Semantic Technologies for Realising Decentralised Applications for the Web of Things

Felix Leif Keppmann
AIFB
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: felix.leif.keppmann@kit.edu

Maria Maleshkova
AIFB
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: maria.maleshkova@kit.edu

Andreas Harth
AIFB
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: andreas.harth@kit.edu

***Abstract*—The vision of the Internet of Things (IoT) promises the capability of connecting billions of devices, resources and things together. In the realisation of this vision, we are currently neglecting the interoperability between devices that is caused by a heterogeneous landscape of things and which leads to the proliferation of isolated islands of custom IoT solutions. A first step towards enabling some interoperability is to connect things to the Web and to use the Web stack, thereby conceiving the so-called Web of Things (WoT). However, even when a homogeneous access is reached through Web protocols, a common understanding is still missing. In addition, decentralised applications, advocated by the IoT vision, and a-priori unknown requirements of specific integration scenarios demand new concepts for the adaptation of things at runtime. Our work focuses on two main aspects: overcoming not only data but also device and interface heterogeneity, and enabling adaptable and scalable decentralised WoT applications. To this end we present an approach for realising decentralised WoT applications based on three main building blocks: 1) semantics of the devices' capabilities and interfaces, 2) rules to enable embedding controller logic within device's interfaces for supporting a decentralised applications, and 3) support for reconfiguring the controller logic at runtime for customising and adapting the application. We show how our approach can be applied by introducing a reference architecture, provide a thorough evaluation in terms of a proof-of-concept implementation of an example use case, and performance tests.**

***Index Terms*—Smart Components, decentralised applications, Web of Things, REST, Linked Data**

I. Introduction

The vision of the Internet of Things (IoT) promises the capability of connecting billions of devices, resources, and things together in the Internet. Still, what we are currently witnessing is the proliferation of isolated islands of custom IoT solutions, which support a restricted set of protocols and devices and cannot be easily integrated or extended. Furthermore, while IoT advocates decentralised applications, where devices communicate and offer an added value without a decentralised controlling unit, in reality the implementation is done via a centralised execution or a registry. A first step towards enabling some interoperability in the IoT is to connect things to the Web and to use the Web stack, thereby conceiving the so-called Web of Things (WoT). However, even when a homogeneous access is reached through Web protocols, a common understanding is still missing – specifically in terms of heterogeneous devices, different programmable interfaces, and diverse data. Semantic technologies [1] can be used to describe dataflows on a meta level, capturing the meaning of devices' inputs and outputs, and thus abstracting away from the syntactic structure. However, having the semantics of the data is not enough. While we can describe the exchanged data, the resulting applications are limited to a specific domain, and the heterogeneous device integration is still lacking.

Our work focuses on two main challenges: overcoming not only data but also device and interface heterogeneity, and enabling adaptable and scalable decentralised WoT applications. In terms of handling the plenitude of existing devices we advocate an approach based on providing a unified view on devices and describing them in terms of their programmable interfaces, since this is how their integration as part of applications is realised. In terms of realising decentralised WoT applications, naturally, we want to omit a centralised controlling unit. Furthermore, it is important to be able to adapt devices to the requirements of specific integration scenarios, at deployment time, but more importantly at runtime.

In this context we make the following contributions. First, we present an approach for realising decentralised WoT solutions based on three main building blocks: 1) the semantics of the devices' capabilities and interfaces, 2) rules to enable embedding controller logic within the device's interfaces for supporting a decentralised solution, and 3) support for reconfiguring the controller logic at runtime, for customising and adapting the application. Second, we show how our approach can be applied by introducing a reference architecture, based on Web and Semantic Web paradigms and technologies. Third, we back up the architectural design by a specific framework implementation. Next, we exemplify our work based on a use case from the domain of factories of the future. Finally we provide a thorough evaluation in terms of a proof-of-concept implementation, evaluation of an example use case, and performance tests.

The remainder of this paper is structured as follows. In Section II we introduce our motivation scenario, illustrating the challenges that we are targeting. Section III describes the design requirements for building WoT systems and the preliminaries that we build upon. In Section IV we present our approach, provide an architecture to realise this approach, and describe our implementation. For evaluation, in Section V we demonstrate the adaptability of our system at runtime, integrate an evaluation scenario, and measure the performance in terms of overhead. We describe related work in Section VI and conclude in Section VII.

II. Motivation Scenario

We motivate the challenges that we want to address with a specific use case scenario, which we use as a running example throughout the paper. Current technology developments influence not only our day-to-day activities but also businesses and the way products and services are developed and produced. In this context, we look at a typical factories of the future situation. In the manufacturing areas of factories, the safety of humans is an ongoing effort. In particular, the unintentional intrusion of humans into the operational areas of machines or robots increases the risk of injury. One way to tackle this problem is by tracking human bodies and movements, and matching them against floor plans and safety areas in order to automatically trigger warning alarms or an emergency shutdown.

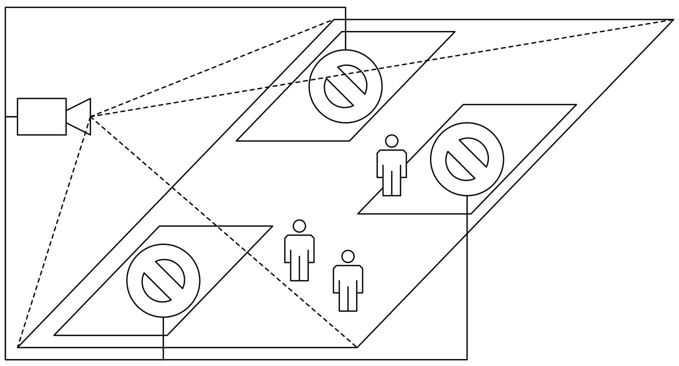


Fig. 1. Scenario: monitoring of a factory floor

Figure 1 shows a simple overview of the monitoring setup, which involves different devices and services. On the one hand, the setup includes a tracking service, with depth video cameras and body tracking algorithms, that tracks bodies, and identifies the joints and the skeleton structure. On the other hand, the setup comprises production machines, alarms, mobile devices, and sensors that provide data to the tracking service and enable appropriate reactions. This scenario is implemented as a distributed application with the added value functionality of using warnings or emergency shutdowns to save humans from injuries. To reach this goal, the involved devices (i.e., machines, robots, alarms, or mobiles) need to exchange data to provide information about the current human safety state (i.e., if a human joint is within a warning zone or not). Furthermore, determining when a human joint intrudes a danger zones is realised by matching the joint coordinates, calculated by the tracking algorithms, against the floor plan and safety areas. Both, the matching of coordinates as well as informing devices about required actions are computational elements, e.g., application logic, that need to be provided by the distributed application.

