Received February 21, 2018, accepted March 25, 2018, date of publication April 10, 2018, date of current version May 16, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2825295

IoT-DDL–Device Description Language for the ‘‘T’’ in IoT

AHMED E. KHALED[1], **ABDELSALAM HELAL**[2], **(Fellow, IEEE), WYATT LINDQUIST**[2], **AND CHOONHWA LEE**[3]
[1]Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL 32611, USA
[2]School of Computing and Communication, Lancaster University, Lancaster LA1 4WA, U.K.
[3]Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, South Korea

Corresponding author: Ahmed E. Khaled (aeeldin@ufl.edu)

This work was supported in part by the Philips, in part by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Science, and in part by ICT under Grant 2017R1A2B4010395.

ABSTRACT We argue that the success of the Internet of Things (IoT) vision will greatly depend on how its main ingredient—the ‘‘thing’’—is architected and prepared to engage. The IoT's fragmented and wide-varying nature introduces the need for additional effort to homogenize these things so they may blend together with the surrounding space to create opportunities for powerful and unprecedented IoT applications. We introduce the IoT Device Description Language (IoT-DDL), a machine- and human-readable descriptive language for things, seeking to achieve such integration and homogenization. IoT-DDL explicitly tools things to self-discover and securely share their own capabilities, entities, and services, including the various cloud-based accessories that may be attached to them. We also present the Atlas thing architecture—a lightweight architecture for things that fully exploits IoT-DDL and its specifications. Our architecture provides new OS layers, services, and capabilities we believe a thing must have in order to be prepared to engage in IoT scenarios and applications. The architecture and IoT-DDL enable things to generate their offered services and self-formulate APIs for such services, on the fly, at power-on or whenever a thing description changes. The architecture takes advantage of widely used device management, micro-services, security, and communication standards and protocols. We present details of IoT-DDL and corresponding parts of the thing architecture. We demonstrate some features of IoT-DDL and the architecture through proof-of-concept implementations. Finally, we present a benchmarking study to measure and assess time performance and energy consumption characteristics of our architecture and IoT-DDL on real hardware platforms.

INDEX TERMS Internet of Things architecture, thing description, microservices, OMA, IPSO, CoAP, MQTT.

I. INTRODUCTION

The spaces around us are getting full of things! Things are the basic building blocks and the main ingredient of the emerging and revolutionary Internet of Things (IoT) technology [1], [2]. IoT actively brings more informative and interactive flavors to our lives, enabled by the evolution of low-power wireless technologies and embedded computing and intelligence. Empowered by the fact that now almost all devices are Internet-connected [3], [4], IoT is transforming the standard view of the Web as a set of digital documents and links into a fully integrated Internet that includes physical devices as well as cyber elements. This new Web, combining the cyber and physical worlds, creates a new ecosystem with new programmability opportunities through the various interactions and interconnections of the two realms [5], [6]. While the IoT is perceived as a generic and a generalized concept, in practice, it is not. Its various specializations and full taxonomy are yet to shape up and be fully learnt. A simple categorization of the IoT that captures its scale and applicable domains de jour is illustrated in Fig. 1.

According to the scale and place of deployment we classify IoT into personal IoT (e.g., in smart homes or connected cars), industrial IoT (e.g., a smart factory floor or a physical plant), and at-scale IoT (e.g., a smart city deployment). Classifying IoT in this simple manner at this stage in the evolution of IoT is surely missing many important parameters. However, the classification helps us state the focus and applicability of our work in this paper, which is on the

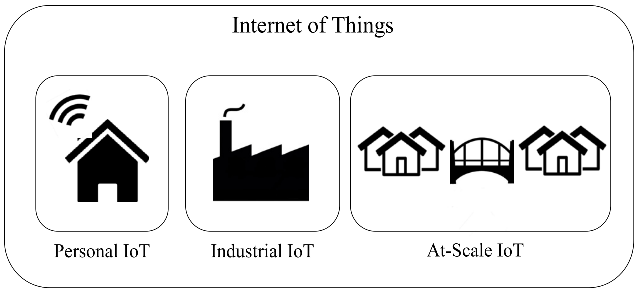


FIGURE 1. A simple IoT domain classification.

personal IoT, where the set of things is located in a bounded personal space.

The highly fragmented nature of IoT and the wide heterogeneity in types, capabilities, and technologies raise thing integration as a significant challenge. Integrating such a wide spectrum of things in the ecosystem requires considerable effort and limits programming opportunities for smart spaces. Such challenges introduce further questions: How can space users (e.g., developer, vendor, and space owner) manage and configure such wide heterogeneity? How can such fragmented things securely interoperate and interact not only with cloud platforms and space users but also with other things in the space? These challenges introduce, as a first step, a requirement for a uniform way of describing things in smart spaces in terms of what a particular thing is (e.g., its components), what it does (e.g., its offered services) and how it communicates (e.g., what it can speak or which protocols it understands). Such a description paves the way to solve thing integration, configuration, and management challenges, while also enabling interactions. Such an approach requires an architecture for things in IoT, as a second step, that fully exploits thing description specifications (the first step) and supports the promising vision of IoT.

Different approaches in the literature target these challenges by describing things in a space in terms of metadata, resources, and access methods. These approaches link the access of things to a central point (e.g., cloud platform, or edge), through which space users can access resources and collect data. Such a restricted paradigm ignores the distributed nature of IoT, which should also allow things to communicate with one another in a space, forming thing-to-thing as well as thing-to-cloud or thing-to-edge communication paradigms. To enable such communication paradigms, a thing should be fueled by an additional set of attributes and properties that describe its various aspects as well as how it can engage in smart spaces. This set of attributes should not only describe the thing and its components, but also how it can be managed and configured, the different communication languages it supports, and eventually the important IoT semantics of how a thing can be used by other things or utilized within an IoT application. We argue that a description language for things that covers these aspects can enable various secure, meaningful interactions between things and thing mates. Thing mates include cloud platforms, edges, space users, and other things.

In this paper, we present the IoT Device Description Language (IoT-DDL), a machine- and human-readable XML-based descriptive language for things in smart spaces. IoT-DDL explicitly tools a thing to self-discover its own onboard capabilities, resources, entities, and services, as well as cloud-based thing accessories. A thing's resources are the components that describe the basic services it needs to be part of the IoT ecosystem (e.g., network module or memory unit). A thing's entities are the physical devices, software functions, and hybrid (virtual) devices that can be attached to, built in, or embedded inside the thing. Each entity provides a set of services to thing mates through a set of well-defined interfaces. In addition, external accessories, which are entities external to the thing that could be added to augment the thing capabilities over time, can provide a cloud-based expansion of the thing (e.g., a database, drivers, convertors, or specific add-on interfaces) through named *attachments*. Enabling the thing to self-discover what it is, what it does, and how it communicates can empower meaningful interactions and interconnections that support the distributed nature of IoT. Such enablement will be required before any useful programming models can be defined within the IoT ecosystem. Atlas IoT-DDL builder is a web service tool that allows a thing's creator (e.g., the original equipment manufacturer (OEM)) or owner to create, update, or upload an IoT-DDL to a thing. The OEM of a thing could be the source of the IoT-DDL; a developer who utilizes space things' services and resources might also be the source. Such flexibility facilitates further adoption of IoT-DDL with changes, and supports thing innovation, in which makers or hobbyists may be assembling new things not established by an OEM. We developed an initial version of the web tool [46] that enables space users to develop an IoT-DDL that reflects the thing's metadata and attachments, as well as its inner entities, resources, and services; as will be discussed later.

The IoT-DDL is proposed within the lightweight Atlas thing architecture, which fully utilizes the specifications of IoT-DDL. The Atlas thing architecture takes advantage of a thing's OS services to provide new layers and functionalities that introduce the novel capabilities a thing needs to engage in ad-hoc interactions and interconnections, as well as IoT scenarios and applications. The architecture enables a thing to self-discover its resources, attachments, components, and services from the uploaded IoT-DDL. The architecture extends the Apache licensed Micro Services project [51] to enable the thing to generate the services it wishes to offer to the smart space. Based on the inputs, outputs, and platform-agnostic actions specified in the IoT-DDL, the architecture dynamically generates and manages a bundle (single executable microservice) that fulfills a specified thing service at the runtime of the thing. The thing can then formulate APIs of the services it offers and enable service-oriented meaningful interactions to take place between the thing and its thing mates.

This paper focuses on IoT-DDL concepts, requirements, and specifications. It also focuses on the details of parts of the Atlas thing architecture that implement and take advantage of the IoT-DDL. The paper is organized as follows. Section II highlights related work, followed by a description of a proposed structure of things and the IoT-DDL specifications in section III. The overall Atlas thing architecture is described in section IV with focus on the architecture layers that implement IoT-DDL. In section V we present our implementation and a proof-of-concept, and in section VI we present a benchmarking study in which we measure and assess memory footprint, latency and energy characteristics of the IoT-DDL and Atlas thing architecture on real hardware platforms. Finally, a discussion and a conclusion are presented in section VII.

II. RELATED WORK

Although there is not much in the literature on explicit architectures for things, quite a bit of work exists on device descriptions. The Device Description Language (DDL) is a machine- and human-readable XML-based device description approach developed by the Mobile and Pervasive Computing Lab at the University of Florida [16], [17]. DDL is a schema for the seamless integration of devices into a smart space, service registration, and discovery. DDL describes the metadata of the physical device and how to access the offered services. DDL was used to develop the Cloud-Edge-Beneath (CEB) architecture [14], [15], [29]. CEB opens access links to devices from the cloud through the use of the Atlas sensor platform and middleware. Atlas middleware, hosted by the Edge (e.g., standalone server), uses the Open Services Gateway initiative (OSGi) for service discovery and configuration of Atlas sensor platforms. The middleware, when contacted by an Atlas sensor platform, retrieves information from the DDL descriptor and creates a Java bundle for that sensor. Sensors are abstracted into sensor service interfaces in the cloud through interlayer collaboration between the Atlas middleware bundles at the Cloud, the Edge, and beneath.

