Applying the BaaS Reference Architecture on different classes of devices

Björn Butzin, Björn Konieczek,
Frank Golatowski, Dirk Timmermann
Institute of Applied Microelectronics
and Computer Engineering,
University of Rostock, Germany
bjoern.butzin@uni-rostock.de
bjoern.konieczek@uni-rostock.de

Christoph Fiehe
MATERNA GmbH
Information & Communications
Dortmund, Germany
christoph.fiehe@materna.de

***Abstract*—This paper presents how the reference architecture of the ITEA2 project Building as a Service (BaaS) could be applied on different classes of devices to show its flexibility and suitability for the heterogeneous environment of building automation systems. For this purpose technologies are identified and discussed that could be used for the implementation. The approach of the reference architecture is applied on three different classes of devices that are used in today's building automation systems or are intended to be used in the future. The selected devices span the range from a small sensor node up to a high performance system.**

***Keywords*—building automation; reference architecture; heterogeneous environment; horizontal and vertical integration; service oriented architecture; domain model; software engineering;**

I. Introduction

Today's systems span a wide area of complexity and computing power, starting from high performance clusters with complex parallel programs down to small sensors with quite simple tasks. The systems used depend on the domain to be covered and the tasks to be accomplished. While some domains, such as sensor networks or cloud storage, have quite narrow expectations on the used systems, other domains like factory or building automation span a wider range of systems and devices. The strategies to solve communication of the heterogeneous systems are as numerous as the domains. Solutions span from the use of a middleware, gateways and proxies to special modifications of protocols and encoding. This paper presents how the Building as a Service (BaaS) reference architecture can be applied. The BaaS project [1] focuses on the cost effective and fast development of building automation systems (BAS). The reference architecture was developed to have a common ground in this heterogeneous domain while being open to being adapted to certain use cases. This work sketches how the BaaS reference architecture can be tailored to develop BAS for real world usage in a heterogeneous system environment. As already mentioned, BAS cover systems of different task levels and computing power. On one hand, we have small embedded devices for sensing and interacting with the physical world. On the other hand, we have automation functions, schedulers and controllers. The approach of BaaS also aims to enable the inclusion of so called "value added" services. They could range from optimizers over learning algorithms to remote web services. The variety of tasks also results in a variety of devices used. The reference architecture should be able to cover small embedded sensing devices in the same way as high performance computer systems for optimization or learning tasks. Furthermore, the goal should be to use the same communication protocol and message encoding throughout the whole system to avoid unnecessary effort for re-encoding or gateways. This paper will briefly present the BaaS reference architecture, what it is about and how it can be used in the context of heterogeneous environments. We will focus on the possibilities regarding the scalability of individual services and the development effort. The remainder of this work is structured the following way. Section II presents all parts of the BaaS reference architecture relevant for the purpose of this paper. In a next step some technologies are discussed that could be used for the implementation. Afterwards, in section IV, the applicability of the BaaS approach is demonstrated on three different classes of devices that are used or intended to be used in building automation systems. In section V, we will wrap up the results of this paper.

II. BaaS Reference Architecture

In this section, the important parts of the BaaS domain model and reference architecture are briefly described. In general the domain model captures the main structural concepts of the building automation domain as well as their relationships, constraints and attributes. The main components of the domain model are data points, BaaS service, BaaS container/device and the BaaS registry. The simplified domain model is depicted in fig. 1 . The BaaS domain model is explained in more detail in [2] including its relationship to the Internet of Things Architecture (IoT-A) domain model. The reference architecture adds specific functionality to the individual components as well as methodologies and tools to create, store, generate or deploy code and other artifacts during the software lifecycle.