The specific challenges that we want to highlight are threefold: 1) different type of devices, with diverse interfaces, and producing and consuming a variety of data forms and structures need to be integrated; 2) while we have components, such as the tracking devices, that need to be coordinated as part of providing the application's functionality, conforming to the WoT vision, we want to avoid having a centralised controller and advocate a distributed solution; 3) the positions of the machines or the safety areas may change over time, however, having to redesign and redeploy the application every time the setup changes is inefficient. Therefore, a flexible solution is required, where the tracking service and devices can be developed at design time and additional adaptation to specific requirements can be integrated later, i.e., during deployment, or runtime.

In the following sections we use this scenario to motivate some of the design requirements for our system and to provide an exemplary implementation as well as a use case-based evaluation.

III. Design Requirements and Preliminaries

Requirements Engineering (RE) [2] aims to determine, model and specify the required and desired properties of software systems. However, what we are currently witnessing in the context of developing WoT systems is the diversity of domain-specific and use case-specific systems that are not so much concerned with thorough requirements analysis but are rather focused on quickly providing the functionality that is needed. We argue that the reasons for this are twofold. First, RE is only now starting to develop the means to support WoT system development, beginning with first steps, for example, in the representation of context. Second, traditional RE builds on the assumption that the knowledge, which is used to formulate the requirements exists a-priori and can be captured and specified. However, for WoT systems this assumption quite frequently does not hold.

Since it is based on the IoT, the WoT shares many characteristics with Wireless Sensor and Actor Networks (WSANs) and Cyber Physical Systems (CPS), which involve the connection of real world objects into networked information systems including the Web [3]. Therefore, we approach the requirements analysis by exploring how the requirements for CPS propagate to also define the WoT systems. In particular, the top requirements for building cyber-physical systems (CPS requirements (CRQ)) are as follows [4]:

WoT frameworks for CPS systems can be developed to augment the IoT and thus deal with issues such as information-centric protocols, deterministic QoS, context-awareness, etc. In this way some of the requirements listed above can already be addressed. Still, quite a few of the listed points remain relevant on the WoT level or translate to new ones. In the following we have used the CPS requirements and the motivation scenario in order to define requirements for realising WoT applications (WoT Requirements (WRQ)) based on semantic technologies.

In our work we focus on the first five requirements (WRQ1-WRQ5) and group these into three main objectives that we

CRQ1 Compositionality:
Defining components and composing them
CRQ2 Distributed Sensing, Computation and Control:
No centralised sensing, computation or control
CRQ3 Physical Interfaces and Integration:
Realising contact with the physical world
CRQ4 Human Interfaces and Integration:
Need to interface the CPS with human influence and perception
CRQ5 Information:
From Data to Knowledge: capturing raw−data−to−trusted−knowledge dependency
CRQ6 Privacy, Trust, Security:
Privacy, trust and security requirements for systems based on the physical layer
CRQ7 Modelling and Analysis− Heterogeneity, Scales, Views:
Dealing with heterogeneity in terms of creating scales and views over data
CRQ8 Software:
Traditional programming languages and structures are not really suitable
CRQ9 Robustness, Adaptation, Reconfiguration:
Dealing with dynamic environments
CRQ10 Societal Impact:
Need for social acceptance of the new systems
CRQ11 Verification, Testing and Certification:
Approaches for ensuring correctness

Listing 1. Requirements for building Cyber Physical Systems

WRQ1 Provide Device Abstraction in Terms of Components:
Overcoming devices heterogeneity by defining a converging abstraction over devices in terms of components (CRQ1)
WRQ2 Support Uniform Interfaces and Integration:
Defining compatible uniform interfaces for devices, which support the creation of composite applications (CRQ3)
WRQ3 Knowledge Representation:
Overcoming data heterogeneity via semantic representation (CRQ5, CRQ7)
WRQ4 Support Distributed Computation and Control:
No centralised computation or control, application logic is distributed among participating components, without having a centralised controller (CRQ2)
WRQ5 Enable Robustness, Adaptation, Reconfiguration:
Supporting adaptability and reconfiguration not only at design time but also at deployment and runtime (CRQ9)
WRQ6 Provide Human Interfaces and Interaction:
Need to interface with human influence (CRQ4)
WRQ7 Ensure Privacy, Trust, Security:
Privacy, trust and security specific for WoT systems (CRQ6)
WRQ8 Provide Adequate Software:
Programming languages and structures suitable for WoT (CRQ8)
WRQ9 Consider Societal Impact:
Need for social acceptance of the new systems (CRQ10)
WRQ10 Support Verification, Testing and Certification:
Approaches for ensuring correctness (CRQ11)

Listing 2. Requirements for building WoT systems

want to address: 1) Overcoming heterogeneity of data, devices and interfaces; 2) Realising decentralised applications in terms of computation and control; 3) Supporting adaptability at deployment and runtime.

We reuse existing Web and Semantic Web paradigms and technologies to cope with the requirements, and introduce these briefly as part of preliminaries.

Addressing the heterogeneity of data, devices and interfaces. In relation to handling the heterogeneity of existing devices, we advocate an approach based on providing a unified view on devices and describing them in terms of their programmable interfaces, since this is how their integration as part of applications is realised. In particular, we focus on Web Application Programming Interfaces (Web APIs) conforming to the Representational State Transfer (REST) [5] architectural style and data modelling according to the Linked Data (LD) [6] principles. This is not a novel approach but is rather adapted from the area of Semantic Web Services and Application Programming Interfaces (APIs) [7]. REST introduces flexible and loosely coupled integration of systems, relying on Uniform Resource Identifiers (URIs) for the identification of resources in combination with Hypertext Transfer Protocol (HTTP) for data transport, accessing and modification of resources. The Linked Data principles introduce with the Resource Description Framework (RDF) [8] a formal graph-based knowledge model, semantic annotation of data, as well as interlinking of datasets, again based on URIs. The combination of REST and Linked Data principles has been recently standardised as a first version of the Linked Data Platform (LDP) [9].

