Exposing Internet of Things Devices via REST and Linked Data Interfaces*

Sebastian Richard Bader, Tobias Käfer, Lars Heling, Raphael Manke, and Andreas Harth

Institute AIFB, Karlsruhe Institute of Technology (KIT), Germany
{sebastian.bader|tobias.kaefer|harth}@kit.edu,
{lars.heling|raphael.manke}@student.kit.edu

Abstract. Web technologies are widely used open standards to interconnect devices on virtually any platform. The W3C's Web of Things effort wants to bring web technologies to the Internet of Things to address the interoperability challenges in this area and to transfer the extensive work on Web standards towards this domain. In this paper, we report on four independently developed implementations whose aim was to expose Internet of Things devices via web technologies, more precisely REST and Linked Data interfaces. While they were all developed with the same goal in mind, and with technologies that promote uniformity on the interfaces, the implementations still exhibit different schemas and architectures with different communication patterns.

1 Introduction

In the last couple of years, developments of embedded devices have made networked sensors and actuators from various vendors in many domains affordable. To provide a common term for considerable adoption and deployment of such devices, the term Internet of Things (IoT) has been coined. But to expose IoT device data and functionality on a network interface, various vendors in different domains came up with contrary ways of describing data and interacting with the devices. This heterogeneity makes an integrated processing of data from various devices a hard task: For example, manufacturing machine suppliers equip their machines with numerous networked sensors and actuators. While the machines from the individual suppliers can be optimised using its defined methods, holistic optimisation over the whole shop floor of a machine operator is hindered because of incompatible protocols, data formats or other interoperability problems. Similarly, in the smart home scenario, there are solutions like ZigBee Light Link[1] (Philips Hue[2], Osram Lightify[3]), which give interoperability on a local scale in

*A preliminary version of this work was presented at the 2nd International Workshop on Interoperability & Open Source Solutions for the Internet of Things, 2016

[1] http://www.zigbee.org/zigbee-for-developers/applicationstandards/zigbee-light-link/

[2] http://www.meethue.com/

[3] http://led.osram.de/led_de/lightify/lightify-produkte/index.jsp

the light domain. But their heterogeneity is a non-trivial challenge when building applications that make use of devices from different domains and vendors. This topic needs to get addressed, at latest for the Industrial Internet of Things (also known as Industrie 4.0), where connections between heterogeneous systems of different companies have to be designed, built, and maintained which are driven first and foremost by economic considerations and not common technology.

Before IoT applications can be built, the devices' data has to be made accessible. Accomplishing data accessibility, one faces two degrees of freedom:

1. How to interact with the device to obtain the data?
2. How to describe and represent the data?

We propose to use established Web technologies to build IoT systems, thus embracing the W3C's Web of Things effort (see also Section 2). Specifically, we rely on HTTP (the Hypertext Transfer Protocol), an implementation of the REST (representational state transfer) architectural style, for the interaction and on RDF (the Resource Description Framework), a data model for representing the data. This combination is also referred to as Linked Data as it allows a connection of data entities through HTTP links. Although we take those decisions upfront, the technologies HTTP and RDF still present us with choices for both degrees of freedom even for 1.

Before the advent of IoT, the proliferation of mobile devices gave rise to cloud applications that expose RESTful APIs (Application Programming Interface) for mobile apps, which mainly only consume data. The architecture of such cloud applications is characterised by a central server, on which all users' data is maintained. Such centralisation of data brought privacy issues and led to data silos with a proprietary data model, where it is hard to get any data out. Nevertheless, such an architecture depending on a central server is also a way to connect IoT devices. But while in the mobile apps scenario, the distributed and power-limited device is mainly the consumer of data, regarding IoT we have highly distributed, unreliable and power-limited devices data producers.

In this paper, we want to present two design approaches to get data and functionality from sensors and actuators accessible through web technologies such that they can get integrated in applications. Those applications, as regarded in this paper, are research prototypes that consume Linked Data APIs. The paper is a report on four independently developed implementations, two for each approach, that we independently carried out from scratch to provide Linked Data access to our IoT devices. The paper is structured as follows: In Section 2, we give basic definitions and introductions to the technical terms we use in the remainder of the paper. We describe four implementations to get sensors and actuators into the Internet of Things in Section 3. Last (Section 4), we summarise our observations and conclude.