A. Data Point

The data point is the basic term in the reference architecture. The concept of data points in building automation is not new. In its initial meaning, it describes one

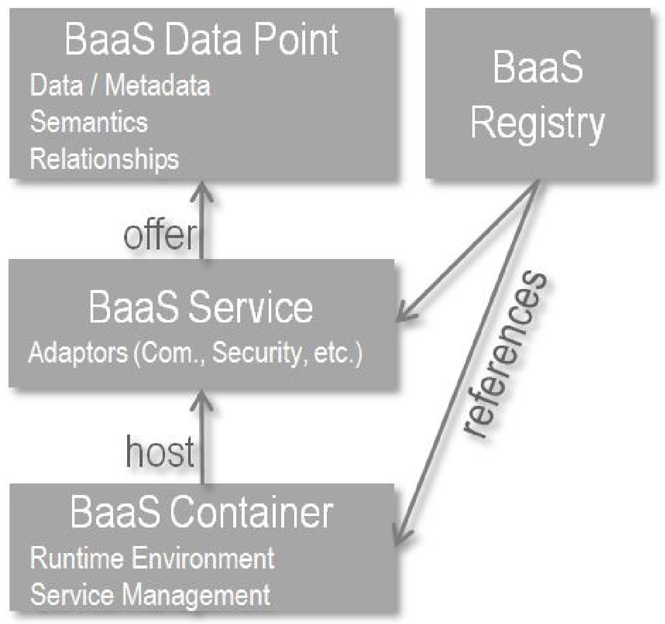

Fig. 1. Simplified BaaS Domain Model

measurement value including its type and unit. Today, the data point is still the basic measurement for the size of a BAS. Nevertheless the meaning of the term has evolved over time. In the context of BACnet (see III-B), e.g., the definition of a data point equals the structural element of a BACnet Object. Thus, it is not just about data type and value, but also about additionally required and optional primitive properties and their values. We can see this as further metadata, i.e., data about the data. In BaaS, two things are added to this definition, semantic information and relationships. Furthermore, BaaS allows using complex data types as properties. The semantic information allows not just to identify the values but also their meaning. This is achieved by providing linking information to an ontology. This ontology can describe the functionality and context of a service and provide the possibility to reason or infer interesting data points instead of using specific implementations that have to be known already at development time. Furthermore, relationships to other data points are added in the BaaS definition. This allows to check whether all requirements for the use of a data point are met in a system. To have a brief example one can imagine a data point representing a record of temperature values in a room, where the data is a series of time stamps and the respective measured value. The meta data gives the unit of measurement as well as measurement precision, the location of the sensor and others. The relationship might be the connection to the data point that represents the sensor where the measurements has been taken from. Finally the semantic description points to an ontology concept that explains this data point on a conceptual level by a web link.

B. BaaS Service

Data points are the description of the functionality, its relationships, and meaning. The implementation of a data point is a BaaS service. Thus, the BaaS service adds details for different implementation aspects, so called adaptors. There are adaptors for, e.g., communication, parallel computation, runtime management, security, and authentication. In contrast to the data point, the BaaS service is technology dependent as it imposes, e.g., the use of a certain communication technology. However, the service implementation still remains independent of the programming language. The technology to use for the individual adaptors is not fixed by the reference architecture. Hence, it is possible to choose the best suitable. There are only some limitations on the choice, e.g., the communication protocol needs to support patterns like publish/subscribe and discovery of other communication partners. This approach is also the reason why this is a reference architecture: it defines guidelines to make the system consistent but it is not as specific as an architecture would be.

C. BaaS Container and Device

BaaS services can be hosted on containers. Containers provide the runtime environment and lifecycle management for the services. The lifecycle management is responsible for installing, starting, stopping, monitoring, and updating services. What exactly is contained in the runtime environment is, again, up to the concrete implementation. It can contain external libraries, like a cryptography library, or complete operating systems including all dependencies as in the case of a docker container [3]. This choice highly depends on the dependencies in a concrete architecture. The container should be selected to ease the flexible deployment of services independent of the operating system and underlying hardware. Thus, it is possible to run it on a controller, e.g., of an air-conditioner, but also on a dedicated server or even somewhere in the internet.