Addressing the need for decentralised applications. The vision of Smart Web Services (SmartWS) [10] introduces a new type of Web Services (WS) that not only provides remote access to resources and functionalities, by relying on standard communication protocols, but also encapsulates "intelligence". Smartness features can include, for instance, context-based adaptation, cognition, inference and rules that implement autonomous decision logic in order to realize services that automatically perform tasks on behalf of the users, without requiring their explicit involvement. SmartWS conform to the natural evolution of WS, starting with a number of heterogeneous protocols, moving on the standardisation in terms of the WS-* stack and adding task automation trough semantics and Semantic Web Services [11]. In this context, we address the need for decentralised applications by advocating the SmartWS approach, where component interfaces are enhanced with simple logic in order to perform some autonomous decisions.

Addressing the need for adaptability at deployment and runtime. Currently we are faced with integration scenarios, where applications must be adapted to meet certain needs that are hardly known during the design time. We apply two approaches to realise adaptability at runtime and design time. First, the simple logic within the SmartWS is implemented in terms of rules, where we take advantage of Notation3 (N3) [12] as an assertion and logic language for RDF. We provide mechanisms to overwrite these rules, to be able to reconfigure the components. Second, we describe what parts of the resources of a component are exposed via the interface by using SPARQL Protocol and RDF Query Language (SPARQL) [13] CONSTRUCT queries. These queries can also be overwritten, thus providing options to reconfigure the interface. The overwriting of rules and SPARQL CONSTRUCT queries can be done while the application is being set up or while it is already running.

IV. Realising Decentralised WoT Applications

The fulfilment of the vision of the WoT requires to extend the current Web with support that enables real-world objects to seamlessly become a part of it. Our goal is not to impose the one architecture for realising WoT application. Instead we aim to provide a set of constraints and principles backed up by specific building blocks for creating WoT applications, which address the requirements described in Section III. The main contribution of our work is to take the next logical step beyond having only data semantics or only interconnected devices in order to achieve a Web where real world objects can become first class citizens.

Our approach for developing WoT applications is based on introducing an abstraction for the convergence of all participating devices, data sources, algorithms, implemented functionalities, etc., in terms of components that provide resources and are accessible via uniform interfaces. These components can be composed into a WoT application. To provide support for a decentralised solution, without centralised computation or control, we introduce an interpretation layer between the data and functionality of components, and the APIs, which they expose to the network. This layer enables, on the one hand, adaptability of the interfaces to the requirements of specific integration scenarios and, on the other hand, deployment of intelligence (formalised as rules) on behalf of distributed applications. This intelligence ranges from simple calculations to custom behaviour or decisions, which can be reconfigured at both design time and runtime. As a result, our component-based approach enables a flexible way to compose larger distributed applications.

A. A Smart Components-based Architecture for Things

We introduce the abstraction of ***Components***, which encapsulate certain data or functionality, and can be composed by integrating these data and functionality into distributed applications to achieve added values. What components encapsulate can range from pure data sets (e.g., the floor plan with safety zones in our scenario) to devices, which dynamically produce data (e.g., our tracking service), to systems that react to state changes (e.g., machines, which shut down if their safety zones are violated).

We can build components that – ***passively*** – provide an API, with which other components interact, or we can build components, which – ***actively*** – interact with APIs of other components, or we can build components which include both. The differentiation between active and passive is based on whether the component triggers the communication or not. This distinction is necessary because in the context of WoT, the classical client and server roles are becoming inapplicable. A mobile device can have a client role by displaying information (e.g., a map or the current temperature) and at the same time act as a server by providing the current geolocation, all in one scenario. Therefore, we do not distinguish between clients and servers, but whether a component actively triggers the communication with other components or not.

We introduce the notion of ***Smart Components***[1], when they are built following our architectural approach: 1) REST for realising interfaces and the communication between components, 2) semantics for describing the exchanged data, interface resources, and components' capabilities, and 3) decentralised smartness of each component, described in terms of rules.

[1]Hereafter, when we talk about components, we mean smart components.

The use of "smart" as a way to characterise certain features is currently very common and a bit overused. However, it captures very well the properties of the components that we want to highlight, namely the encapsulation of autonomous logic and the adaptability.

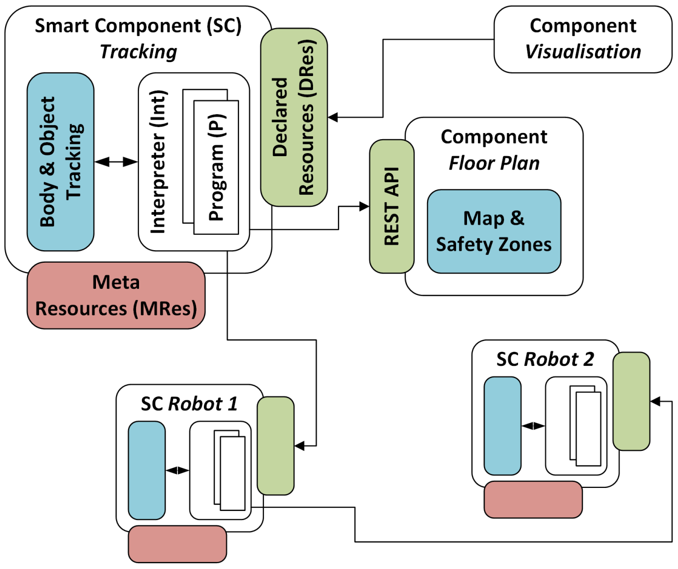


Fig. 2. Monitoring with Smart Components. The is scenario realised with five components: three smart components (tracking and two robots), one component accessible over a REST API (floor plan), and one component, which only requests and displays data (visualisation). No centralised controller.