2 Preliminaries

The Internet of Things is under active development, with many standardisation organisations (e.g. the ISO/IEC Internet of Things working group) and industry-

led consortia (e.g. the AllSeen Alliance[4], the Open Connectivity Foundation[5]and the Industrial Internet Consortium[6]) that work on creating and establishing dedicated IoT standards. Given that these efforts are rather young and their standardisation processes are in constant flux, we do not attempt to provide a survey of the different approaches. Rather, we embrace the W3C's effort around the "Web of Things"[7], assuming IoT is going to use established web technologies such as URIs and HTTP. We therefore share the aim and base technologies with the Web of Things effort, despite our research focus is different from the standardisation work of the Web of Things group. While the Web of Things effort is mainly concerned with descriptions of offerings and capabilities of things that *may* have Linked Data interfaces, we assume Linked Data interfaces and are concerned with the semantic data provided by the thing, the interaction with the thing and the execution of programs that employ the thing of interest. On the other hand, we note that there are other less widely deployed web standards such as WebSockets, which can also be used to connect IoT devices [10].

The web can be described as a system that implements the architectural style of Representational State Transfer (REST) [8]. We use the terminology introduced in [8] to describe and contrast our different approaches for accessing IoT devices as part of applications. Broadly speaking, applications following the RESTful architectural style consist of data, connectors and components. In the remainder of the section, we address each in turn, starting with data.

In RESTful architectures, the central notion is that of a "resource"; a resource is any identifiable and referenceable thing in the context of an application. To reference resources, we use URI templates as specified in [9]. To access and transfer the potentially dynamically changing state of resources between components, we need a way to represent the specific, currently valid state. In addition, those resource representations need to include references to other resources in order to allow linking and discovering previously unknown resources in decentralised networked environments such as the Web or the Web of Things.

Connectors are concerned with the transfer of representations of resource states between components [8]. Connectors provide, in other words, the means for communication. REST defines clients, servers, cache, resolvers, and tunnels. From this list, only the two first are relevant in the scope of this paper as we only regard basic interactions. For the figures, we use components from the UML component diagram, where a REST server connector is depicted as a UML provided interface, and a REST client connector as a UML required interface. We use the Hypertext Transfer Protocol (HTTP) [7] as our communication protocol. An interaction between components in HTTP consist of polling, in particular a request-response pair of messages. Depending on the type of request, the message may includes data in the body section of the response.

[4] `http://allseenalliance.org/`
[5] `http://openconnectivity.org/`
[6] `http://www.iiconsortium.org/`
[7] `http://www.w3.org/Submission/wot-model/`

In REST, a connector to send or answer requests resides on a so-called component. Among intermediaries (see Section 3.2), there are user agents (the client program initiating a request to a resource) and origin servers (the source of authoritative information on the resource). The terms client and server are only concerned with the roles in an exchange of one request-response pair, as one component may act as client in one exchange and as server in a different exchange. For our implementations, we started out from scratch and different circumstances made us go different routes, which we describe in Section 3.

The Linked Data principles [2] are a set of practices to publish data on the web making use of the combination of RDF and HTTP: The first two principles suggest to rely on HTTP URIs to identify things for the retrieval of resources. The third principle recommends the use standard-conforming data representations such as RDF. To enable the discovery of new information, the fourth principle advocates to include links in the representation to other relevant data.

While the Linked Data principles are about data publishing, the interaction with web resources was not in their scope. The W3C's Linked Data Platform (LDP)[8] specification closes the gap between the HTTP and the RDF specification for RESTful interactions with web resources. The recommendation defines Linked Data Platform Resources and Linked Data Platform Containers. A LDP Resource guarantees a minimal set of common read operations and specifies how servers publishing such resources need to react to HTTP requests. LDP containers are collections of LDP Resources with additional functionalities for creating new resources.