If a container runs on an embedded system, like in the air-conditioner, the container is called BaaS device. The BaaS device is a specialized form of a container which also provides some semantic information about the physical device itself.

As in small systems a lifecycle management may not be required and the required libraries are guaranteed to be available, the container may be omitted to reduce the hardware requirements. However, this limits the ability to update and monitor these systems. An alternative approach is to implement a container/device with a subset of features.

D. BaaS Registry

The last component to be discussed is the BaaS registry. The registry keeps track of all available BaaS devices and BaaS services in the network. Like the other concepts, the registry is not strictly specified. Therefore, it is possible to use a centralized or distributed registry depending on the needs of individual implementations.

E. The BaaS Approach

Next to the structural elements of the domain model, the reference architecture defines the methodology to create all required artifacts of a BaaS instance. In a first step, the data points will be described by domain experts. The idea behind is that services and functions in the domain of BAS will be very similar in many applications. Therefore, the description of data points can be shared and reused or specialized through inheritance and extension. There are a multitude of possible ways to share data point definitions, e.g., within a company, through a public crowd sourced platform or by standardization committees. As domain experts are not necessarily computer experts, a tool support will be provided to hide technical details. Until this point, the description is independent of the technology to be used in installations. The next step is to specify the services that represent a data point in a specific technology. At this point, also decisions about the adaptors of a service need to be made. While a communication interface is always required, management, security, and parallelism adaptors can be selected or deselected depending on the requirements of the system or the capabilities of the hardware. The technologies to be used are not defined by the reference architecture, so it is possible to choose the best suitable. After defining the adaptors and technologies, code generators can be used to create service skeletons. For this purpose, the interface and communication behavior can be generated from the data point definition and the technology definition. The parallelism can also be generated, e.g., thread pools can be used to handle incoming requests. The management capabilities are also quite static across different services. Only the business logic itself (e.g. reading a physical sensor, computation of a value) needs to be implemented manually. As already mentioned, the technologies to be used are not defined in the reference architecture. In the platform implementation that is developed in the project just covers one specific set of technologies (Java, CoAP, oAuth, Akka, OSGi, ...) If one wants to use other technologies, libraries or frameworks the generator templates needs to be manually implemented once for each technology. After generating and implementing the service, it can be deployed on a BaaS container or device. More details on the reference architecture are shown in [4].

III. State of the Art

As already mentioned, the BaaS approach is not limited to the usage of a specific communication protocol or a specific solution for the BaaS container. In the following, we will highlight communication technologies used in the factory automation domain and the building automation domain as well as two communication technologies of the internet of things. Those protocols have been chosen to meet the requirements of the BaaS reference architecture. Additionally, there is a section on OSGi which will be used as container by the reference implementation of BaaS.

A. OPC-UA

A protocol that is already addressing the problem of heterogeneous devices and therefore is worth noticing, is the "Open Platform Communications - Unified Architecture" also known as OPC-UA [5]. It is widely used in the factory automation and is scalable and platform independent [6]. OPC-UA uses a layered approach with different communication properties depending on the performance and application of the systems. In general, a service oriented architecture is used. For the communication on the management layer, for devices with sufficient computing power and energy supply, data is encoded in XML and carried by SOAP over HTTP. On smaller systems, on the monitoring and automation layer, XML is replaced by a proprietary representation called UA-binary. On even smaller devices used on the plant floor the HTTP/SOAP transport mechanism is replaced by a proprietary modification of TCP called UA-TCP. The approach requires additional efforts to send messages between different layers. This is suitable in the factory automation domain because of the common real-time requirements on the plant floor layer. However, the real-time capabilities are not of significant interest in the building automation domain. Thus, this approach is not helpful in our context as we would need to introduce proxies or gateways.