We take advantage of the resource-oriented viewpoint within the architectural paradigm REST. Resources expose relevant parts of a component's state to the network, identified by URIs and accessible as well as modifiable through HTTP. Remote components may interact with the resources of a component, to react to the local component' state or to transition the component into a new state. The HTTP communication with resources is stateless and the resources are predefined or may be created as sub-resources of container[2] resources. In particular, some resources are defined as SPARQL CONSTRUCT queries, which can be modified (at design or runtime) to redefine, which states are exposed by the component. We do not make the assumption that all participating components need to be smart or that all devices must offer REST interfaces directly, already provided by each individual thing. To the contrary, for certain use cases, it makes more sense to take particular implementations, including highly specialised protocols, as they are and to encapsulate them to expose their resources through a REST API. In this way we enable the overall integration, while the interactions behind the encapsulating interface remain invisible.

Components have a set of ***Resources***, which provide state representation adhering to the RDF model, and can be available only internally or be exposed as part of the interface.

[2]Container resources conceptually contain a set of sub-resources and follow a defined behaviour for accessing and modifying this set, as, for example, specified by the LDP specification.

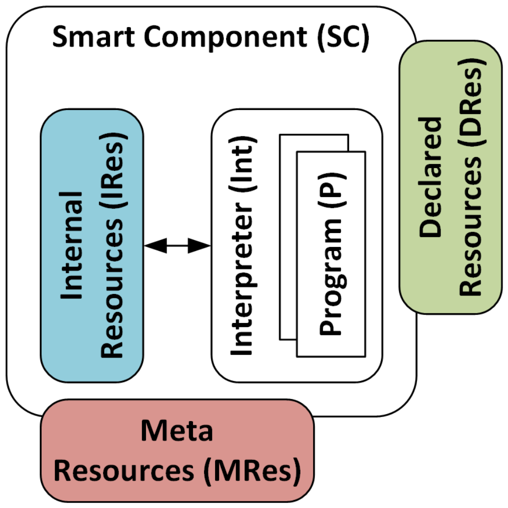


Fig. 3. Smart Component

Therefore, we distinguish between: 1) internal resources, 2) declared resources, and 3) meta resources. ***Internal Resources (IRes)*** encapsulate all data and functionality custom to a component and represent its state. By accessing these resources we can get the component's state or can cause the system to react by changing its state through resource modification. However, these resources are only accessible internally and are – following our black box approach – not identifiable or accessible in a way, which is prescribed by the architecture. ***Declared Resources (DRes)*** form the public API of a component. These resources adhere to both architectural paradigms, Linked Data and REST, i.e., they are identified by URIs, accessible by HTTP with interaction restricted to HTTP verbs, and represented according to RDF. We declare theses resources as graph patterns, defining which part of the internal resources should be exposed over the public interface. The declared resources can be accessed or modified by the other components. ***Meta Resources (MRes)*** provide the means to reconfigure and adapt a component. By interacting with these resources, we are able to access and modify both the definition of declared resources as well as the rules defining the decentralised logic of the component.

Each component encapsulates an ***Interpreter (Int)***. It provides the capabilities to 1) read and modify the state of internal resources, 2) read and modify the state of resources of other components, 3) evaluate graph patterns of declared resources, and 4) interpret decentralised logic written in a declarative rule language. In other words, the purpose of the interpreter is to negotiate between the private API of a component represented by internal resources, the public API of a component, represented by declared resources, and the interaction with resources provided by other components.

We specify the decentralised logic within a component as a set of rules that we refer to as ***Programs (P)***. The declarative rule language, in which these programs are defined, provides adequate means to express RDF graph transformations, inferencing, and interaction with resources of other components. Optionally, the language may support, in coordination with the interpreter, further capabilities to ease the declaration of programs, e.g., built-in mathematical functions for calculations, which exceed pure rule-based logic.

We call the actual interpretation of a program and evaluation of triple patterns a program run. During a run, the interpreter maintains an internal RDF graph with all states of known resources. This RDF graph is 1) enriched by the states of internal resources, 2) by requested states of declared resources, and 3) by inferred knowledge. The integration of the interpreter and the internal resources, i.e., the encapsulated data and functionality, is seen as part of the black box within the component.

Designed in such a way, smart components introduce a lot of flexibility by adding only little overhead (see Section V). There are three different phases in a component's life cycle: design, deployment, and runtime. With a generic set of meta resources, we introduce the only pre-designed part of the component's interface. It enables us to declare at runtime custom behaviour (e.g., decisions, or transformations) and use case-specific declared resources. Therefore, quite general components can be developed at design time, and be subsequently configured and adapted to fit specific scenarios during runtime.

B. Implementation

In this section we describe how we practically implement individual smart components as well as complete scenarios. First we provide details on the interpreter and rule-based programs, followed by description of the resources and interfaces.

The ***Interpreter*** is implemented by Linked Data-Fu (LD-Fu)[3] [14], [15], which offers a generic approach for the composition and integration of Linked Data REST resources. The ***Programs*** are realised with the LD-Fu declarative rule language, which is based on N3, and provides, for example, deduction of knowledge, by supporting reasoning over ontologies, interaction with external resources, by supporting HTTP GET, PUT, POST, and DELETE methods, and mathematical calculations, by supporting built-in functions. In addition, evaluation of conjunctive queries, i.e., triple patterns, over RDF graphs is integrated. The LD-Fu interpreter handles the evaluation of rules, the evaluation of queries, as well as the execution of HTTP-based interaction with external resources. Internally, the interpreter maintains a RDF graph consisting of all RDF triples that are added during a run, e.g., by adding the payload returned HTTP GET requests, or inferencing, and evaluates the programs and queries against this graph until a fixpoint is reached. Different integration scenarios with LD-Fu as central controller have been explored, e.g., the integration of static and dynamic Linked Data [16], or high-performance Linked Data processing for virtual reality environments [17]. We reuse and extend the implementation by enabling dynamic adaptation of declarative rule programs and queries at runtime, i.e., enable the access to and modification of interpreter instances via meta resources.

[3]See http://linked-data-fu.github.io for details.