To enable thing-to-thing, thing-to-cloud and thing-to-edge communication paradigms, IoT-DDL extends the descriptive power of Atlas DDL to address a thing's self-discovery. IoT-DDL, as a self-description tool uploaded to a thing, explicitly powers the thing to discover its inner resources, characteristics, services, communication languages, and cloud-based attachments. IoT-DDL and the Atlas thing architecture enable the thing to identify itself to thing mates and formulate APIs for the offered services. The IoT-DDL enables the seamless integration of things into the ecosystem, and equips the thing with a set of attributes that enable thing management and configuration with minimal human intervention.

The Web of Things (WoT) framework by the World Wide Web Consortium (W3C) [32], [33] is an active research field that explores access to and handling of things' digital representations through a set of web services. These services are based on event-condition-action rules that involve these virtual representations as proxies for physical entities. Such objects are modeled in terms of metadata, events, and actions, along with the RESTful protocol. Servers provide an interface for instantiating and registering such proxies for the things along with their descriptions. A client script interacts with these proxies exported by the server, where applications can register callbacks for events. Käbisch and Anicic [49] utilize Thing Description (TD) to describe the different things in the WoT, in terms of their metadata, how to access them, and their different events and corresponding actions. The TD relies on the Resource Description Framework (RDF) [50] as an underlying data model that can be extended to involve domain specific information.

The Constrained RESTful Environments (CoRE) [47] realizes the Representational State Transfer (REST) architecture for the discovery of resources hosted by constrained nodes to build M2M applications. CoRE extends the universal resource identifiers (URI) for such resources with a set of attributes and descriptions of relations between such resources. A client, for his application, utilizes such resource discovery architecture with the appropriate resource description, along with possible application-specific attributes. Datta and Bonnet [8] and Datta *et al.* [48] highlights an evolution in Thing Description (TD) from the CoRE Link Format to describe physical things in the IoT. TD represents the different sensors and actuators in terms of events and actions. The authors proposed a thing management framework that resides in an M2M gateway.

Google's Weave [18], [19] is a communication platform that allows smartphones and cloud services to interact with things through mobile devices and the Web. Weave supports cloud services such as device discovery, provisioning, state subscription, remote access, and push notifications. Weave introduces two main ideas of device description schema: 1) *trait*, which describes device functions, commands, and state definitions, and 2) *component*, which describes the relationships between traits. Weave is provided in conjunction with Brillo, Google's new Android-based OS for embedded development. Brillo offers device management, a hardware abstraction layer, and a development kit. Several Brillo-compatible boards can be accessed and managed through a Linux developer machine. IoT applications can be developed directly over Brillo, with development taking place on the developer machine and the resulting image is flashed on the target hardware.

A related approach to IoT-DDL is Amazon Web Services IoT (AWS-IoT) [20], [21], a cloud-based platform that provides bidirectional communication between the AWS cloud platform and things in a space (sensors, actuators, and embedded devices). AWS's primary focus is collecting and analyzing data reported by multiple devices. A thing-registry module stores and organizes thing-related information and resources, while users can associate up to three custom attributes with each thing. On the other side, each thing has a thing-shadow that stores thing-state and metadata in response to application requests.

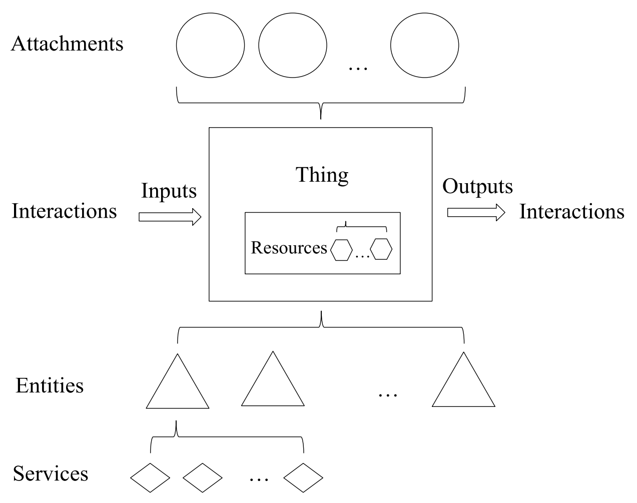


FIGURE 2. **Thing structure.**

The aforementioned approaches link the access of things to a central point (e.g., cloud platforms or edge) where space users can access resources and collect data. Such a restricted paradigm ignores the distributed nature of IoT, which requires things to communicate with other things as well as with cloud platforms and edge. To enable secure ad-hoc interactions between a thing and its thing mates in a space, a thing should be fueled by an additional set of attributes and properties. In the absence of a device description language that supports basic thing requirements for a smart space, significant effort is required to interact with and manage the wide heterogeneity of things. At the same time, the thing description should be part of the thing itself to facilitate a thing's smooth migration from one space to another. In the next section, as a first step toward describing a thing, we define the essential requirements that must be met for a thing to be part of the IoT ecosystem. We also introduce a structural definition for things.

III. IoT-DDL SPECIFICATIONS

The first step in describing a thing in a smart space is identifying the different parts that make up its structure. The thing, as illustrated in Fig. 2, is composed of a set of resources, entities, and attachments and engages with thing mates through some interactions. Resources are the components that shape the OS services a thing needs to be part of the IoT (e.g., network module, memory unit, etc). Each resource is shaped through a set of properties that configures such operating services. Moreover, thing entities are the physical devices, software functions, and hybrid devices that can be attached to, built in, or embedded inside the thing. Each entity provides a set of services to the smart space through a set of well-defined interfaces (APIs). Furthermore, a thing can have one or more external accessories or attachments. Thing attachment is a cloud-based expansion of the thing that provides further representations (e.g., thing virtualization) and services (e.g., log server, database, or dashboard) that are considered too heavyweight to be hosted on the thing or require additional resources that are not available on such constrained devices. Thing mates include cloud platforms, edges, humans (e.g., space users or developers), and other things.

A thing in a smart space engages with thing mates in the IoT ecosystem through a set of information- and action-based interactions. Information-based interactions (referred to in this paper as tweets) enable a thing to announce its identity, capabilities, and APIs to thing mates. A thing uses a tweet to describe what it is, what it does, and what it knows to the other thing mates. Action-based interactions include management commands, lifetime updates, and configurations from authorized parties as well as the applications that target the thing's capabilities and services.

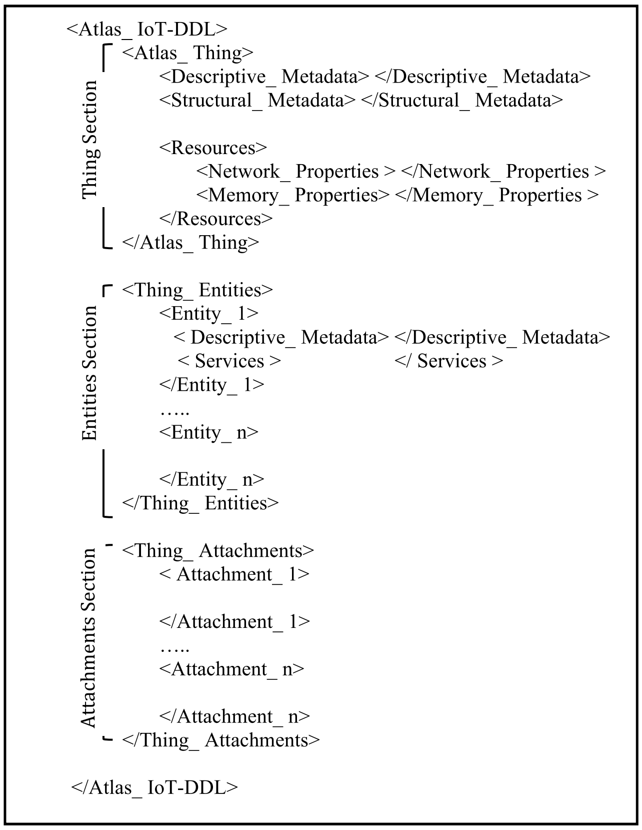
Consider a smartphone as an example of a thing in smart space. To be part of the smart space, a smartphone must be equipped with an internal memory and Wi-Fi network module. It also contains a set of sensors (e.g., proximity, accelerometer) as embedded entities that can offer services to the smart space. The phone can also be connected to a cloud attachment for lifetime OS updates provided by the vendor. The three requirements and the proposed structure for the thing shape the specifications required to describe things in smart spaces and highlight the design aspects of the architecture that can fully utilize such specifications.

Based on the thing structure outlined above, we now present our IoT-DDL specifications that describe the different parts and accessories of a thing through a set of attributes, parameters, and properties. IoT-DDL is based on Atlas DDL [16], which uses an XML-based schema to describe devices to facilitate their integration in a smart space. It has been used to develop the Atlas Cloud-Edge-Beneath (Atlas-CEB) architecture [14], which uses DLL to generate Java bundles that represent the devices and that can be deployed on an edge and/or cloud to connect back and interact with the devices the DDL describes. DDL is used to describe a single device (sensor, actuator, or hybrid) through the device's metadata, functions, and operations. Atlas IoT-DDL extends Atlas DDL to address additional thing requirements and to match the thing structure outlined above. The IoT-DDL structure is divided into the Atlas thing, thing entities, and thing attachments sections, as illustrated in Fig. 3. These three main sections are described below.

A. ATLAS THING SECTION

The *Atlas thing* section of IoT-DDL characterizes the thing as a whole, describing the different resources and components on board. It is further structured into the following subsections:

1) The *descriptive metadata* section holds the thing's identification, including its name, model, short description, type (e.g., software, hardware, hybrid), vendor, and owner.
2) The *structural metadata* section highlights the structure of IoT-DDL file as a short description of the thing's resources, entities, and attachments.