B. BACnet

In contrast to OPC-UA, Building Automation and Control Networks (BACnet) [7] is a protocol specialized for the building automation domain. It separates the modelling of BAS data, which follows the resource-oriented paradigm, from the technology used for transport of datagrams to exchange BAS data and commands. Due to this design decision, it supports the integration of heterogeneous transport technologies such as Ethernet, ARCNET, LonTalk or even transport over twisted pair wiring using EIA-485. With the addendum "a" of BACnet-1995 [8], the Internet Protocol made its way into the list of supported protocols. BACnet specifies a strict data model, where the main entity is a BACnet device which contains a number of BACnet objects. Each BACnet object contains a number of BACnet properties. There are standardized objects with predefined mandatory and optional properties. Also, a number of properties are already defined in the standard. Due to the lack of auto-configuration in BACnet, all participating devices need to be known at engineering time. Adding a device requires to go back into the engineering phase. This makes the approach inflexible regarding to reconfiguration or extension and results in higher engineering effort. Additionally, BACnet strongly relies on known structures even if it is allowed to add custom objects and properties. This makes it inconvenient to add functions originating from typical web technologies (e.g. weather-forecast) or representing functionality that is not fitting into the categorization of device object and property.

More flexible protocols suitable for small embedded devices that originate from the machine to machine communication on a less domain specific level are presented in the following subsections.

C. DPWS

The devices profile for web services (DPWS) is a subset of WS-* standards initially selected to cover many common problems in the domain of embedded devices in office environments. Nevertheless, it can also be used in other domains as there are no assumptions made regarding the structure of data or special interests like real-time. As the name already implies DPWS is following the service oriented architecture approach. DPWS is using the following WS-* subset:

- WS-Addressing for an addressing independent of underlying technologies
- WS-Discovery for the discovery of devices and services
- WS-Evening for an event driven communication pattern
- WS-MetadataExchange to retrieve metadata e.g. WSDL documents
- WS-Security for encryption and signatures
- WS-Policy to specify requirements for e.g. service version or security features

For the message exchange SOAP over HTTP and XML are used by default. Nevertheless, the applicability of DPWS in small embedded systems has been shown in [9] by replacing SOAP over HTTP by SOAP over CoAP and XML by the efficient XML interchange format (EXI). As CoAP itself is meeting the requirements of BaaS, it could be used also without DPWS on top.

D. CoAP

The Constrained Application Protocol (CoAP) was designed by the IETF CoRE (Constrained RESTful Environments) Working Group to bring the benefits of the Representational State Transfer (REST) to the domain of resource constrained devices. In contrast to other RESTful protocols like HTTP, CoAP focuses on the key features for machine-to-machine (M2M) communication. These key features include among others the discovery of devices and their services, a publish/subscribe mechanism and group communication capabilities [10]. For the communication CoAP uses the client-server principle, where a client sends a request to a server to either invoke a service on, retrieve data from, or upload data to the server. All services and data objects are represented as resources. Every resource is identified through a unified resource identifier (URI) and supports the four basic operations GET, PUT, POST and DELETE. In order to reduce the message size and parsing complexity to better fit the needs of resource constrained devices, it relies on a non-human-readable binary message encoding. The small message sizes and complexity combined with the provided M2M communication features make CoAP the perfect candidate for low power and low cost M2M applications. Although CoAP is a comparatively young protocol, a multitude of implementations in a variety of programming languages have already emerged. Table I gives an overview of the most important ones, but there are many more available.

TABLE I. List of significant CoAP implementations.