3 Approaches and Implementations

The first two implementations directly connect the devices with sensors/actuators to the web. The second two implementations use an intermediary. All approaches expose the information in form of web resources. We describe the approaches and implementations in this section. We cover in detail the involved components and their interaction. We also name the vocabularies that are used for describing the data for each implementation, but omit the detailed presentation of the RDF data, as this is not in the focus of the paper. Yet, we provide links to the source code of each implementation for the interested reader.

3.1 Direct Access to the Device with the Sensor/Actuator

In this section, we present two implementations that implement the REST component type "origin server", i. e. there is direct access to the definitive source of information about the resources. The origin server thus implements a REST "server connector".

[8] `http://www.w3.org/TR/ldp/`

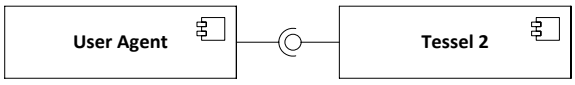

Fig. 1. Component Diagram of the Implementation using the Tessel 2.

Connecting two Modules of a Tessel 2 In this section, we describe how we built a REST and Linked Data interface for the Tessel2[9]. The corresponding code can be found online[10]. Tessel2 is a PCB (Printed Circuit Board) that has been developed to easily build networked sensor/actuator devices. The PCB runs Node.js[11] such that one can program the PCB in JavaScript. We use a Tessel2 PCB with the ambient module, which includes sensors for light and sound, and the relay module, which includes two relays to switch 240 V at 5 A, see Figure 1.

URI Template	Method	Description
`/ambient/`	`GET`	Returns an overview over the available sensors of the ambient module.
`/ambient/{sensor}`	`GET`	Requests a reading from the sensor, i. e. light or sound, and returns it.
`/relay/`	`GET`	Returns an overview over the available power switches.
`/relay/{n}`	`GET`	Returns the status for the n-th power switch.
`/relay/{n}`	`PUT`	Sets the status for the n-th power switch.

Table 1. API Description of the Implementation for the Tessel 2.

On the HTTP interface, we describe the board with its two sensors using the LDP vocabulary. The root resource thus points to two resources describing the modules through the `ldp:contains` property and declaring it as a LDP Container. The ambient module points to the LDP RDF Source Resources for the sensors in the same manner. The sensors are described as observations from the data cube vocabulary[12]. The sensor readings are connected to the sensor using a custom property. The relay module also uses the `ldp:contains` property to point to its relays. The relay's state (on/off) is also connected to the relay itself using a custom property. Using a PUT request on the relay's URI sending the new desired state of the relay leads to an update of its representation and consequently switches the connected relay on and off. Relying on the LDP definitions of Container and RDF Source Resource implicitly contains the information about possible interaction patterns. We implemented the HTTP interface using the Express framework[13] for building Web APIs in JavaScript and JSON-LD as way to represent the RDF data.

[9] `http://tessel.io/`
[10] `http://github.com/kaefer3000/t2-rest-relay-ambient`
[11] `http://nodejs.org/`
[12] `http://purl.org/linked-data/cube#`
[13] `http://expressjs.com/`

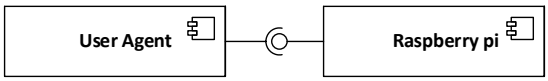

Fig. 2. Component Diagram of the Implementation using the SenseHat, which is Connected to a Raspberry Pi.

Connecting a SenseHat In this section, we describe how we built a REST and Linked Data interface to interact with a Sense Hat[14], an add-on board for the Raspberry Pi. The Sense Hat board offers several sensors (humidity, gyroscope, accelerometer, magnetometer, temperature, barometric pressure), and a programmable LED matrix. The code can be found online[15].

