel. A Cloud-based Internet of Things Platform for Ambient Assisted Living. *Sensors*, 14, 2014.

[8] F. Curbera, Y. Goland, and J. Klein et al. Business Process Execution Language for Web Services (BPEL4WS) Version 1.1, May 2003.

[9] K. Dar, A. Taherkordi, H. Baraki, E. Eliassen, and K. Geihs. A Resource-oriented Integration Architecture for the Internet of Things: A Business Process Perspective. *Pervasive and Mobile Computing*, 20, 2015.

[10] M. de Weerdt and B. Clement. Introduction to Planning in Multiagent Systems. *Multiagent Grid Systems*, 5(4), 2009.

[11] D. Dimick. As World's Population Booms, Will its Resources be Enough for Us? National Geographic, http://news.nationalgeographic.com/news/2014/09/140920-population-11billion-demographics-anthropocene, October 2014.

[12] Y. Duan, X. Sun, A. Longo, Z. Lin, and S. Wan. Sorting Terms of "aas" of Everything as a Service. *International Journal of Networked and Distributed Computing*, 4(1), January 2016.

[13] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[14] J. Foroughi, T. Mitew, P. Ogunbona, R. Raad, and F. Safaei. Smart Fabrics and Networked Clothing: Recent developments in CNT-based fibers and their continual refinement. *IEEE Consumer Electronics Magazine*, 5, October 2016.

[15] K. Främling, T. Ala-Risku, M. Kärkkäinen, and J. Holmström. Agent-based Model for Managing Composite Product Information. *Computers in Industry*, 57(1), 2006.

[16] K. Främling, M. Harrison, J. Brusey, and J. Petrow. Requirements on Unique Identifiers for Managing Product Lifecycle Information: Comparison of Alternative Approaches. *International Journal of Computer Integrated Manufacturing*, 20(7), October 2007.

[17] K. Främling, J. Holmström, T. Ala-Risku, and M. Kärkkäinen. Product Agents for Handling Information about Physical Objects. Technical Report TKO-B 153/03, Department of Computer Science and Engineering, Helsinki University of Technology, 2003.

[18] The Open Group. Open Messaging Interface (O-MI), an Open Group Internet of Things (IoT) Standard, October 2014 (visitied in August 2017, available online at http://www.opengroup.org/iot/omi/index.htm and as PDF at www.opengroup.org/bookstore/catalog/c14b.htm. US ISBN 1-937218-60-7).

[19] The Open Group. Open Data Format (O-DF), an Open Group Internet of Things (IoT) Standard, October 2014 (visitied in August 2017, available online at www.opengroup.org/iot/odf/index.htm and as PDF at www.opengroup.org/bookstore/catalog/c14a.htm. US ISBN 1-937218-59-1).

[20] D. Guinard. *A Web of Things Application Architecture - Integrating the Real-World into the Web.* PhD thesis, ETH Zurich, 2011.

[21] W. Hofman. Federated Platforms for Seamless Interoperability in the Physical Internet. In *Proceedings of the 2nd International Physical Internet Conference*, Paris, France, 2015.

[22] H.A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R.S. Kazemzadeh. *The PADRES Publish/Subscribe System.* IGI Global, 2010.

[23] S. Jubler, K. Främling, and W. Derigent. P2P Data synchronization for Product Lifecycle Management. *Computers in Industry*, 66, 2015.

[24] M. Karkkäinen, T. Ala-Risku, and K. Främling. The Product Centric Approach: a Solution to Supply Network Information Management Problems? *Computers in Industry*, 52, 2003.

[25] J. Kopecký, K. Gomadam, and T. Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *Proceedings of 2008 IEEE/WIC/ACM International Conference on Web Intelligence (WI'2008)*, Sydney, NSW, Australia, 2008.

[26] S. Kubler, J. Robert, A. Hefnawy, K. Främling, C. Cherifi, and A. Bouras. Open IoT Ecosystem for Sporting Event Management. *IEEE Access*, 5, 2017.

[27] R. Lucchim, M. Millot, and C. Elfers. Resource Oriented Architecture and REST: Assessment of Impact and Advantage on INSPIRE. JRC Scientific and Technical Reports EUR 23397 EN - 2008, JRC European Commission, 2008.

[28] Z. Maamar, E. Dorion, and C. Daigle. Toward Virtual Marketplaces for E-Commerce Support. *Communications of the ACM*, 44(12), 2001.

[29] S. Mayer, D. Guinard, and V. Trifa. Facilitating the Integration and Interaction of Real-World Services for the Web of Things. In *Proceedings of Urban Internet of Things Towards Programmable Real-time Cities (UrbanIOT'2010)*, Tokyo, Japan, 2010.

[30] S. Mayer, J. Hodges, D. Yu, M. Kritzler, and F. Michahelles. An Open Semantic Framework for the Industrial Internet of Things. *IEEE Intelligent Systems*, 32(1), 2017.

[31] S. Nastic, S. Sehic, D.H. Le, H.L. Truong, and S. Dustdar. Provisioning Software-defined IoT Cloud Systems. In *Proceeding of the 2014 International Conference on Future Internet of Things and Cloud (FICLOUD'2014)*, Barcelona, Spain, 2014.