Name	Conformity	Advanced Features
Java Implementations		
Californium	RFC7252	Block Transfer, Observe, DTLS
jCoAP	RFC7252	Block Transfer, Observe
nCoAP	RFC7252	Block Transfer, Observe
C Implementations		
Erbium	RFC7252	Block Transfer, Observe
libcoap	RFC7252	Block Transfer, Observe

As aforementioned, building automation systems comprise of a heterogeneous set of devices ranging from high performance servers to low cost sensor nodes. Accordingly, the implementation used on a device may be chosen in regard of the device class. While Californium is a very feature rich CoAP implementation ideally suited for larger and more powerful server systems, jCoAP is more lightweight and better suited for middle class devices like RaspberryPi and the Intel Galileo Board [11] [12]. However, small sensor nodes usually have far less computational power and cannot cope with the computational overhead posed by a java virtual machine. Erbium is perfectly suited for heavily resource constrained devices like sensor nodes. It is very light weight and already integrated into the Contiki operating system for embedded devices [13]. In [14] Kovatsch et al. have shown an example application, where a Tmote Sky sensor node with 48 kB of program flash and 10 kB of RAM can easily host and process up to five non-trivial services.

As CoAP is very promising, the evaluation in the next section will use it as an example communication protocol for BaaS services. Moreover, it was agreed in the project to use OSGi for the implementation of containers. Therefore, it will be explained in the following subsection.

E. OSGi

The OSGi Service Platform is a dynamic module system for Java and comprises two important aspects that are not covered by native Java applications: modularity and dependency management.

In general, we can distinguish between physical and logical modularity. Physical modularity refers to how code is packaged for deployment. Logical modularity refers to a form of code visibility and defines a logical boundary within the application code. It impacts code visibility in a fashion similar to access modifiers. Physical and logical modularity are two distinct concerns, this means that logical modularity is possible without physical modularity and that multiple logical modules can be packaged into a single physical one.

The OSGi service platform follows the principles of component-based software development. Bundles are software components that provide and consume services. Their lifecycle is fully managed, so that bundles can be installed, updated and removed at runtime. A bundle is an ordinary Java Archive (JAR) with some extra headers and metadata in its manifest file. By enforcing a higher level granular structure on Java application code, bundles strongly encourage good software engineering practice. They are self-containing deployment units, comprising classes and resources required for execution. Bundles define their external dependencies via package imports and their external API visible to other bundles via package exports. By enforcing this dependency constraint, OSGi makes it abundantly clear what dependencies are needed for a given bundle to run, and also transparent as to which bundles can supply those dependencies. Especially in a high dynamic environment comprising bundles from different vendors, versioning becomes a serious concern. Without versioning it becomes nearly impossible for software components to evolve over time. In this context, we can distinguish between internal code modifications that are transparent for the user and are regarded as bugfixes or improvements of some non-functional quality parameters and those ones that directly impact the consumer implementation code. Things get even more complicated when components provide new functionality that breaks the existing behavior. This scenario requires that module changes are accompanied by a version change. OSGi recommends a scheme called semantic versioning.

Interactions between bundles are carried out on top of services. They provide a flexible and powerful way for sharing object instances transparently without the need to expose any internal implementation details to external consumers. For this purpose, a service provider defines a service interface as a contract between service provider and service consumer. In combination with semantic versioning, this feature allows developing high dynamic modular and dependable software applications. By hiding the service implementation details and promoting truly decoupled modules, OSGi services effectively enable a single-JVM service-oriented architecture. Services are highly dynamic and can "come and go" at every time during runtime. Services are registered by a bundle using one or more class names that mark the API of the service, and the service object itself. Optional properties provide extra information about the service and are used by clients to filter which services get returned when they are looking for one. Services can be looked up using a simple API. OSGi allows services to be accessed declaratively and injected as a dependency at runtime. For this purpose, OSGi provides a powerful extension called Declarative Services (DS) that introduces a service component model for publishing, finding and binding OSGi services. The component's lifecycle can be bound to the existence of specific services and configuration settings. A service component gets activated only when all of its preconditions are fulfilled. Combined with lazy instantiation, this approach provides developers with a very flexible and lightweight approach for service provisioning within high dynamic environments. It shifts the time for activation to this point in time, when the service gets actually used for the first time.