To describe the data on the interface, we use parts of the popular Semantic Sensor Network ontology (SSN) [4]. But SSN proved not to be specific enough as way to specifically describe one device is not part of the SSN ontology, nor how the data of a sensing device can be structured. Therefore, we extended the SSN ontology by the class `LedMatrix`. Each sensor on the Sense HAT board is an instance of the SSN class `SensingDevice`, and the LED matrix of `LedMatrix`. A `SensingDevice` is connected to the quantity kind (such as humidity and temperature) it observes using the `featureOfInterest` property. On top of that a `SensingDevice` is connected to the RDF literal representing the measured value using the `observationValue` property and to the URI representing the corresponding unit using the `observationUnit` property. Those descriptions can make use e. g. of dbpediaor the QUDT ontology[16]. Using the `hasLed` property, the connection between a `SingleLed` (defined by their X and Y coordinate) and a `LedMatrix` can be stated. The colour of each `SingleLed` can be defined using the three RGB properties in a range from 0 and 255. We serve dereferencable URLs for the data about the Sense Hat as described in Table 2.

We implemented the interface to the Sense Hat of the Raspberry Pi in Python. The Sense HAT python library[17] allowed to program access to the sensors of the Sense Hat and the LED matrix. We used the framework Flask[18] to implement the interaction with the resources. The library RDFlib[19] served in the serialisation and deserialisation of RDF data. Upon a request to a URI on the REST interface, the corresponding RDF data is retrieved from the Sense Hat, and then sent as a response. Some extra focus must be taken on the way the LEDs can be updated when several LEDs are addressed in the same request. We rely on N-ary relations with blank nodes, with relations to the x and y coordinates identifying the LED and an additional predicate for its new state.

[14] `http://www.raspberrypi.org/products/sense-hat/`
[15] `http://git.scc.kit.edu/ujdpo/sensehat`
[16] `http://www.qudt.org/`
[17] `http://pythonhosted.org/sense-hat/`
[18] `http://flask.pocoo.org/`
[19] `http://github.com/RDFLib`

URI Template	Method	Description
`/`	`GET`	Returns information on the Raspberry Pi linking to the installed boards, e. g. the Sense Hat.
`/sensehat/`	`GET`	Returns information on the Sense Hat linking to the sensors and the LED matrix.
`/sensehat/led/`	`GET`	Returns information on the LED matrix linking to the individual LED and all LEDs' current value.
`/sensehat/led/`	`PUT`	Overwrites the information on the LED matrix. Can be used to set the values of the LEDs.
`/sensehat/led/{x}/{y}`	`GET`	Returns the colour of the LED at (x, y) on the LED matrix.
`/sensehat/{sensor}`	`GET`	Returns the value of a sensor.

Table 2. API Description of the Implementation for the SenseHat.

3.2 Access via an Intermediary to the Device with the Sensor/Actuator

In addition to the direct access to the information source we present two implementations that use an intermediary. There are two components: one component that has a sensor/actuator connected (which we call the sensor/actuator-bearer) and a client connector. The second component, the intermediary, with a server connector exposing a resource that represents the state of the sensor/actuator connected to the other component. The state of this resource is periodically updated by the sensor/actuator-bearer using PUT requests.

What is this Approach in REST? In REST terms, this pattern is hard to grasp. One could argue that there is a resource on the first connector that cannot get accessed directly, because the sensor/actuator-bearer has no server connector, and that there is a logical correspondence between the not directly accessible resource and the accessible resource on the second component. The resource on the intermediary "represents" the sensor or actuator although it is only loosely coupled. Next, we discuss the origin server and the different intermediary component types REST offers and why they do not fit the pattern:

Origin Server An origin server is defined as "the program that can originate authoritative responses for a given target resource" [7]. While our intermediary is the source for information on a given resource, the authoritative source is the sensor/actuator-bearer. Moreover, the term origin server would fit for the latter, because a characteristic of an origin server is to "be the ultimate recipient of any request" [8]. But the sensor-bearer cannot be called origin server, because it does not have a server connector, therefore it cannot give authoritative *responses*.

Proxy A proxy is defined as "a message-forwarding agent that is selected by the client [...] to receive requests [...] and attempt to satisfy those requests via translation through the HTTP interface" [7]. While our intermediary is not

selected by the client, because the client does not know that the source of the information is not the intermediary, our scenario looks like the example proxy in [8, Fig. 5-10 c].