TABLE I
INTERFACE FORMED BY META RESOURCES

Resource	Content Type	Methods
Instance Container	RDF	GET/POST
↪ Instance (1-n)	RDF	GET/PUT/DELETE
↪ Program Container	RDF	GET
	N3	POST
↪ Program (1-n)	N3	GET/PUT/DELETE
↪ Resource Container	RDF	GET
	SPARQL Query	POST
↪ Resource (1-n)	SPARQL Query	GET/PUT/DELETE

To support the adaptation of components at runtime we use the ***Meta Resources***, which form a (meta)-interface that provides access to the interpreter's state and, thereby, a generic way to influence the component's behaviour. In Table I, we show an overview of our current meta interface implementation, which provides access to 1) interpreter instances, 2) rule programs, and 3) declared resources. We support separate instances of the interpreter to allow independent integration in more than one distributed application, i.e., we support application-specific combinations of interpreter configurations, programs, and resources, which run independently of each other. Furthermore, we can specify an arbitrary number of programs, with the content type of N3, and declared resources, with the content type of SPARQL query. All this changes can be made at runtime via the meta resources.

TABLE II
INTERFACE FORMED BY DECLARED RESOURCES

Resource	Content Type	Methods
Instance	RDF	POST
Instance	SPARQL Query	POST
Resource	RDF	GET

Once created via interaction with the meta interface, we can interact with the ***Declared Resources***, as shown in Table II, having the content type of a supported RDF serialisation, and retrieve representations of the declared resources (i.e., the evaluation result of the underlying SPARQL construct query). In addition, we support data handling processes for resources of instances through HTTP POST requests.

We support different ways for the integration of the LD-Fu interpreter with the ***Internal Resources*** provided by components: first, as HTTP-based standalone wrapper, where the interpreter interacts with already provided Linked Data REST resources; second, via command line-based integration with support for files and pipes; third, via code-based programmatic integration by utilising LD-Fu libraries.

In summary our implementation covers: 1) the interpreter (LD-Fu), 2) programs (LD-Fu N3 rule programs), which are interpreted by the interpreter during program runs (LD-Fu interpreter runs), 3) declared resources (SPARQL CONSTRUCT queries), and 4) meta resources, which enable adaptation at runtime (LD-Fu API). Currently, we support time-based (frequency) interpreter runs, except for data handling processes.

For the purpose of demonstration and in order to support our evaluation, we built a smart component that encapsulates body tracking capabilities. We developed Natural Interaction via REST (NIREST)[4] to encapsulate depth camera support and body tracking algorithms on top of the depth video image, as well as RDF resources for the relevant data, i.e., for skeleton information of tracked bodies. We integrated NIREST and the LD-Fu interpreter at code level, and use these as part of the evaluation given in the following section.

V. EVALUATION

We provide a thorough evaluation of our approach and implementation in terms of: 1) showing the conformity to the design requirements derived in Section III, 2) building up a simplified evaluation scenario based on our motivation scenario with an explanation of provided internal resources, 2a) evaluating the deployment of decentralised controlling logic at runtime by initialisation of interpreter instances and deployment of calculation programs, 2b) evaluating the deployment of passively provided declared resources and actively executed interaction between components at runtime, 3) showing the re-adaptation for the distributed application at runtime, by switching the interaction direction as well as involving a new component, and 4) providing performance measures my determining the overhead, in terms of delay in milliseconds, resulting from using smart components.

A. *Conformity to the Design Requirements*

Our approach and implementation are based on components, which are an abstraction that we introduce in order to overcome devices heterogeneity (WRQ1). Furthermore, we support uniform interfaces and integration by relying on REST and Linked Data principles (WRQ2). Similarly, we use semantic representations for the interfaces and data exchange to overcome data heterogeneity (WRQ3). The approach of having distributed logic, within each component, supports distributed computation and control (WRQ3). Finally, we enable the key features of robustness, adaptation, reconfiguration by providing interfaces and internal logic that can be reconfigured both at design and runtime (WRQ5). Even tough not in the focus of our work, we also provide adequate software, in terms of, for example, the LD-Fu interpreter and declarative rule language (WRQ8).

B. *Evaluation Scenario and Internal Resources*

Due to space constraints, we simplify our motivation scenario for the evaluation. We consider two smart components: component C1 (Sensor), which includes a depth video sensor and provides body tracking functionality, and component C2

[4]See http://github.com/fekepp/nirest for details.

Listing 3. Internal Resources

```
<nirest://user/0>
    nirest:skeleton [
        nirest:jointPoint [
            nirest:coordinate [
                nirest:x "459.8463"^^xsd:float ;
                nirest:y "404.0497"^^xsd:float ;
                nirest:z "2037.2391"^^xsd:float ;
                a nirest:Coordinate ] ;
            a nirest:RightHandJointPoint ] ;
        ...
```

(Robot), which represents a machine or robot. In the distributed application, which we compose first, the robot component informs itself and reacts on distance alarms provided by the sensor. In a second step, we re-adapt the application by switching from pull to push communication between the components and by moving decisions partially to the robot.

In Listing 3, we show a snippet from the RDF[5] provided by internal resources of the sensor component. For every person in front of the depth video sensor, the integrated tracking software calculates – once the person is recognised – coordinates of the center of mass and of different joint points as well as confidence values for each coordinate. In the listing, we show one out of several coordinates in the tracked skeleton of a person that, joined with all other tracked skeletons, forms the internal knowledge graph, which is included in each interpreter run of the sensor component.

C. Deployment and Adaptability of Decentralised Logic

TABLE III
ADAPTATION OF SMART COMPONENT C1 (SENSOR) AT RUNTIME

#	Identifier	Method	Content Type
	Payload		
1	http://c1/scenario	PUT	text/turtle
	<> ldfu:delay 100 ; a ldfu:Configuration .		
2	http://c1/scenario/p/pro1	PUT	text/n3
	{ ?point nirest:coordinate ?coordinate . ?coordinate nirest:x ?x ; nirest:y ?y ; nirest:z ?z . (?x "2") math:exponentiation ?x_ex . (?y "2") math:exponentiation ?y_ex . (?z "2") math:exponentiation ?z_ex . (?x_ex ?y_ex ?z_ex) math:sum ?sum . ?sum math:sqrt ?square_root . ?square_root math:lessThan "1000.0" . } => { ?point scenario:distance ?square_root . ?point scenario:alarm "true" . } .		
3	http://c1/scenario/r/res1	PUT	application/sparql-query

	CONSTRUCT { ?point scenario:shutdown "true" . }
	WHERE { ?point scenario:alarm "true" . }		

We assume that both components are already deployed and started, but neither provide the data nor the communication that is required by our evaluation scenario. The components provide their meta interfaces as entry point for adaptation at "http://c1/" (C1) and "http://c2/" (C2). We list all HTTP interactions with the meta interfaces in Table III (C1) and Table IV (C2). Specified for each command are the identifier, i.e., the target URI, the method, i.e., HTTP verb, the content type, as well as the payload to be sent to the target URI. These commands may be executed by any HTTP-conform client.