There are several frameworks which implement the OSGi specification. They are provided by commercial or open source projects. An overview over the most important implementations is given in Table II. Apache Felix and Eclipse Equinox are the two that are mostly applied. Some might additionally require Java SE [1] or Java ME CDC [2].

[1] J2SE requires 32-bit CPU, 128 MB RAM and 124 MB ROM

[2] J2ME CDC requires 32-bit CPU, 2 MB RAM and 2.5 MB ROM

TABLE II. OSGi Open Source Framework Implementations

Name	Spec	Platform	Size (kB)
Eclipse Equinox	R6	J2SE	1234
Apache Felix	R6	J2SE	1001
Knopflerfish	R5	J2SE	325
Concierge	R5	J2ME CDC	250
nOSGi	R4	C++	233

Beside of the fact that Apache Karaf is a small OSGi based runtime which provides a lightweight container for application deployment. Apache Karaf can be configured to use each of them and adds additional functionality like hot deployment, dynamic provisioning and configuration, advanced logging, security mechanisms, and native OS integration.

IV. Flexibility of Concept

In this chapter, the implementation of the BaaS reference architecture is sketched on the different kinds of computing nodes covering different tasks of BAS.

Nodes with sufficient computing power, RAM, and ROM can obviously be used in the BaaS approach. These could be desktop PC's, in-house server or server systems on the internet. They may be used to take over responsibility for services providing tasks of high complexity. Examples would be optimizations, learning algorithms, reasoners for semantic analysis, or gateways to other (legacy) technologies. For these complex tasks, the BaaS services can make use of the parallelism option. This feature will be enabled at development time. In the following step, the required skeletons are generated and the software developer can make use of threads in his implementation. Nevertheless, these nodes can also serve as host for a number of services of lower complexity to reduce the load on smaller eventually energy constrained devices. In this case, the parallelism feature can be used to handle a high number of requests. For the communication of BaaS services, any CoAP implementation can be used. For the management of the services on the node, the BaaS container is used. It can help managing dependencies of services, starting and stopping them as required and allows the services to be updated. This enables to plug in services at runtime without additional engineering effort. The availability of an update mechanism is a novum in BAS as current solutions are not designed to be updated after they are put into operation. As technical solution, any OSGi implementation can be used. For an increased maintainability, Apache Karaf enables to manage several containers in the same way.

Nodes with less computing power might have some limitations on the supported features. As an example for this class of device, we use the automation station DDC4000 of "Kieback & Peter"; a building automation provider and project partner. This node has a MPC855T 32-Bit CPU with 80 MHz, 96 MB RAM and 256 MB flash disk. Common tasks of such nodes are the automation and monitoring of several hundreds of (BACnet) data points. With this specification, only few implementations of CoAP and OSGi can be used. Especially implementations using J2SE cannot be applied. Nevertheless, the OSGi implementations Concierge (2 MB RAM and 2.85 MB ROM) or nOSGi (233 kB ROM) can be used. The same applies for the CoAP implementations libcoap and Erbium. Therefore, these nodes are also capable to do lifecycle management, runtime deployment and updating mechanisms of BaaS services and enable communication. Multi-threading might be used on this kind of system to handle concurrent requests but eventually with a limited number of threads. As the embedded Linux used is not known we might assume 16 MB ROM and 32MB RAM usage for this. This would mean that about 62 MB RAM and 237 MB ROM are still left for the service implementations. The number of services that can be hosted on a node of this type has to be elaborated in a later stage of the project as it is not clear what the resulting services will require for their static descriptions, code, and dynamic memory.