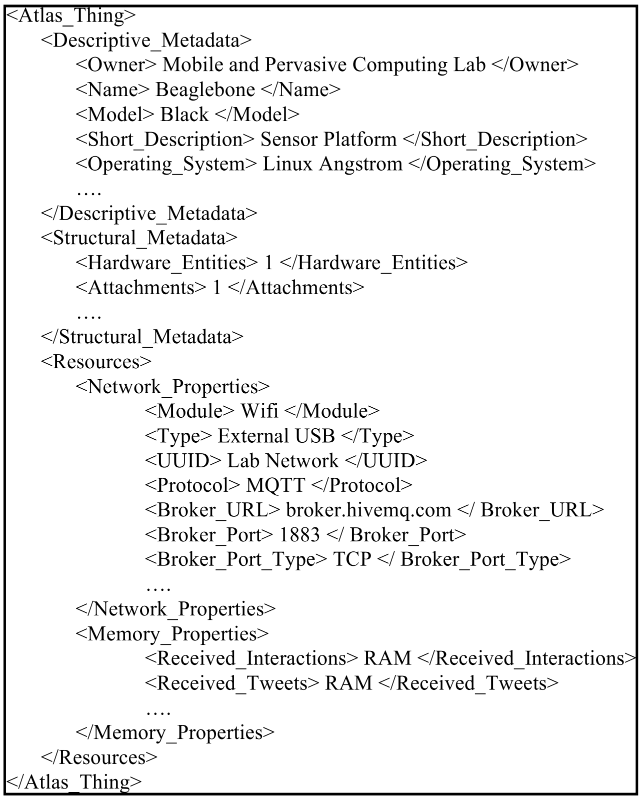
```
<Atlas_ IoT-DDL>
    <Atlas_ Thing>
        <Descriptive_ Metadata> </Descriptive_ Metadata>
        <Structural_ Metadata> </Structural_ Metadata>

        <Resources>
            <Network_ Properties > </Network_ Properties >
            <Memory_ Properties> </Memory_ Properties >
        </Resources>
    </Atlas_ Thing>

    <Thing_ Entities>
        <Entity_ 1>
          < Descriptive_ Metadata> </Descriptive_ Metadata>
           < Services >                 </ Services >
        </Entity_ 1>
        …..
        <Entity_ n>

        </Entity_ n>
    </Thing_ Entities>

    <Thing_ Attachments>
        < Attachment_ 1>

        </Attachment_ 1>
        …..
        <Attachment_ n>

        </Attachment_ n>
    </Thing_ Attachments>

</Atlas_ IoT-DDL>
```

FIGURE 3. IoT-DDL specifications.

3) The *resources* section holds the attributes and properties of the underlying OS services the thing needs to be part of the IoT. The Atlas thing section holds a separate section for each resource. In this paper, we support both network module and memory unit as the two main resources a thing needs to engage with a smart space.
   - The *network properties* subsection describes the network connection capabilities of a thing in terms of the mounted network module (e.g., Wi-Fi, Bluetooth), network access information, and preferences. The subsection covers the attributes and properties of the supported communication protocols (e.g., MQTT, CoAP).
   - The *memory properties* subsection highlights the various memory units available to process applications, generate APIs, and archive information in a specific format.

B. THING ENTITIES SECTION

The *thing entities* section describes the different types of entities (hardware, software, or hybrid) that can be embedded, built in, or connected to the thing. Each entity is detailed in terms of its descriptive information, the services and functions it offers, and the different types of interactions it can engage in. Each entity is further divided into components as follows:



```
<Atlas_Thing>
    <Descriptive_Metadata>
        <Owner> Mobile and Pervasive Computing Lab </Owner>
        <Name> Beaglebone </Name>
        <Model> Black </Model>
        <Short_Description> Sensor Platform </Short_Description>
        <Operating_System> Linux Angstrom </Operating_System>
        ….
    </Descriptive_Metadata>
    <Structural_Metadata>
        <Hardware_Entities> 1 </Hardware_Entities>
        <Attachments> 1 </Attachments>
        ….
    </Structural_Metadata>
    <Resources>
        <Network_Properties>
                <Module> Wifi </Module>
                <Type> External USB </Type>
                <UUID> Lab Network </UUID>
                <Protocol> MQTT </Protocol>
                <Broker_URL> broker.hivemq.com </ Broker_URL>
                <Broker_Port> 1883 </ Broker_Port>
                <Broker_Port_Type> TCP </ Broker_Port_Type>
                ….
        </Network_Properties>
        <Memory_Properties>
                <Received_Interactions> RAM </Received_Interactions>
                <Received_Tweets> RAM </Received_Tweets>
                ….
        </Memory_Properties>
    </Resources>
</Atlas_Thing>
```

FIGURE 4. Beaglebone Atlas thing section.

1) The *descriptive metadata* section holds an entity's identification, including name, model, short description, type (e.g., software, hardware, hybrid), category (built-in, embedded, attached), vendor, and owner.
2) The *service* section holds descriptive information about the different services offered by the entity in terms of functional descriptions, inputs, and outputs. Each service input or output is characterized in terms of a short description, data types, units, and the acceptable range of values.

C. THING ATTACHMENTS SECTION

The thing attachments section describes the different cloud-based expansions of the thing. Each attachment is described in terms of the type (e.g., data-log server, repository, device management server), the access information (e.g., URI, key), and the protocols used (e.g., REST).

D. EXAMPLE IoT-DDL

To illustrate IoT-DDL, we use the example of a coffee maker, which is part of the proof of concept presented in the implementation section (Section V). In this example, a coffee maker entity and a web-logging attachment are part of a Beaglebone Black Atlas thing. The overall IoT-DDL in this example consists of three parts: The Beaglebone Black Atlas thing description, the coffee maker description, and the web-logging attachment description, just as outlined in Fig. 3.

The Beaglebone Black Atlas thing section is shown in Fig. 4. The descriptive metadata section describes the thing

```
<Entity_1>
  <Descriptive_Metadata>
      < Name>Coffee Maker</ Name>
      < Category>Hardware</ Category>
      < Owner>Bosch</ Owner>
      < Type>Attached</ Type>
      < Description>Prepare Coffee</ Description>
        ….
  </Descriptive_Metadata>
  < Services>
      <Service_1>
        < Description>Turn On For Duration </ Description>
        < InputTypes>Integer </ InputTypes>
        < InputDescription>Duration in Minutes </ InputDescription>
        < InputRange> [0:59] </ InputRange>
        < OutputTypes>NULL</ OutputTypes>
        < FunctionMap>HighVolt_GPIO_Duration</ FunctionMap>
        ….
      </Service_1>
      <Service_2>
        < Description>Turn Off </ Description>
        < InputTypes>NULL </ InputTypes>
        < OutputTypes>NULL</ OutputTypes>
        < FunctionMap>LowVolt_GPIO</ FunctionMap>
        ….
      </Service_2>
  < /Services>
</ Entity_1>
```

FIGURE 5. Coffee maker entity section.

```
<Thing_Attachments>
      <Attachment_1>
            < Type>Log_Server </ Type>
            <Description> HTTP1.1 Server </ Description>
            <URI>192.168.0.4</ URI>
            <Port>8080</ Port>
            <REST_Method>PUT</ REST_Method >
            <Access_Type>Public</ Access_Type>
            <API_Key>NULL</ API_Key>
            <Message_Format>Plain</ Message_Format>
            <Update_Duration_Value>2</ Update_Duration_value>
            < Update_Duration_Unit>minute</ Update_Duration_Unit>
            <Backend_Technology>NodeJS</ Backend_Technology>
            ….
            ….
      </ Attachment_1>
</ Thing_Attachment>
```

FIGURE 6. Web-Log server thing attachments section.

in terms of the owner, name, and operating system. The structural metadata section summarizes information about thing's resources, entities and attachments. The resources section shows the overall properties of both network and memory resources of the underlying operating system. The Beaglebone Black thing utilizes the MQTT protocol to communicate with thing mates and to set up a TCP connection with an online MQTT broker as will be shown in section V. The broker domain name and port number are part of the Network Properties sub-section that describes the Beaglebone Black Atlas thing.

The coffee maker entity section is illustrated in Fig. 5. The coffee maker is described in the descriptive metadata section as a hardware device attached to the Atlas thing and offers two services: to turn the coffee maker on for a maximum amount of time, and to turn the maker off. Each of these services is described in a separate service section within the Services section of the entity in terms of function description, inputs and outputs. The Web-log service attachment section—as Fig. 6 shows—is a NodeJS-based server that allows the Atlas thing to send status updates to the server periodically (every two minutes in this example) through an HTTP "PUT" method.

As noted earlier, the IoT-DDL is proposed within the Atlas thing architecture, which takes advantage of a thing's OS services to provide new layers and functionalities that introduce novel and necessary capabilities. The Atlas architecture is briefly discussed in the next section. We do not attempt to fully present the details of the architecture, and keep our focus on device descriptive layers that reflect IoT-DDL specifications.

The difference between the original DDL [16], [17] and the proposed IoT-DDL can be summarized as follows: 1) The focus of the DDL is to generate a run-time representation of the thing's service on the edge or the cloud for service discovery, whereas the focus of the IoT-DDL is to generate a run-time representation of the thing on the thing itself to enable ad-hoc interactions and interconnections. 2) The DDL focuses on describing the services offered by the thing, while the IoT-DDL focuses on additional attributes to describe the thing in the smart space by the thing's entities, resources, services, attachments, and device management and communication protocols. 3) The focus of the DDL is to enable the thing-to-cloud communication paradigm, whereas the focus of the IoT-DDL is to enable both thing-to-thing and thing-to-cloud communication paradigms for the seamless building of IoT scenarios that involve different things.

IV. OVERVIEW OF THE ATLAS THING ARCHITECTURE

In this section, we present a brief overview of the Atlas thing architecture, which fully exploits the specifications and design principles of IoT-DDL. The Atlas thing architecture is designed to meet the requirements for a thing to be part of the IoT ecosystem. Its goals are to make the thing capable of 1) self-discovering its characteristics, resources, and entities through the uploaded IoT-DDL, and generating APIs for the available services; 2) opening a channel with a device management server for provisioning, management, and configuration purposes, and 3) enabling secure interactions with thing mates, The architecture also takes advantage of lightweight device management standard OMA-LwM2M [9]–[11], [13], object modeling standard IPSO [7], [12], IoT communication standards CoAP [34], [35] and MQTT [28], [30], and the AES [39] security standard to enable thing management and configuration with minimal human intervention, and to empower secure ad-hoc interactions between things and thing mates.