Gateway A gateway is defined as "intermediary that acts as an origin server for the outbound connection but translates received requests and forwards them inbound to another server or servers" [7] While in our case the intermediary indeed acts as an origin server, it does not forward the request. On the other hand, the encapsulation of other services is one of the examples of the gateway, and it may communicate with other servers using any protocol [7]. Thus, the gateway could be regarded as the closest fit.

The Intermediary Strategy as a Way of Addressing IP-layer The approach with a central server is necessary e. g. in situations where the ongoing transition from IPv4 [12] to IPv6 [5] is solved using so-called DS-lite [6] connections: In IPv4, every consumer gets one single an IPv4 address from his Internet Service Provider (ISP), which is reachable from the Internet. For a provider to nevertheless connect multiple devices to the internet, network address translation (NAT) [13] is used: The customer employs a router, which gets the one IPv4 from the ISP. The devices on the local network can connect to the Internet by sending a request to the router, which replaces the local IP that initiates the connection with his own IP, and assigns a free port to the connection. The router then forwards data that comes to this port from the Internet to the local IP and port that initiated the connection. All computers in the local network thus appear as one computer on the Internet. Connections initiated from the Internet can only reach a local device if the router is configured to do port forwarding: The router maintains a mapping from his own ports to a tuple of local IP and port of local devices and forwards traffic accordingly.

As we go from IPv4 to IPv6, many ISPs employ a technique called Dual Stack lite (DS-lite) [6]: Instead of giving both an IPv4 and IPv6 address to their customers (Dual Stack [11]), ISPs only give IPv6 addresses to their customers. If the customer requests a connection to an IPv4 address on the Internet, the ISPs tunnel the traffic accordingly. Conversely, this means that there cannot be a IPv4 connection initiated from the Internet, because the customer does not have an IPv4 address. This is particularly an issue for devices on cellular network, where typically only IPv4 is deployed. Moreover, if no further port forwarding is employed, only connections to IPv6-enabled local devices can be initiated from the Internet. However, not only the deployment of DS-lite is an argument for the proposed architecture, but also the fact that it alleviates the necessity to have access to the router with the necessary rights to configure the port forwarding.

Addressing Device Limitations using the Intermediary Strategy We imagine two scenarios here:

- The continuous availability of the device is not guaranteed, but a high availability of the data is important. An off-the shelf caching intermediary for HTTP would be sufficient here, though

– The intermediary can answer more complex requests on top of HTTP based on the same data. To have the data both on the device and the component, which is proposed to be an intermediary, would be an unnecessary duplication of data.

Connecting Weather Sensors and a 433 MHz Transceiver In this section, we describe how we built an Linked Data interface to (a) the functionality of a 433 MHz transceiver (b) a directly connected temperature sensor. The code can be found online[20]. In our scenario, the transceiver wirelessly controls two power sockets, and receives data from a weather station, in particular temperature and humidity values. The transceiver is connected to the GPIO pins of a Raspberry Pi together with another temperature and humidity sensor directly connected to the GPIO pins. In this implementation, the access to the client is implemented indirectly: The Raspberry Pi exchanges data with a central server, which stores data about the current state of the sensors and sockets. For the sockets, the data on the server can also mean the desired state. One rationale for this approach is to alleviate the necessity to directly access the clients to retrieve sensor data and control actuators.

The implementation describes a sensor reading as `Observation` from the SSN ontology. To describe the value, we use Wikidata[21] and the Smart Appliances Reference ontology[22]. The location of the reading is described using the WGS84[23] ontology. The values are described using XSD data types.

The client periodically takes snapshots from the sensors directly connected to the Raspberry Pi. Additionally, a 433Mhz transceiver is used to first wirelessly receive temperature and humidity values from a weather station sensor every five minutes, and second wirelessly control two power sockets.