TABLE IV
ADAPTATION OF SMART COMPONENT C2 (ROBOT) AT RUNTIME

#	Identifier	Method	Content Type
	Payload		
4	http://c2/scenario	PUT	text/turtle
	<> ldfu:delay 100 ; a ldfu:Configuration .		
5	http://c2/scenario/p/pro1	PUT	text/n3
	{ [] http:mthd httpm:GET ; http:requestURI <http://c1/scenario/r/res1> . }		

We first initialise an interpreter instance, i.e., LD-Fu instance, for our evaluation scenario at both components by executing Command #1 in Table III for C1 and Command #4 in Table IV for C2. With distinct interpreter instances, we support separately interpreted programs and thus enable the smart component to be part of multiple independent distributed applications at the same time. Both commands create interpreter instance resources, identified by "http://c1/scenario" for C1 and "http://c2/scenario" for C2. They contain the same payload, which configures the LD-Fu engine to continuously interpret programs with a delay of 100 milliseconds.

As part of the meta interface, interpreter instance resources provide container[6] sub-resources, which allow the creation of programs and declared resources, e.g., for C1 the container for programs is identified by "http://c1/scenario/p/" and the container for declared resources is identified by "http://c1/scenario/r/". Both, programs and declared resources, that are created as sub-resources of these containers, are included and evaluated during each interpreter run. To adapt the component to our evaluation scenario, we interact with the program container of C1 and create a program sub-resource, identified by "http://c1/scenario/p/pro1", as shown in Command #2 in Table III. The program included in the payload contains a single rule, which calculates the Euclidean distance from the sensor for every point in the internal RDF knowledge graph. Therefore, built-in mathematical functions are used, that are interpreted by the LD-Fu engine, which calculates the result and binds it to the specified variables. Once the condition – distance less than 1000 millimetres – is true, the internal RDF graph is enriched with a triple adding the distance as well as a triple adding a custom alarm to the subgraph of the respective point. We are now able to use this information in further rules and queries.

[5]Due to space constraints, prefixes are omitted in all listings and tables.

[6]The container resources are designed as LDP basic containers.

D. Deployment and Adaptability of Interfaces and Interaction

With respect to our evaluation scenario, we need to establish a communication between the components. In particular, C2, the robot, should request information about an emergency shutdown, which C1, the sensor, provides.

First, we use a declared resource for passive provisioning of alarm information, which is created as sub-resource of the resources container by Command #3 in Table III and identified by the URI "http://c1/scenario/r/res1". The SPARQL CONSTRUCT query, that we include as payload, constructs a custom shutdown triple if the internal RDF graph was enriched by the calculation program, i.e., sub-graphs of points have been marked with an alarm triple. During every interpreter run, i.e., every 100 milliseconds, the SPARQL CONSTRUCT query is evaluated and the result provided for HTTP requests[7] at the identifier.

Second, we use an interaction rule deployed within a program at C2 to request the information from the declared resource of C1. Command #5 in Table IV shows the program as payload, which is identified by the URI "http://c2/scenario/p/pro1". The single rule included in the program is a head-only rule, i.e., no condition in the rule body has to be meet and the head is evaluated during every run. We use ontologies, in particular marked by the prefixes "http" and "httpm", that are interpreted by the LD-Fu engine as HTTP commands and executed as HTTP requests. The payload of the requests, if given and parseable as RDF, is added to the internal RDF graph of the current interpreter run and may be subject to further rules. In our case, we cause the interpreter of C2 to issue a HTTP GET request to the content of the declared resource of C1, identified by the URI "http://c1/scenario/r/res1", during every run and add it to the internal RDF graph.

Thereby, the data flow between C1 and C2, as required by the first version of the evaluation scenario, is established in a pull-based manner. We do not explicitly show how information about a shutdown is internally handled by C2. This may be, for example, analogously solved by a query registered at the interpreter that causes the robot to react appropriately if the information about an emergency shutdown is available.

E. Re-Adaptation of the Application to new Requirements

In order to show the flexibility of our approach, we re-adapt the distributed application at runtime to new requirements. Instead of pulling shutdown information from the sensor (C1), the robot (C2) should get informed as soon as a distance alarm is recognised. Furthermore, the C2 should itself decide to inform a subcomponent (C3), if the distance is less than half of the original threshold. We show the commands used for the adaption in Table V.

First, we re-adapt the program that we deployed before at C2, as shown in Command #6 in Table V. The program, identified by "http://c2/scenario/p/pro1", is overwritten by single rule encoding the decision and interaction with C3. Therefore, once information about a point, marked with an alarm and accompanied by a threshold and distance value, is available in the internal RDF graph of C2, the threshold is divided by two and compared to the distance. If the condition is true, the rule head is evaluated by the interpreter, which includes an interaction rule that updates a resource at C3, identified by "http://c3/res". Compared to the situation before, we split calculation logic and distributed it to two components.