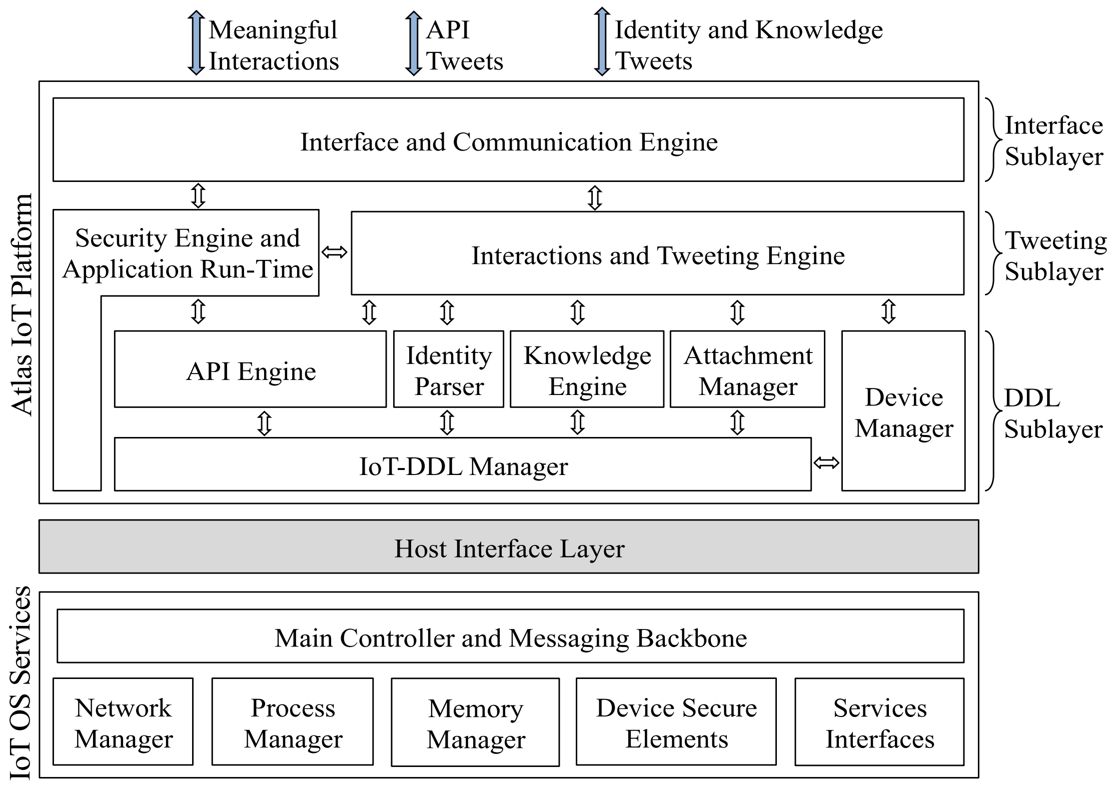


FIGURE 7. Atlas thing architecture.

The architecture can be developed into a set of software layers or firmware that can be flashed into the thing using the vendor's provided IDE or OS (e.g., C/C++ for Linux OS-based platforms such as Beaglebone and Raspberry Pi, Java/C++ for Android smartphones, or Arduino IDE for Arduino Things). The Atlas thing architecture, as illustrated in Fig. 7, consists of three main layers: Atlas IoT platform, host interface, and IoT OS services. IoT OS services are the basic functionalities provided by the thing's operating engine and represent a thing's resources according to the proposed thing structure. Such services enable the thing to be part of the IoT through its network module, memory units, I/O ports and interfaces (e.g., I/O, ADC, etc.), and its process manager. Services may also include hardware-based security if available, such as embedded secure elements [42], [43]. The process manager is responsible for offering services for command execution and threading for concurrency if supported by the IoT OS services.

The Atlas IoT platform represents the logical layer of the architecture that runs on heterogeneous things to provide new needed services not currently provided by embedded OS's. Such new services focus on descriptive and semantic aspects of things to better enable thing engagement, interaction, and programmability into an IoT. The host interface layer shields the platform and gives it the portability and interoperability features it needs. This layer also maintains the platform's lightweight nature by maximally relying on lower-level services provided by the underlying IoT OS. The host interface layer manages the internal interactions between the Atlas IoT platform and the set of services provided by the underlying OS. However, the host layer assumes the responsibility of providing any services that the underlying OS does not provide. An extreme case is when there is no underlying OS in a given platform (e.g., Arduino sensor platform); in this case, the host layer must implement all required services.

Next, we present the details of the DDL sublayer within the Atlas IoT Platform layer of the architecture that is responsible for mapping and managing the IoT-DDL specifications. We also briefly summarize the other sublayers of the architecture.

A. DDL SUBLAYER

The DDL sublayer allows a thing to discover its own resources, entities, and capabilities through the uploaded IoT-DDL configuration file. It is composed of the following modules:

- The *IoT-DDL manager* opens a gate to access the uploaded IoT-DDL configuration file, parses the various sections and subsections, and regulates access to the IoT-DDL from the other modules.
- The *identity parser* models the identification and descriptive information of the thing and its entities. This module interacts with the IoT-DDL manager to parse

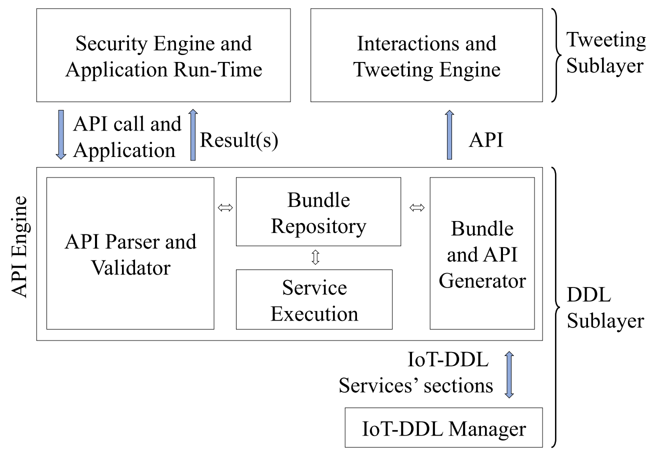
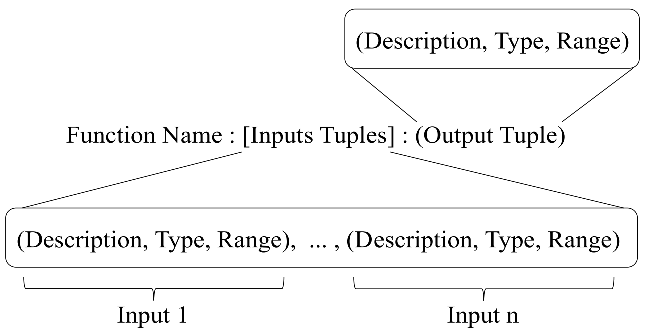


FIGURE 8. API engine structure.



(Description, Type, Range)
Function Name : [Inputs Tuples] : (Output Tuple)
(Description, Type, Range), ... , (Description, Type, Range)
Input 1
Input n

FIGURE 9. API structure.

a thing's metadata for how uniquely the thing can be viewed through the smart space.

- The *attachment manager* parses the information that models the thing's cloud-based attachments and its entities and builds the required interactions with these attachments.
- The *device manager* opens a communication channel with a device management server that either resides on the edge or cloud. This module interacts with the IoT-DDL manager module to access information about the management server (e.g., server IP address and access parameters). The Atlas thing then registers itself as a client at the server side for provisioning, managing, and configuring different attributes during the thing's lifetime as well as authorized management commands.
- The *API engine* formulates descriptive interfaces for the services offered by the thing's entities. As Fig. 8 shows, the API engine consists of four sub-modules: 1) the Bundle & API generator interacts with the IoT-DDL manager to parse services' information (inputs, outputs, types, ranges, descriptions, and functionality) stored in the IoT-DDL and builds a descriptive API for each service, after generating a bundle that fulfills the specifications of the service; 2) service execution oversees packaging the relevant input values, sending them to the running bundle, and retrieving the result values before handing them off; 3) the API parser and validator parses the received applications (API calls) from thing mates, checks their validity, and hands them to the bundle repository; 4) the bundle repository stores and manages the generated bundles and handles bundle execution through the service execution sub-module. Each API, as Fig. 9 illustrates, has three parts: 1) the function name as a short description of the offered service; 2) a list of inputs to the service, where each input is represented in a tuple that holds an input description, data type, and value range; and 3) the expected output of executing the service, where the output is represented in a tuple that also holds the description, data type, and value range. The generated API is announced to the thing mates through tweets. On the other hand, an application (API call) is captured and parsed in the architecture's tweeting sublayer. This API call holds the function name followed by a list of values representing the corresponding inputs to the service.

B. SUMMARY OF OTHER SUBLAYERS

The tweeting sublayer tools the thing with an explicit capability to uniquely define itself in the smart space, in addition to discovering thing mates and securely interacting with them. The tweeting sublayer represents the gateway that regulates the communication and queries between the interface and DDL sublayers, and is further composed of the following modules:

- The *interactions and tweeting engine* models a thing's descriptive information about its identity and generated APIs into sets of identity and API tweets, respectively. Identity tweets hold metadata about a thing's identity, description, entities, and capabilities. API tweets hold metadata about the generated APIs and the corresponding APIs. Such tweets are forwarded to the interface sublayer to be announced to thing mates. At the same time, the engine parses the received tweets (information-based interactions) as well as action-based interactions (e.g., applications, management commands, and configurations) from thing mates. The engine then forwards the interaction to the DDL sublayer modules according to the parsed content.
- The *security engine and application runtime* decodes received interactions from thing mates to ensure authorization and authentication, and encodes interactions from the thing to its thing mates. The security engine and application runtime interacts with the interactions and tweeting engine to build certified interactions from one side, and with the architecture's host interface layer from the other side.