For small devices, like sensors or actors, the approach is also suitable. A representative of this device class would be the Tmote Sky sensor MSP430 16-Bit CPU with 3.9 MHz, 48 kB ROM and 10 kB RAM. Within this specification, it is not possible to use OSGi as the smallest implementation already requires 233 kB ROM. Nevertheless, the container does not require using OSGi and we can drop some features. In this case, the lifecycle management is simplified. The service(s) will be executed at start-up and exit on shutdown. The implementation of the sensor needs to make sure that all runtime requirements are met by default for all BaaS Services hosted on such a device. As we know at development/engineering time which services are hosted on the device and what runtime requirements they have, these requirements can be checked and fulfilled before runtime. Altogether, this does not increase the footprint. It has already been proven by Kovatsch et al. that the implementation of an operating system including a full IP and CoAP stack for such devices is possible [14]. It has been shown that the mentioned Tmote sensor is capable of providing 5 non trivial resources over CoAP which is enough to represent at least one BaaS data point. What's missing on this device, is the support of parallelism inside the individual services. However, we consider this irrelevant under the assumption that the sensor is not capable of handling many requests. To still provide the sensing results to many consumers, the observe mechanism of CoAP can be used to send the latest readings in a publish/subscribe manner keeping simultaneous request small.

V. Conclusion

This paper presented the BaaS reference architecture regarding the main elements required at runtime. Several technologies that can be used for an implementation of the reference architecture were described and discussed in Sec. III. With the chosen technologies, CoAP and OSGi, the applicability of the BaaS reference architecture to different devices was shown. The devices are selected according to devices used in real BAS or are intended to be used in future BaaS installations. In all examples mentioned above, the communication is realized using CoAP on top of the UDP/IP protocol stack. Additionally, all devices use the same data model and exchange format. In this way, all devices are integrated horizontally and vertically without using gateways or proxies. Open points are the implementation and evaluation containing implementations of BaaS services. This requires some advancement in the project, especially regarding data point descriptions of existing devices.

Acknowledgment

This work has been achieved in the European ITEA project "Building as a Service" (BaaS) and has been funded by the German Federal Ministry of Education and Research (BMBF) under reference numbers 01IS13019H and 01IS13019B. We want to thank all partners in the BaaS project for the stimulating discussions and their contributions to the project. All project partners can be found on http://baas-itea2.eu.

References

[1] "Baas project home." [Online]. Available: http://baas-itea2.eu/cms/

[2] B. Butzin, F. Golatowski, C. Niedermeier, N. Vicari, and E. Wuchner, "A model based development approach for building automation systems," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Barcelona, Spain, Sept 2014, pp. 1–6.

[3] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *Cloud Computing, IEEE*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[4] N. Vicari, E. Wuchner, A. Bröring, and C. Niedermeier, "Engineering and operation made easy - a semantics and service oriented approach to building automation," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2015, pp. 1–8.

[5] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. [Online]. Available: http://link.springer.com/10.1007/978-3-540-68899-0

[6] A. Frejborg, M. Ojala, L. Haapanen, O. Palonen, and J. Aro, "Opc ua connects your systems," in *Finnish Society for Automation, Biannual Seminar no. XX*, 2013, p. 6.

[7] *ANSI/ASHRAE Standard 135-2012*, ANSI Std.

[8] American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc., "ANSI/ASHRAE Addendum a to ANSI/ASHRAE Standard 135-1995," 1995. [Online]. Available: http://www.bacnet.org/Addenda/Add-1995-135a.pdf

[9] G. Moritz, F. Golatowski, C. Lerche, and D. Timmermann, "Beyond 6lowpan: Web services in wireless sensor networks," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 4, pp. 1795–1805, Nov 2013.

[10] Z. Shelby, K. Hartke, and C. Bormann, *RFC 7252: The Constrained Application Protocol*, online, IETF Std.

[11] B. Konieczek, M. Rethfeldt, F. Golatowski, and D. Timmermann, "Real-time communication for the internet of things using jcoap," in *18th IEEE Symposium on Real-Time Computing (ISORC)*, Auckland, New Zealand, April 2015.

[12] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[13] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on.* IEEE, 2004, pp. 455–462.

[14] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power coap for contiki," in *8th IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*, Valencia, Italy, October 2011, pp. 855–860.