All data from the sensors is enriched with meta data, sent to a server where it gets stored in a relational database. Moreover, there are records in the data base for the sockets, too. The relational database is made accessible using a RESTful API, which allows for requests to read and store data. On the interface the sensors and actuators are identified using URIs and therefore can be requested through GET requests and action for actuators can be sent using POST to the corresponding URI.

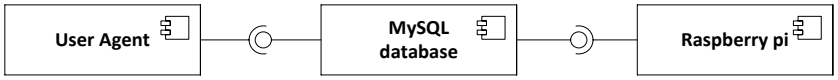

Fig. 3. Component Diagram of the Implementation for the Weather Sensors and the 433 MHz Transceiver, which are connected to a Raspberry Pi.

[20] `http://github.com/Lars-H/home-automation`
[21] `http://www.wikidata.org/`
[22] `http://ontology.tno.nl/saref.ttl`
[23] `http://www.w3.org/2003/01/geo/wgs84_pos`

URI Template	Method	Description
`/sensor/outside/{type}`	`GET`	Returns the latest record of the sensor value for the wirelessly connected weather station.
`/sensor/inside/{type}`	`GET`	Returns the latest record of the sensor value of type humidity or temperature.
`/actuator/plug{n}/status`	`GET`	Get the current status for the n-th wireless power plug.
`/actuator/plug{n}/status`	`PUT`	Set the status for the n-th wireless power plug.

Table 3. API Description of the Implementation for the Weather Sensors and the 433 MHz Transceiver.

The database is a MySQL database which can be accessed using an HTTP interface to store and retrieve data. The REST-interface is realised as a server using the Flask Framework[24] for building RESTful APIs in Python. A description of the API can be found in Table 3. The API supports content negotiation for different RDF serialisations.

Connecting a SensorTag The goal of this approach is to publish various outputs of a Texas Instruments SensorTag CC2650[25]. The SensorTag delivers sensors for temperature, humidity, acceleration, and light which can be accessed via a Bluetooth Low Energy interface. The range of the bluetooth connection limits the distance between the sensor itself and an according control unit where at the same time the positioning of the SensorTag should not depend on space requirements of a powerful hardware. Further on, we want to query the real-time observations while also be able to access the whole data for analytical tasks on the historical data.

We use the `Observation` from the SSN ontology to describe a sensor reading, and describe the data using XML Schema datatypes, vCard[26], and QUDT. We connect the SensorTag to a Raspberry Pi Model B equipped with a Bluetooth 4.0 dongle. The Raspberry Pi is programmed to periodically request data from the SensorTag using bluepy[27]. Bluepy supplies JSON data. Another process ("Administration Shell" in Industrie 4.0 terminology [1]) checks whether this data from the SensorTag has changed and if so, the process lifts this sensor data to RDF and represents it as an observation according to SSN, XML

Fig. 4. Component Diagram of the Implementation using the SensorTag, which is connected to a Raspberry Pi.

[24] `http://flask.pocoo.org`
[25] `http://www.ti.com/sensortag`
[26] `http://www.w3.org/TR/vcard-rdf/`
[27] `http://github.com/IanHarvey/bluepy`

URI Template	Method	Description
`/marmotta/ldp/ldbc`	`GET`	Responds with information on the used Linked Data Platform Basic Container
`/marmotta/ldp/ldbc`	`POST`	Creates a new Linked Data resource
`/marmotta/ldp/ldbc/{x}`	`GET`	Returns information about the data object x
`/marmotta/ldp/ldbc/{x}`	`PUT`	Overwrites an existing resource or creates a new one
`/marmotta/ldp/ldbc/{x}`	`DELETE`	Erases resource x

Table 4. The Linked Data Platform API as implemented by Apache Marmotta.

Schema datatypes, and vCard. This administration shell sends the RDF data to a collection resource on a Linked Data Platform implementation, more precisely to a `ldp:BasicContainer`. As the LDP nature of the resources is communicated as part of the representation of the root container and its containing resources a client receives enough information to further interact with them as specified by the LDP specification. We use Apache Marmotta as an implementation of the Linked Data Platform, as it both offers a Linked Data interface for RESTful interaction and a SPARQL interface for queries demanded by our analytics use-case. Especially for the requirements of a profound analytical application, we assume the Raspberry Pi not powerful enough to store the amount of data and do a sufficient query processing. That's the reason why we have the Marmotta instance running on an Ubuntu 14.04 based virtual machine, separating the data management from the data producer. The code can be found online[28].