The interface sublayer holds the different communication protocols (e.g., MQTT, CoAP, etc) that allow the thing to engage with its thing mates. The sublayer announces the built tweets to the smart space and captures others' tweets and action-based interactions to be forwarded to the tweeting sublayer for processing.

V. IMPLEMENTATION

In this section, we describe the implementation details of the parts of the Atlas thing architecture that address the various aspects of IoT-DDL and thing requirements. The Atlas thing architecture takes advantage of: 1) OMA-LwM2M, which is lightweight device management standard that targets low-power and constrained devices with the low-overhead REST data model and point-to-point communication in a client-server fashion, 2) IP Smart Object (IPSO) that provides a common design pattern and semantic interoperability across IoT devices that support LwM2M for a more reusable design to composite modular objects, 3) CoAP, which is a client/server protocol that provides a request/report paradigm model over UDP and interoperates with HTTP and the RESTful Web through simple proxies, 4) MQTT, which is one of the most widely-used communications protocols that uses a publish/subscribe architecture on top of the TCP/IP protocol.

The implementation demonstrates the feasibility of the architecture's deployment on a variety of real platforms. Finally, we provide a proof-of-concept implementation of an IoT application using the architecture implementation and IoT-DDL to show the interaction between things on the one hand, and between the thing and the device management server and log server attachments on the other hand. As mentioned earlier, our focus in this paper is the device descriptive layers that reflect the different parts of IoT-DDL.

A. THING PROVISIONING AND MANAGEMENT

The first requirement for a thing to be part of the IoT ecosystem is that it must be seamlessly integrated into the ecosystem, so it can be managed and configured with minimal human intervention. In this section, we target this essential requirement using two widely used standards for device management and object modeling: the Open Mobile Alliance Lightweight M2M (OMA-LwM2M) and the IP Smart Object (IPSO) Alliance, respectively.

Liblwm2m is an open source implementation for OMA-LwM2M developed by the Wakaama project in Eclipse [22], [23]. The Atlas architecture extends Liblwm2m to allow device management [24], [25] not only for OMA standard objects but also for the different aspects of IoT-DDL. The architecture translates the different sections and subsections of IoT-DDL into a set of dynamic objects, called Atlas objects. Atlas objects, as shown in Fig. 10, represent the different entities, services, resources, and attachments of an Atlas thing. Atlas objects are based on the object resource data representation model proposed by OMA-LwM2M and utilize the IPSO idea of composite objects for higher modular object design. Extending the broad standard for OMA with Atlas objects allows seamless engagement of Atlas things with OMA-LwM2M/IPSO-powered entities in a smart space.

The device manager module of the DDL sublayer in the architecture communicates with the IoT-DDL manager

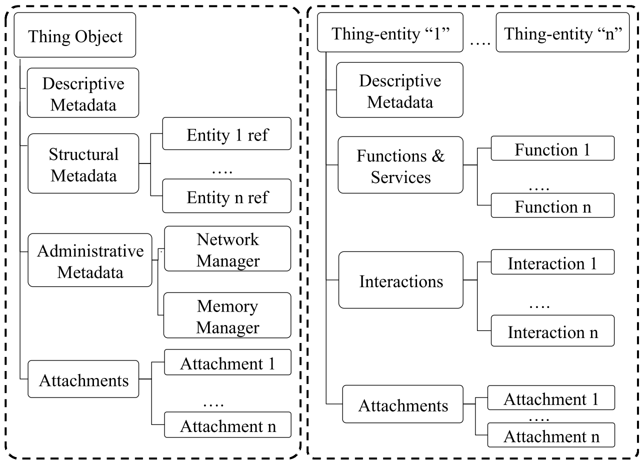


FIGURE 10. Atlas objects' tree.

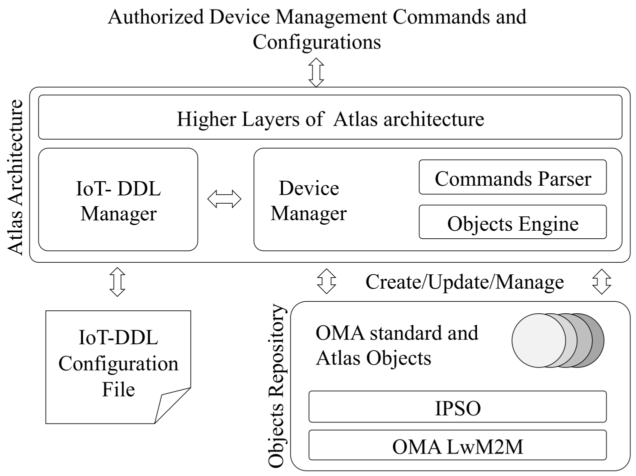

Authorized Device Management Commands and Configurations
Atlas Architecture
Higher Layers of Atlas architecture
IoT- DDL Manager
Device Manager
Commands Parser
Objects Engine
Create/Update/Manage
IoT-DDL Configuration File
Objects Repository
OMA standard and Atlas Objects
IPSO
OMA LwM2M

FIGURE 11. Device manager and OMA objects.

module to access information about the OMA management server (e.g., server IP address and access parameters). The Atlas thing registers itself with the server as an OMA client, where the registration process requires a thing to register a tree of its programmed objects at the server side. Such a tree is a hierarchal structure of both OMA-standard objects and Atlas objects generated by the thing. The device manager module, as shown in Fig. 11, creates and manages both OMA standard objects and Atlas objects through an object engine. Atlas object creation and management occur on demand when IoT-DDL is uploaded to the thing during lifetime updates. At the same time, authorized lifetime management and updates from the management server trigger the device manager's object engine module to maintain the corresponding objects and enables authorized dynamic updates to the IoT-DDL parameters and attributes during the lifetime of the thing.

B. ATLAS THING COMMUNICATION

To enable thing-to-thing and thing-to-edge or thing-to-cloud information- and action-based interactions, the

TABLE 1. **Atlas topics for the MQTT protocol.**

Atlas IoT Interactions	Publish Topics	Subscribe Topics
Information-based Interaction (Tweet)	Tweet/Thing/Identity Tweet/Entity/Identity Tweet/API	Tweet/Thing/Identity Tweet/Entity/Identity Tweet/API
Action-based Interaction	Interaction/ (ThingMate-ID)	Interaction/ (ThingMate-ID)

TABLE 2. **Atlas resources for the CoAP protocol.**

Atlas IoT Interactions	Thing Resources (URL)	REST Method
Information-based Interaction (Tweet)	Tweet_ThingIdentity Tweet_EntityIdentity Tweet_API	GET
Action-based Interaction	Interaction_(ThingMate -ID)	POST/PUT

architecture exploits widely used communication protocols for IoT and constrained environments: Message Queuing Telemetry Transport (MQTT) and the Constrained Application Protocol (CoAP). In this section, we give an overview of the protocols and describe how their implementation within the Atlas thing architecture goes beyond just enabling communication to support other thing requirements and IoT-DDL design choices. The interface and communication engine of the Atlas thing architecture adopts the open-source C/C++ implementation of MQTT developed by the Paho project in Eclipse [30]. The implementation allows Atlas things to connect to an MQTT broker, subscribe, and publish with respect to a predefined set of topics. As a proof of concept, the architecture utilizes a connection with the cloud-based MQTT broker HiveMQ dashboard [31] to publish and subscribe to the different topics.

As discussed earlier, an Atlas thing tweets identity information about itself, thing entities, and generated APIs to its thing mates, and similarly receives other things' tweets. Things can also interact (e.g., through issuing an API call) with each other. The network manager subsection of the administrative thing metadata of the IoT-DDL specification lists the required configuration of MQTT. The configuration includes the URL to the MQTT broker, the listening port, and a list of topics to subscribe to and publish accordingly. Table 1 lists the different topics the Atlas thing publishes and subscribes to in order to announce its own and receive others' tweets, respectively. Furthermore, Atlas things interact using another set of topics that include the things' IDs. For an Atlas thing to receive interactions, it must subscribe to an interaction topic that holds its own ID. To forward an interaction to another thing mate, it must publish a topic that includes the thing mate's ID.

The interface and communication engine of the Atlas thing architecture adopts the open-source C++ implementation of CoAP developed by Noisy Atom [40]. This implementation allows Atlas things to tweet and interact with respect to a predefined set of resources. The Atlas architecture uses CoAP protocol support for multicasting, which allows the thing to broadcast tweets to all listening things. We extend the imported library with Unix multicast sockets to enable the CoAP multicast feature.

The network manager subsection of the thing Resources metadata of the IoT-DDL specification lists the required configuration of CoAP. The configuration includes the listening port, a list of RESTful methods, and the corresponding resources. Table 2 lists the resources an Atlas thing uses to announce its own tweets and ask for others' tweets. At the same time, Atlas things interact using another set of resources that include the thing mates' IDs.

We are currently working on enabling Atlas thing communication between thing mates that speak different protocols (currently limited to REST, CoAP and MQTT). By developing protocol adaptors as thing attachments, two things will be able to communicate despite their protocol differences.

C. ATLAS THING BUNDLES

To best facilitate the creation and modification of arbitrary thing services, the architecture adopts the Apache licensed C++ Mircro Services project [51] for dynamic service-oriented API functionality. Each thing service is represented as an independent microservice, and can be executed through a single call to the corresponding API taking inputs and returning outputs as specified by the IoT-DDL. In this section, we describe the steps taken to convert the textual service description of the IoT-DDL into a compiled microservice capable of running on an Atlas thing and discuss how the architecture is expanded to allow for generation of services on the fly.