TABLE V
RE-ADAPTATION OF SMART COMPONENTS C1/C2 AT RUNTIME

#	Identifier / Payload	Method	Content Type
6	http://c2/scenario/p/pro1	PUT	text/n3
	{ ?point scenario:alarm "true" ; scenario:threshold ?threshold ; scenario:distance ?distance . (?threshold "2") math:quotient ?quotient . ?distance math:lessThan ?quotient . } => { [] http:mthd http-m:PUT ; http:requestURI <http://c3/res> ; http:body { <> scenario:shutdown "true" . } . } .		
7	http://c1/scenario/p/pro2	PUT	text/n3
	{ ?point scenario:alarm "true" ; scenario:distance ?distance . } => { [] http:mthd http-m:POST ; http:requestURI <http://c2/scenario> ; http:body { <> scenario:alarm "true"; scenario:threshold "1000.0"; scenario:distance ?distance . } . } .		
8	http://c1/scenario/r/res1	DELETE	-
	-		

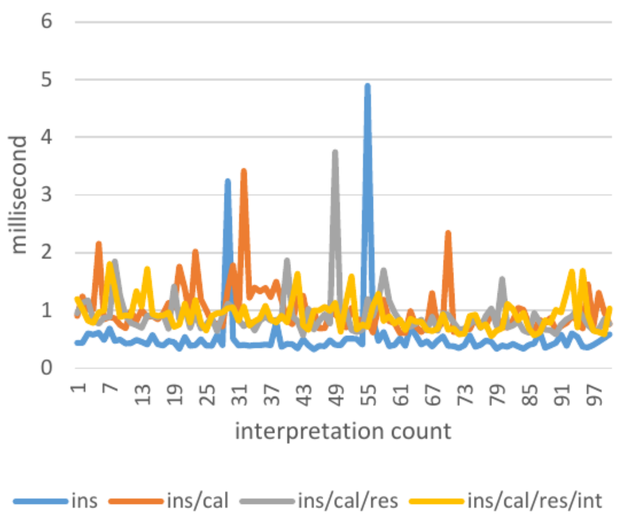
Second, we deploy a separate program with an interaction rule at C1, shown in Command #7 in Table V and identified by the URI "http://c1/scenario/p/pro2". Thereby, we replace the – already overwritten – pull-based interaction of C2 with C1 by a push-based interaction of C1 with C2. The program included as payload consists of one interaction rule that instructs the interpreter to execute a HTTP POST request at the URI of the interpreter instance resource for our evaluation scenario of C2, i.e., at "http://c2/scenario". In the payload we include the distance and threshold information. The execution is triggered by information about points that have been marked with an alarm triple and corresponding distance. HTTP POST request at interpreter instance resource cause the interpreter to add a give payload to the internal RDF graph and run an interpretation of all programs and declared resources.

Optionally, we can delete the obsolete declared resource created for the first version for our evaluation scenario, shown in Command #8 in Table V.

F. Performance Measures

Finally, we focus on processing overhead caused by the interpreter instance and program runs. In Figure 4 we visualise the measurements of interpreter runtimes for our scenario

[7]The result is provided for any valid content type of a supported RDF serialisation format set in the accept header of the HTTP GET request.



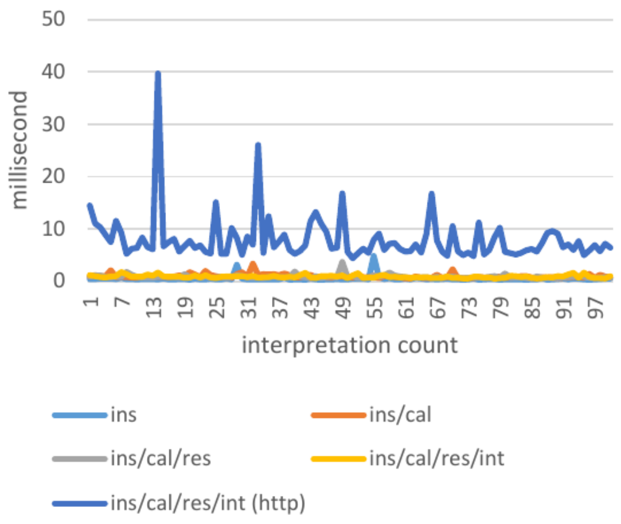


Fig. 4. Runtimes of Interpretations; HTTP interaction excluded/included

example. During 100 consecutive runs, with 100ms delay between interpretations, on an average computer[8], we measured the duration of running an instance without programs or declared resources (ins), and sequentially added the calculation program (ins/cal), the declared resource (ins/cal/res), and the interaction program (ins/cal/res/int). The diagram at the top shows the measured durations, without the actual execution of the HTTP interaction. In the diagram at the bottom, we include the HTTP interaction to a host on the same machine (ins/cal/res/int (http)). Despite some outliers, the duration for an instance without programs or declared resources (ins) is about 0.4ms (median: 0.42ms), and the duration for an instance with programs and resources is about 0.9ms (median: 0.86ms, 0.83ms, and 0.89ms). Finally the measured duration with executed local HTTP interaction is about 6.6ms (median: 6.63ms). Therefore, the overall overhead of using smart components results in minimal delays.

[8]Intel Core i7-3520M CPU @ 2.90GHz.

VI. Related Work

Related work can be split into three main areas: 1) read-write Linked Data (LD), 2) the Web of Things (WoT), and 3) the Semantic Web of Things (SWoT).

Read-write Linked Data Read-write Linked Data is build upon the idea of combining the architectural paradigms Linked Data (LD) [6] and Representational State Transfer (REST) [5]. This combination has been used in several approaches, e.g., Linked Data Fragments (LDF) [18], Linked APIs (LAPIS) [7], Linked Data Services (LIDS) [19], RESTdesc [20], or Linked Open Services (LOS) [21]. Recently, standardization efforts for the integrated use of Linked Data and REST led to the Linked Data Platform (LDP) [9] W3C recommendation, that standardizes a read-write Linked Data architecture, including RDF and non-RDF resources, container resources for collections, as well as rules for HTTP-based interaction with resources.

Web of Things The IoT [22] paradigm is about connecting every device, application, object, i.e., thing, to the network, in particular the Internet and thus to ensure connectivity. The Web of Things (WoT) [23] builds on top of this paradigm to provide integration not only on the network layer but also on the application layer, i.e., the Web. The goal is to make things part of the Web by providing their capabilities as REST services. Therefore, common existing Web technologies are introduced, e.g., URIs for identification and HTTP as application protocol for transport and interaction. Integrating these technologies has been, for example, addressed for embedded devices in [24].