4 Discussion and Conclusion

We described four independently developed implementations of a REST and Linked Data interface for IoT devices. The interface is thus built on open standards. We hid the technology fragmentation of the sensors and actuators behind a uniform interface and data model. The uniform interface of REST and Linked Data makes the devices easier to connect to as a bunch, or together with other devices that share this interface and data model. Despite this uniformity as aim of all implementations, we ended up with four implementations varying in vocabularies and interaction direction with the device that bears the sensor/actuator.

In terms of interaction direction, the implementations can be categorised according to whether there is direct access to the device with the sensor attached or not. While the direct access would be the expected pattern in a RESTful architecture, there are strong reasons why in some settings the access via an intermediary is reasonable. The intermediary again provides a uniform interaction direction in all implementations.

The different implementations resulted in vocabulary heterogeneity, although the use-cases are very similar. This is because while there are elaborate established vocabularies to describe sensors and values, there are no established simple constructs to describe properties of physical objects. Part of the W3C's effort of the Thing Descriptions[29] is to come up with such constructs. They note that

[28] `http://github.com/sebbader/KSRI-KM_STEP`

[29] `http://www.w3.org/WoT/IG/wiki/Thing_Description`

also on the capability description level, ontologies do not much agree on terminology [3]. In the meantime, the use of the semantic data model RDF allows for employing reasoning techniques in the applications that interact with the devices to integrate the data described using different vocabularies.

Acknowledgements This work is partially supported by AFAP, a BMBF Software Campus project (FKZ 01IS12051), the BMBF ARVIDA project (FKZ 01IM13001G), and the BMWi project STEP (FKZ 01MD16015B).

References

1. Umsetzungsstrategie industrie 4.0 (2015), http://www.bitkom.org/Bitkom/Publikationen/Publikation_5575.html
2. Berners-Lee, T.: Design issues – linked data (2009), http://www.w3.org/DesignIssues/
3. Charpenay, V., Käbisch, S., Kosch, H.: Introducing thing descriptions and interactions: An ontology for the web of things. In: 1st Workshop on SemanticWeb technologies for the Internet of Things (SWIT) at ISWC (2016)
4. Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., et al.: The ssn ontology of the w3c semantic sensor network incubator group. Web semantics: science, services and agents on the World Wide Web 17 (2012)
5. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (Dec 1998), http://www.ietf.org/rfc/rfc2460.txt, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112
6. Durand, A., Droms, R., Woodyatt, J., Lee, Y.: Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion. RFC 6333 (Proposed Standard) (Aug 2011), http://www.ietf.org/rfc/rfc6333.txt, updated by RFC 7335
7. Fielding, R., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard) (Jun 2014), http://www.ietf.org/rfc/rfc7230.txt
8. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Diss. University of California, Irvine (2000)
9. Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., Orchard, D.: URI Template. RFC 6570 (Proposed Standard) (Mar 2012), http://www.ietf.org/rfc/rfc6570.txt
10. Merkle, N., Kämpgen, B., Zander, S.: Self-service ambient intelligence using web of things technologies. In: 1st Workshop on Semantic Web Technologies for Mobile and Pervasive Environments (SEMPER) at ESWC (2016)
11. Nordmark, E., Gilligan, R.: Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213 (Proposed Standard) (Oct 2005), http://www.ietf.org/rfc/rfc4213.txt
12. Postel, J.: Internet Protocol. RFC 791 (Internet Standard) (Sep 1981), http://www.ietf.org/rfc/rfc791.txt, updated by RFCs 1349, 2474, 6864
13. Srisuresh, P., Egevang, K.: Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational) (Jan 2001), http://www.ietf.org/rfc/rfc3022.txt