The C++ Micro Services project aims to allow for the creation of service-based applications based on the dynamic module system of OSGi [52]. Each microservice, called a bundle, is a specially packaged shared library object that runs in its own context within the microservices framework. The state of these bundles can be managed dynamically during the lifetime of the Atlas thing. The Atlas architecture uses C++ Micro Services to allow loading, running, and destroying its service bundles throughout execution, and to access the running bundles' functionalities through a generic interface. Before a thing's services can be ran, their bundles are dynamically generated by the API Engine of the architecture from the specification of the uploaded IoT-DDL. Once the architecture has parsed the IoT-DDL, the inputs and outputs are mapped to their corresponding native types and the generic actions are mapped to platform-specific function calls. The API Engine handles the execution of the bundle upon receiving the corresponding API call from other things in the smart space.

D. ATLAS THING SECURITY

As mentioned earlier, the third essential requirement for things to be part of the ecosystem is the capability to interact with thing mates in a secure way that enables both the authorization and authentication aspects of communication. The architecture exploits the lightweight symmetric key [38] encryption of AES to create secure dynamic communication channels between things and thing mates.

AES is a widely used symmetric lightweight block cipher security protocol [36], [37], [39]. For AES, the default block size is 128 bits or 16 bytes. A mode of operation describes how to repeatedly apply a cipher's single block operation to securely transform amounts of data larger than a block. Each mode requires an initialization vector (IV) to ensure distinct ciphertexts are produced even when the same plaintext is independently encrypted multiple times with the same key. In the cipher block-chaining (CBC) mode of operation, each block of plaintext is XORed with the previous ciphertext block before it is encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. CBC is the most commonly used mode of operation.

The Atlas thing architecture utilizes AES with CBC as the mode of operation using the Crypto++ library [41]. The security engine of the tweeting sublayer assumes a master secret key and IV, and encryption/decryption methods are securely stored within the device secure elements of the OS services. These security elements are unique to Atlas things in smart spaces, where the space owner can generate a new master key and IV to be securely deployed to the things through the device management server. The security engine generates dynamic session keys from the symmetric predefined master key. Atlas thing A and Atlas thing B can establish a session key as follows:

1. Thing A generates a random number (R1) and encrypts a message holding its own ID and R1 using the master key.
2. Thing B decrypts the message using the predefined master key, saves R1, and generates a new random number (R2).
3. Thing B encrypts a message holding its own ID and R2 using the master key.
4. Thing A and thing B each encrypt a concatenated value of R1 and R2 using the master key and generate a session key.

E. ATLAS THING ARCHITECTURE AND IoT-DDL PROOF-OF-CONCEPT

Our proof of concept utilizes two things in a smart space, a Raspberry Pi model B sensor platform running Raspbian OS, and a Beaglebone Black sensor platform running Angstrom OS. The Beaglebone Black thing is connected to the on/off circuit of a coffee maker as an attached hardware entity, while the Raspberry Pi thing offers an alarm clock service as a built-in software entity, as shown in Fig. 12. The IoT-DDL configuration files are developed and uploaded on both things. These files indicate the identity of each thing, including their inner entities, resources, and services. The Beaglebone Black thing offers two services, one to start

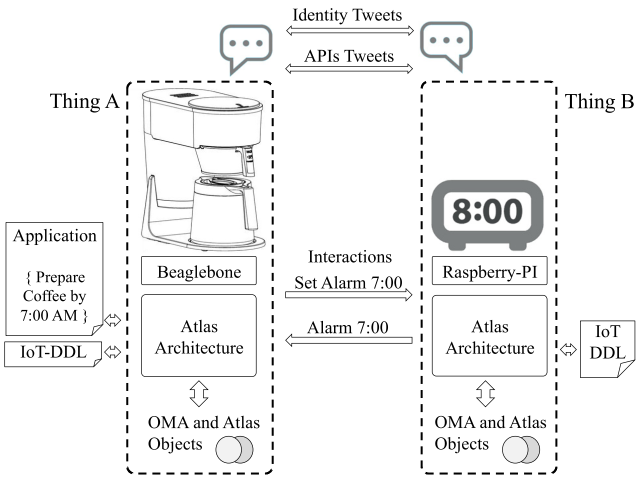


FIGURE 12. Proof-of-concept implementation.

brewing coffee and remain on for a specific time duration, and the other for switching off the coffee maker. The Raspberry Pi thing, on the other hand, offers two services to set and clear the software alarm clock entity. When both things are powered, the proof-of-concept implementation of the Atlas thing architecture in starts parsing the different sections and subsections of the uploaded IoT-DDL. Each thing now identifies itself, discovers the different services and functions it can offer to the smart space it is located in, and starts generating its own APIs. Each thing starts looking for thing mates by broadcasting thing identity tweets, entity identity tweets, and generated API advertisement tweets.

The prototype starts by assuming that an IoT application in the Beaglebone Black thing requires turning on the coffee maker when the alarm triggers. At the same time, both things register themselves as OMA clients that can connect to the OMA server and register their OMA standard objects and Atlas objects. On the server side, an authorized user can browse the different connected clients, view the list of registered objects, and update their attributes. We also provide a NodeJS-based HTTP-log server that resides on an edge in the smart space as an example of a thing attachment. The attachment manager module of the DDL sublayer parses the attachment settings (e.g., server URL, port, access information, and update interval) through the IoT-DDL manager module. The Raspberry Pi creates a communication channel to PUT the current status of the thing (e.g., tweeting, executing an application, management phase) when the status changes or at every update interval if there are no changes.

Full details about the IoT-DDL configuration files for both the Beaglebone and Raspberry Pi things, as well as a short video of the coffee maker demo are available online as supplemental materials [45].

VI. BENCHMARKING

In this section, we provide a benchmarking study to measure time and energy consumption of the different Atlas thing

TABLE 3. Sensor platform specifications.

Specifications	Raspberry Pi Model B	Dragon Board 410C	Beaglebone Black
OS	Raspbian	Debian	Angstrom
Processor	900-MHz Quad-Core ARM Cortex A7	1.2-GHz Quad-Core ARM Cortex A53 Snapdragon	1-GHz Sitara AM3359 ARM Cortex A8
Network Module	Raspberry Pi Wi-Fi adapter 2.4-GHz CanaKit	Integrated Digital core 2.4	Edimax USB Wi-Fi 2.4 GHz
RAM	1GB	1GB	512MB
Flash Storage	4GB SD card	8GB eMMC	4GB eMMC

architecture aspects developed on three heterogeneous things. The study aims to show the feasibility of deploying the architecture on real platforms. The aspects benchmarked are the thing's capability to generate tweets, to encrypt and decrypt action-based interactions, to be configured and managed, and to interact using widely accepted communication protocols. The things used in this study are the Raspberry Pi Model B, Qualcomm Dragon Board 410C, and Beaglebone Black sensor platforms with the specifications listed in Table 3.

For a unified measurement, we uploaded the same IoT-DDL configuration file (which was shown earlier in Section III) to the three things. The uploaded IoT-DDL shows that the thing contains an attached coffee maker as a hardware entity that provides two services (turn on for a certain duration and turn off). The code footprint of the IoT-DDL in addition to the current version of Atlas thing architecture that imports the OMA-LwM2M standard and AES protocol and supports both CoAP and MQTT communication standards is 13 megabytes approximately. The code footprint—the actual machine instructions that resides the flash memory—shows the proposed framework is lightweight enough to fit on constrained devices. Such small code footprint also reflects that the actual running code does not require too much RAM to execute.

Time is measured using the Unix-Chrono library for a high-resolution clock cast to microseconds. Energy consumption is measured using a PowerJive USB-based device that calculates voltage and capacity [44]. To avoid data outliers, the time measurement of a single operation is averaged over five measurements. The energy consumption of a single operation is the average value obtained from running the operation a large number of times in a 10-minute period. The energy consumption of the processes running in the background of the thing's OS is also calculated and is subtracted from the calculated energy consumption of the operation.

In the first subsection, we provide a benchmarking study that focuses on the functionalities of both tweeting and DDL sublayers of the architecture, where we measure—in terms of the time performance and energy consumed (Table 4)—the Atlas thing's capability to generate tweets and to encrypt and decrypt action-based ad-hoc interactions. In the second subsection, we provide a set of experiments to benchmark both time performance (Fig. 13) and energy consumption (Table 5) of the different device management capabilities supported by the architecture. The device management capabilities include the ability of the thing to generate the different Atlas objects, connect to the OMA server, and register the objects. The second subsection also provides a set of experiments to benchmark both time performance and energy consumption for adopting the widely accepted IoT communication protocols MQTT (Fig. 14 and Table 6) and CoAP (Fig. 15 and Table 7). The communication capabilities include the different functionalities required by the thing to send tweets

TABLE 4. Benchmark time (in microseconds) and energy consumption (in watt seconds).

		Interaction Encryption	Interaction Decryption	Tweet Generation
Raspberry Pi Model B	Time	2152	947	128484
	Energy	7.4e-5	7.8e-5	0.014
Dragon board 410C	Time	1438	916	26159
	Energy	4.5e-5	5e-5	0.03
Beaglebone Black	Time	3277	1414	54298
	Energy	0.000114	0.000116	0.0506

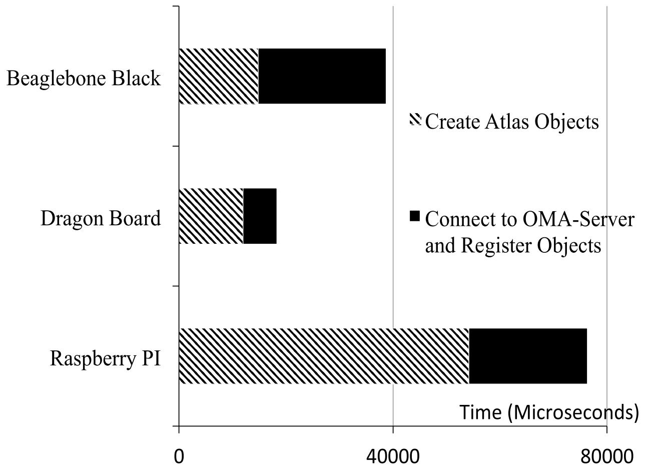


FIGURE 13. Time comparison for device manager functionalities.