Semantic Web of Things The extension of IoT to WoT is primarily focused on the interoperability between things on the application layer. In order to foster horizontal integration and interoperability the Semantic Web of Things (SWoT) [25] focuses a common understanding of multiple capabilities and resources towards a larger ecosystem by introducing Semantic Web technologies to the IoT. Challenges related to SWoT have been, for example, addressed by the SPITFIRE [26] project, or the Micro-Ontology Context-Aware Protocol (MOCAP) [27], both in the area of sensors. We build upon several synergies introduced by a common resource-oriented viewpoint of the Linked Data and REST paradigms. These paradigms also play a key role in WoT and in particular SWoT to cope with heterogeneous data models and interaction mechanisms. However, integrating decentralised components into applications without central control, even with a clear interaction model and semantically powerful data model, requires to distribute the controlling intelligence, at least to some extent, to the components. Our approach aims to enable the adaptation of components to specific application scenarios at runtime, while still being compatible with other approaches based on read-write Linked Data REST resources.

VII. Conclusion

Currently we are witnessing the increased use and popularity of mobile devices, wearables and sensors. This trend impacts not only our daily lives but also the way that products and services are being designed, manufactured and offered.

The IoT and WoT lay the foundation for integrating devices by providing network connectivity and a stack of communication protocols. The SWoT aims to enhance these in order to address the lack of interoperability, resulting from the heterogeneity of data and resources. We aim to contribute to the evolution of the Web by taking these developments one step further. In particular, our work focuses on two main aspects: overcoming not only data but also device and interface heterogeneity, and enabling adaptable and scalable decentralised WoT applications. To this end we use semantics to capture the devices' capabilities and interfaces, and rules to enable embedding controller logic within the device's interfaces for supporting a decentralised solutions. We support the reconfiguration of the controller logic at runtime in order to provide options for customising and adapting the application. Furthermore, we show how our approach can be applied by introducing a reference architecture, backed up by a specific framework implementation. We believe that providing support for interoperability but also offering simple mechanisms for adapting the interfaces and controller logic of devices are key for contributing towards the evolution of the Web.

References

[1] P. Hitzler, M. Krotzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*. CRC Press, 2009.

[2] M. Jackson, "Defining a Discipline of Description," *IEEE Software*, vol. 15, no. 5, pp. 14–17, 1998.

[3] T. S. Dillon, H. Zhuge, C. Wu, J. Singh, and E. Chang, "Web-of-things framework for cyber-physical systems," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 9, pp. 905–923, 2011.

[4] National Science Foundation, "Cyber-Physical Systems Summit," National Science Foundation, Tech. Rep., 2008.

[5] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, USA, 2000.

[6] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[7] S. Stadtmüller, S. Speiser, and A. Harth, "Future Challenges for Linked APIs," in *Workshop on Services and Applications over Linked APIs and Data at the European Semantic Web Conference*, 2013.

[8] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C, Recommendation, 2014, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Latest version available at http://www.w3.org/TR/rdf11-concepts/.

[9] S. Speicher, J. Arwe, and A. Malhotra, "Linked Data Platform 1.0," W3C, Recommendation, Feb. 2015, http://www.w3.org/TR/2015/REC-ldp-20150226/. Latest version available at http://www.w3.org/TR/ldp/.

[10] M. Maleshkova, P. Philipp, Y. Sure-Vetter, and R. Studer, "Smart Web Services (SmartWS) - The Future of Services on the Web," *Transactions on Advanced Research*, vol. 12, no. 1, pp. 15–26, 2016.

[11] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.

[12] T. Berners-Lee and D. Connolly, "Notation3 (N3): A readable RDF syntax," W3C, Team Submission, 2011, http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328//. Latest version available at https://www.w3.org/TeamSubmission/n3/.

[13] C. B. Aranda, O. Corby, S. Das, L. Feigenbaum, P. Gearon, B. Glimm, S. Harris, S. Hawke, I. Herman, N. Humfrey, N. Michaelis, C. Ogbuji, M. Perry, A. Passant, A. Polleres, E. Prud'hommeaux, A. Seaborne, and G. T. Williams, "SPARQL 1.1 Overview," W3C, Recommendation, Mar. 2013, http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. Latest version available at http://www.w3.org/TR/sparql11-overview/.

[14] S. Stadtmüller, S. Speiser, A. Harth, and R. Studer, "Data-Fu: A Language and an Interpreter for Interaction with Read/Write Linked Data," in *International World Wide Web Conference*, 2013.

[15] S. Stadtmüller, "Dynamic Interaction and Manipulation of Web Resources," Ph.D. dissertation, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2016.

[16] A. Harth, C. A. Knoblock, S. Stadtmüller, R. Studer, and P. A. Szekely, "On-the-fly Integration of Static and Dynamic Linked Data," in *International Workshop on Consuming Linked Data at the International Semantic Web Conference*, 2013.

[17] F. L. Keppmann and S. Stadtmüller, "Semantic RESTful APIs for Dynamic Data Sources," in *Workshop on Services and Applications over Linked APIs and Data at the European Semantic Web Conference*, 2014.

[18] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle, "Querying Datasets on the Web with High Availability," in *International Semantic Web Conference*, 2014.

[19] S. Speiser and A. Harth, "Integrating Linked Data and Services with Linked Data Services," in *Extended Semantic Web Conference*, 2011.

[20] R. Verborgh, T. Steiner, D. van Deursen, R. van de Walle, and J. Gabarró Vallès, "Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc," in *International Conference on Next Generation Web Services Practices*, 2011.

[21] R. Krummenacher, B. Norton, and A. Marte, "Towards Linked Open Services and Processes," in *Future Internet Symposium*, 2010.

[22] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, pp. 2787–2805, 2010.

[23] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices," in *Architecting the Internet of Things*. Springer Berlin Heidelberg, 2011, pp. 97–129.

[24] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "The Web of Things: interconnecting devices with high usability and performance," in *International Conference on Embedded Software and Systems*, 2009, pp. 323–330.

[25] A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic Web of Things: an analysis of the application semantics for the IoT moving towards the IoT convergence," *Web and Grid Services*, vol. 10, no. 2-3, pp. 244–272, 2014.

[26] D. Pfisterer, K. Romer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kröller, M. Pagel, M. Hauswirth, M. Karnstedt, M. Leggieri, A. Passant, and R. Richardson, "SPITFIRE: Toward a Semantic Web of Things," *Communications Magazine*, vol. 49, no. 11, pp. 40–48, 2011.

[27] K. Sahlmann and T. Schwotzer, "MOCAP: Towards the Semantic Web of Things," in *Posters and Demos at the International Conference on Semantic Systems*, 2015.