[32] R. Nick, W.M.P. van der Aalst, Arthur H.M. ter Hofstede, and E. David. Workflow Resource Patterns: Identification, Representation, and Tool Support. In *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005.

[33] OASIS. Devices Profile for Web Services (DPWS).

[34] F. Paganelli, S. Turchi, and D. Giuli. A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City. *IEEE Systems Journal*, 10, December 2014.

[35] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[36] C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web (WWW'2008)*, Beijing, China, 2008.

[37] M. Peter and G. Timothy. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011.

[38] L. Richardson and S. Ruby. *RESTful Web Services.* O'Reilly Media, 2007.

[39] J. Robert, S. Kubler, Y. Le Traon, and K. Främling. O-MI/O-DF Standards as Interoperability Enablers for Industrial Internet: a Performance Analysis. In *Proceedings of the 42nd Annual Conference of IEEE Industrial Electronics Society (IECON'2016)*, Florence, Italy, 2016.

[40] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A Service Gateway for Networked Sensor Systems. *IEEE Pervasive Computing*, 3(1), 2004.

[41] G. Schreiber and Y. Raimond. RDF 1.1 primer. Technical report, 2014.

[42] M. Sellami, P. De Vettor, M. Mrissa, D. Benslimane, and B. Defude. DMaaS: Syntactic, Structural, and Semantic Mediation for Service Composition. *International Journal of Autonomous and Adaptive Communications Systems*, 9(3/4), January 2016.

[43] C.J. Su. Web-Oriented Architecture (WOA) Enabled Customer-Centric Collaborative Commerce Platform (WCCP). *International Journal of Computer Theory and Engineering*, 7(5), October 2015.

[44] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5, 2002.

[45] R. Want, K.P. Fishkin, A. Gujar, and B.L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of the ACM SIGCHI 99 Conference on Human Factors in Computing Systems (CHI'1999)*, Pittsburgh, Pennsylvania, USA, 1999.

[46] M. Weiser. The Computer for the 21^{st} Century. *Newsletter ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3), 1999.

[47] X. Xu, L. Zhu, Y. Liu, and M. Staples. Resource-Oriented Architecture for Business Processes. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'2008)*, Beijing, China, 2008.

Appendix 1

Descriptors of ProcessAmbientData (CMR1) and GatewayStoredPatient-Data (PR4) according to the *aaR descriptor model (Section 3.4) are provided in Listing 1 and Listing 2, respectively. For the sake of readability, a simplistic human readable syntax (YAML[11]) is used for the descriptors. However, JSON

[11] YAML Ain't Markup Language, `yaml.org`.

is used for the descriptor representation in the system. Lines 3-4 of Listing 1 describe a computational resource known as CMR1:ProcessAmbientData. This resource's role is given in lines 5-8 while its produced resource PR4 is detailed in lines 9-12. The last element of the descriptor that is method (lines 13-29) represents CMR1' methods. Lines 18-21 show how limited restriction (l) is applied to the activate method so that the call to this method is restricted to authorized users, only.

Listing 1: Descriptor of the computational resource CMR1

```
1  ---
2  URI: https://social.connect.rs/resources/processAmbientData
3  name: process ambient data
4  category: computational
5  Capacity:
6    name: storeRoomTemperature
7    description: "Stores␣sensor's␣data"
8    "Data␣is␣stored␣as␣gateway␣stored␣patient␣data␣(produced␣resource)."
9  produces:
10   URI: https://social.connect.rs/resources/gatewayStoredPatientData
11   name: gateway stored patient data
12   category: produced
13 method:
14 – name: book
15     description: "schedule␣the␣invocation"
16 – name: activate
17     description: "bind␣the␣resource"
18     restriction:
19       - name: l
20         l–type: AuthorizedUsers
21         l–value: [u1, u3, u10]
22 – name: release
23     description: "make␣the␣resource␣available"
24 – name: cancel
25     description: "Drop␣the␣booking␣of␣the␣resource"
26 – name: withdraw
27     description: "Make␣the␣resource␣unavailable"
28 – name: service
29     description: "Maintain␣and/or␣fix␣the␣failed␣resource"
30 ---
```

Line 4 of Listing 2 shows the descriptor of a produced resource known as PR4:GatewayStoredPatientData. As per Listing 1:lines 9-12, CMR1 produces PR4 that can be consumed as well with respect to its capacity element (lines 5-8). PR4's methods are described in lines 9-15.

Listing 2: Descriptor of the produced resource PR4

```
1  ---
2  URI: https://social.connect.rs/resources/gatewayStoredPatientData
3  name: gateway stored patient data
4  category: produced
5  Capacity:
6  – name: patient data
7      description: "Patient's␣collected␣data␣to␣be␣consumed␣by␣another"
8      "computational␣resource"
```

```
Method:
- name: hide
    description: "hide resource for internal use"
- name: convert
    description: "make resource available as a consumed resource"
- name: destroy
    description: "destroy resource"
---
```