TABLE 5. Energy consumption (in watt seconds) measurements for OMA device manager functionalities.

	Generated Atlas objects	Connect to OMA server then register objects
Raspberry Pi	0.0032	0.0157
Dragon board	0.00073	0.003845
Beaglebone Black	0.00147	0.010613

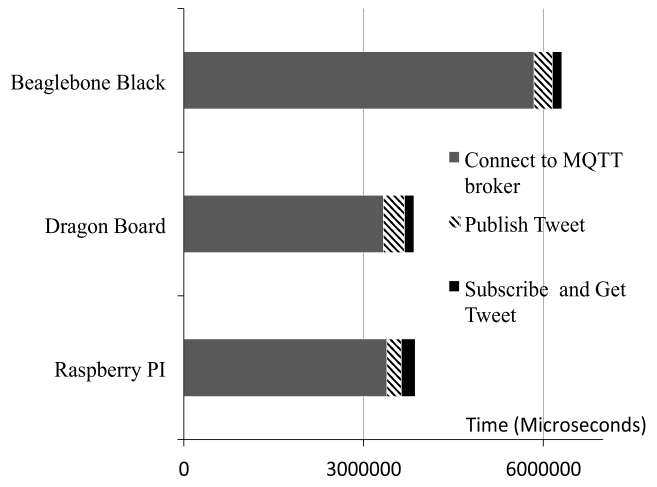


FIGURE 14. Time comparison for the different aspects of MQTT protocols.

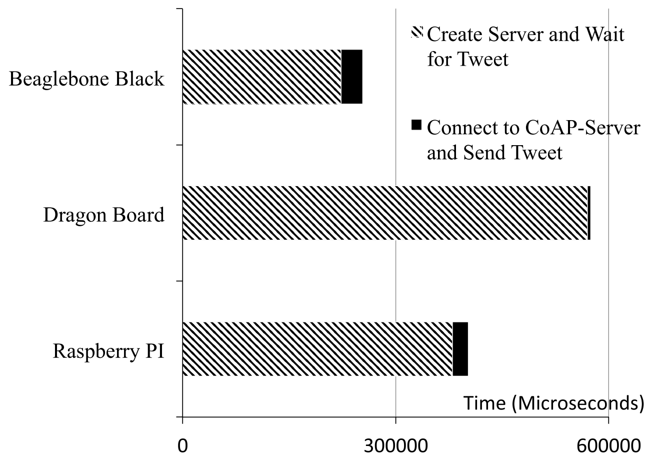


FIGURE 15. Time comparison for the different aspects of CoAP protocols.

TABLE 6. Energy consumption (in watt seconds) measurements for the different aspects of MQTT protocol.

	Open TCP connection with MQTT broker	Publish single tweet (64-bytes)	Subscribe and receive single tweet (64-bytes)
Raspberry Pi	0.222	0.301	0.561
Dragon board	0.251	0.108	0.2729
Beaglebone Black	0.464	0.112	0.1098

TABLE 7. Energy consumption (in watt seconds) measurements for the different aspects of CoAP protocol.

	CoAP server listens and receives single tweet (64-bytes)	CoAP client connects to server and sends single tweet (64-bytes)
Raspberry Pi	0.020	0.007
Dragon board	0.000633	0.003267
Beaglebone Black	0.00287	0.00634

and listen to tweets from thing mates in the smart space. The third subsection provides analysis and discussion on the provided benchmarking study for the different functionalities and capabilities of the Atlas thing architecture required by the thing to engage in wide range of interactions and interconnections with other things in the smart space during the thing's lifetime.

A. BENCHMARKING TWEET GENERATION AND SECURE INTERACTIONS

The first set of measurements focuses on the functionalities of both tweeting and DDL sublayers of the architecture. These functionalities are in terms of the thing's capability to generate tweets and to encrypt and decrypt action-based interactions. The generated tweets are about thing identity (64 bytes), thing entity (64 bytes), and the generated API for each of the two services (60 bytes each). The size of the generated tweets and APIs depends on the developed IoT-DDL for the Atlas thing. However, we limited the sizes to 64 and 60 bytes for unified measurements on time and energy on the different sensor platforms. The secure action-based interaction (API call forwarded by the thing or received from a thing mate) applies the AES-CBC mode of operation. AES uses a key and IV, each at 16 bytes, to encrypt and decrypt a 60-byte interaction. Table 4 shows the measurements of both time (microseconds) and energy consumption (watt-seconds) on the different hardware platforms. Analysis of these measurements is presented in Section C.

B. BENCHMARKING OMA DEVICE MANAGEMENT AND COMMUNICATION PROTOCOLS

The second set of experiments focuses on the architecture's management and communication functionalities. These functionalities are in terms of OMA device management aspects as well as the different communication protocols supported by the architecture.

After the IoT-DDL is uploaded to the thing, the thing starts generating Atlas objects for the corresponding IoT-DDL sections. The device manager module communicates with the IoT-DDL manager module to access information about the OMA management server (e.g., server IP address and access parameters). The Atlas thing then registers itself as an OMA client at the server, where the registration process requires the thing to register a tree of its programmed objects (both Atlas objects and standard OMA objects) at the server side. For sake of simplicity, we limit Atlas object generation to the descriptive metadata of the thing, the entity, and the attachments. Fig. 13 compares the three sensor platform things we used in terms of the time required to create Atlas objects on the one hand, and connecting to the OMA server on the local network and registering the objects' tree on the other hand. Table 5 illustrates energy consumption rate in terms of the consumed watts per second of these functionalities on the different sensor platforms.

Furthermore, the Atlas thing architecture supports the widely accepted IoT communication protocol, MQTT, and utilizes a connection with the cloud-based MQTT broker

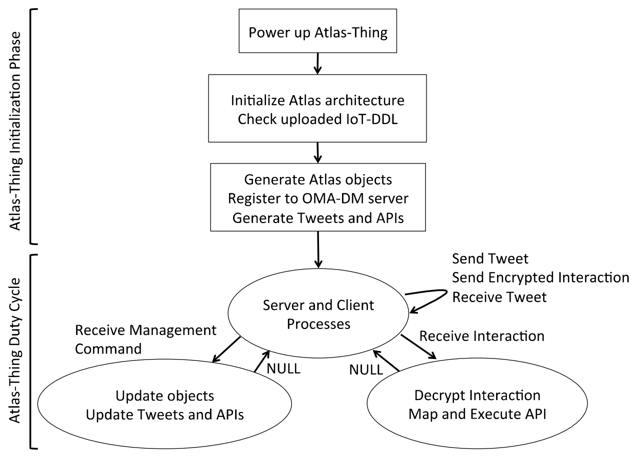


FIGURE 16. **Atlas thing initialization phase and duty cycle.**

HiveMQ dashboard [31] to publish and subscribe to the different topics. Fig. 14 compares the three sensor platforms in terms of the time (in microseconds) required to connect to the MQTT broker, publish a 64-byte tweet, subscribe to a topic, and then get a 64-byte tweet from a thing mate. Table 6 illustrates energy consumption in terms of the consumed watts per second of the different supported MQTT functionalities on the different sensor platforms. Fig. 15 compares the three sensor platforms in terms of the time (in microseconds) required to create a CoAP server at the thing and wait for a tweet from a thing mate from one side, and to create a client side that connects to the CoAP server of a thing mate, then sends a 64-byte tweet from the other side. Table 7 illustrates the energy consumption in terms of the consumed watts per second of the different functionalities on the different sensor platforms. It should be noted that this set of experiments depends mainly on the network connection and network module used. Analysis of the results of this second set of measurements is presented in Section C.

C. ANALYSIS OF THE BENCHMARKING STUDY

We analyze the results of our benchmarking study in terms of time performance and energy consumption. We start first with the energy analysis. Understanding an Atlas thing duty cycle, which is based on the Atlas thing architecture, helps in analyzing the measured energy data. As Fig. 16 illustrates, an Atlas thing goes through an initialization phase, followed by one or more Atlas thing duty cycles until thing termination (e.g., battery depletion). The initialization phase starts with powering up the Atlas thing until it is ready to engage with its thing mates and the device management server. In this phase, the Atlas thing initializes the architecture and verifies that the IoT-DDL is uploaded. The Atlas thing generates Atlas objects, registers itself to the OMA server specified in the IoT-DDL, and then generates tweets and APIs for the offered services. Directly after the initialization phase, the Atlas thing starts engaging with thing mates through tweets and actionable interactions through the Atlas thing duty cycle. The duty cycle starts running concurrent (threaded) server and client processes to receive and send interactions, respectively. Receipt of a management command triggers updates of Atlas and OMA objects, as well as tweets and APIs, while receipt of an interaction requires decrypting the interaction, and mapping then executing the corresponding API.

We can calculate battery lifetime in terms of hours to run an Atlas thing continuously using (1).

$$\begin{aligned} &\textit{Battery lifetime} \\ &\quad \cong \textit{Time of Duty Cycle} \\ &\qquad \times \left(\frac{\textit{Battery Capacity} \times \textit{Battery voltage}}{\textit{Energy consumed per Duty Cycle}}\right) \end{aligned} \tag{1}$$

where battery lifetime is calculated in hours, battery capacity is in milliamp hours (mAh), and battery voltage is the battery's initial voltage. The main assumption is that the battery is capable of maintaining a voltage level over time to operate the thing. Duty cycle includes the Atlas thing duty cycle in addition to the background running processes of the thing's underlying OS. As an example, Rayovac 4AA alkaline batteries with six volts and 2400 mAh capacity can run (according to (1)) a Beaglebone Atlas thing for 28 hours, a Dragon Board Atlas thing for 12 hours, and a Raspberry Pi Atlas thing for 7.5 hours. Such large differences are due to the processes running in the background of the thing's OS and the high capabilities of the Dragon Board and the Raspberry Pi (e.g., keyboard, mouse) compared to Beaglebone Black. To demonstrate the accuracy of the proposed equation for battery lifetime, Rayovac 4AA batteries were able to run Raspberry Pi (which requires a minimum of 2000 mAh and five volts to operate with full peripherals) for approximately six hours. The difference between the expected and the real value for the battery lifetime is due to the drop of the battery voltage below five volts. Such a drop forces the Raspberry Pi to shut down connected peripherals (e.g., Wi-Fi module). However, most of real life examples of things that exist in smart spaces either have their own continuous source of power (e.g., smart home appliances) or efficient power management modules (e.g., smartphones).

We next discuss time performance. The differences in the measured time for the features and functionalities of the Atlas thing architecture depend on the specifications of each platform as mentioned in Table 3. For the first set of measurements, the time to complete the operation mainly depends on the clock frequency of the processor as well as the available RAM to keep track of the different internal operations and the results. A Dragon Board 410C with 1GB of ARM and 1.2GHz quad-core performs faster compared to the other platforms on the same set of operations. For the second set of measurements, the time to complete the operation mainly depends on the network connectivity and the current traffic as well as the properties of the different Wi-Fi modules mounted on each platform. It is worth noting that the Dragon Board 410C with integrated 2.4GHz Wi-Fi module on board performs better compared to the other platforms that are using external USB Wi-Fi modules.

VII. CONCLUSION

In this paper, we argue that the promise and transformative success of the IoT vision will greatly depend on how its main ingredient—the thing—is prepared, aligned, and made able to engage in such a mission. The fragmented nature of IoT things requires significant efforts to integrate, manage, and configure such a wide heterogeneity of things. We propose IoT-DDL, a machine- and human-readable descriptive language that tools a thing to self-discover and share its own capabilities, entities, and services, including the various cloud-based resources that could be attached to it to extend it over time. Making things describable using IoT-DDL enables self-discovery so the thing itself becomes self-aware of what it can offer and what its capabilities are. It also empowers the seamless integration, configuration, and management of things with minimal human intervention and enables the various secure interactions that support the distributed nature of IoT. We also present the Atlas thing architecture, which fully exploits the goals of the IoT-DDL and its specifications. The architecture goes beyond and above standard embedded OS services to provide new layers and services with novel capabilities necessary for things to have to be part of the IoT. In addition, the architecture takes advantage of widely used device management, security, and IoT communication standards to enable thing engagement in secure ad hoc interactions with thing mates and space users. We prove the feasibility of deploying the Atlas architecture and the IoT-DDL on real hardware platforms through a proof-of-concept implementation as well as a benchmarking study to validate the feasibility of our approach. The study measures both time performance and energy consumption rate. We believe that our work in thing architectures will go far beyond just enabling interactions and automatic generation of service bundles and APIs, to pave the way for powerful programming models that are currently missing. We are currently working on refining our Atlas thing architecture with an ultimate aim to formulate new IoT programming models and tools.

REFERENCES

[1] R. Want and S. Dustdar, "Activating the Internet of Things [Guest editors' introduction]," *Computer*, vol. 48, no. 9, pp. 16–20, Sep. 2015.
[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generat. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.
[3] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of Things: Vision, applications and research challenges," *Ad Hoc Netw.*, vol. 10, no. 7, pp. 1497–1516, 2012.
[4] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.
[5] L. Coetzee and J. Eksteen, "The Internet of Things—Promise for the future? An introduction," in *Proc. IEEE IST-Africa Conf.*, May 2011, pp. 1–9.
[6] L. Tan and N. Wang, "Future Internet: The Internet of Things," in *Proc. 3rd Int. Conf. Adv. Comput. Theory Eng. (ICACTE)*, vol. 5. Aug. 2010, pp. V5-376–V5-380.
[7] J. Jimenez, M. Koster, and H. Tschofenig, "IPSO smart objects," in *Proc. Position Paper IoT Semantic Interoperability Workshop*, 2016, pp. 1–7.
[8] S. K. Datta and C. Bonnet, "A lightweight framework for efficient M2M device management in oneM2M architecture," in *Proc. Int. Conf. Recent Adv. Internet Things (RIoT)*, Apr. 2015, pp. 1–6.
[9] M. I. Robles and P. Jokela, "Design of a performance measurements platform in lightweight M2M for Internet of Things," in *Proc. IRTF ISOC Workshop Res. Appl. Internet Meas. (RAIM)*, 2015, pp. 1–4.
[10] M. Robles, D. D'Ambrosio, J. J. Bolonio, and M. Komu, "Device group management in constrained networks," in *Proc. 13th IEEE Int. Conf. Pervasive Comput. Commun. (PerCom Workshops)*, Mar. 2016, pp. 1–6.
[11] C. A. L. Putera and F. J. Lin, "Incorporating OMA lightweight M2M protocol in IoT/M2M standard architecture," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 559–564.
[12] S. Rao, D. Chendanda, C. Deshpande, and V. Lakkundi, "Implementing LwM2M in constrained IoT devices," in *Proc. IEEE Conf. Wireless Sensors (ICWiSe)*, Aug. 2015, pp. 52–57.
[13] G. Klas, F. Rodermund, Z. Shelby, S. Akhouri, and J. Höller, "'Lightweight M2M': Enabling device management and applications for the Internet of Things," Vodafone, ARM, Ericsson, White Paper, Feb. 2014.
[14] Y. Xu and A. Helal, "Scalable cloud–sensor architecture for the Internet of Things," *IEEE Internet Things J.*, vol. 3, no. 3, pp. 285–298, Jun. 2016.
[15] S. Helal and Y. Xu, "Scalable and energy-efficient cloud-sensor architecture for cyber physical systems," in *Proc. NSF Workshop Big Data Anal. CPS, Enabling Move IoT Real-Time Control*, Seattle, WA, USA, Apr. 2015.
[16] C. Chen and A. Helal, "Device integration in SODA using the device description Language," in *Proc. IEEE 9th Annu. Int. Symp. Appl. Internet*, Jul. 2009, pp. 100–106.
[17] *Device Description Language Specification (Version 1.2)*, Mobile Pervasive Comput. Lab., Univ. Florida, Gainesville, FL, USA, Nov. 2008. [Online]. Available: https://www.cise.ufl.edu/~helal/atlas/ddl/DDL-Spec-1.2.2.pdf
[18] (2016). *Google Weave*. [Online]. Available: http://developers.google.com/weave/
[19] (2016). *Google Brillo*. [Online]. Available: http://developers.google.com/brillo/
[20] (2016). *Amazon AWS IoT*. [Online]. Available: http://www.aws.amazon.com/iot/
[21] (2016). *Amazon AWS IoT*. [Online]. Available: http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html
[22] (2017). *Eclipse Wakaama*. [Online]. Available: https://projects.eclipse.org/projects/technology.wakaama
[23] (2017). *Wakaama Implementation of the Open Mobile Alliance's Lightweight M2M*. [Online]. Available: https://github.com/eclipse/wakaama
[24] *LightweightM2M Editor for OMA Objects and Resources*. [Online]. Available: http://devtoolkit.openmobilealliance.org/OEditor
[25] (2017). *LwM2M XML Schema From Editor Schema*. [Online]. Available: http://technical.openmobilealliance.org/tech/profiles/LWM2M.xsd
[26] C. Chen and A. Helal, "A device-centric approach to a safer Internet of Things," in *Proc. Int. Workshop Netw. Object Memories Internet Things (NOMe-IoT)*, Beijing, China, Sep. 2011, pp. 1–6.
[27] C. Chen and S. Helal, "Sifting through the jungle of sensor standards," *IEEE Pervasive Comput.*, vol. 7, no. 4, pp. 84–88, Oct./Dec. 2008.
[28] (2014). *MQTT is a Machine-to-Machine (M2M) 'Internet of Things' Connectivity Protocol*. [Online]. Available: http://Mqtt.org
[29] J. King, R. Bose, H.-I. Yang, S. Pickles, and A. Helal, "Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces," in *Proc. IEEE Conf. Local Comput. Netw.*, Nov. 2006, pp. 630–638.
[30] *Eclipse Paho Open-Source Implementation of MQTT Project*. [Online]. Available: https://eclipse.org/paho/
[31] (2017). *HiveMQ MQTT Dashboard*. [Online]. Available: https://mqtt-dashboard.com/
[32] (2017). *Web of Things at W3C*. [Online]. Available: https://w3.org/WoT/
[33] (2016). *Experimental Implementation of the Web of Things Framework*. [Online]. Available: https://github.com/w3c/web-of-things-framework
[34] (2016). *The Constrained Application Protocol (CoAP)*. [Online]. Available: http://coap.technology/
[35] *The Constrained Application Protocol (CoAP)*, document RFC 7252, 2014. [Online]. Available: https://www.tools.ietf.org/html/rfc7252
[36] M. Katagi and S. Moriai, "Lightweight cryptography for the Internet of Things," Sony Corp., Tokyo, Japan, Tech. Rep., 2008, pp. 7–10.
[37] K. Gaurav, P. Goyal, V. Agrawal, and S. L. Rao, "IoT transaction security," in *Proc. 8th Int. Conf. Internet Things (IoT)*, 2015, pp. 5–6.
[38] M. Ebrahim, S. Khan, and U. B. Khalid. (May 2014). "Symmetric algorithm survey: A comparative analysis." [Online]. Available: https://arxiv.org/abs/1405.0398

[39] J. Thakur and N. Kumar, "DES, AES and Blowfish: Symmetric key cryptography algorithms simulation based performance analysis," *Int. J. Emerg. Technol. Adv. Eng.*, vol. 1, no. 2, pp. 6–12, 2011.

[40] *CoAP Implementation by Noisy Atom*. [Online]. Available: http://www.noisyatom

[41] (2014). *Crypto Library of Cryptographic Schemes*. [Online]. Available: https://www.cryptopp.com

[42] (2013). *NXP A700X_Family for Secure Authentication Microcontroller*. [Online]. Available: https://www.mouser.com/ds/2/302/A700X_FAM_SDS-119904.pdf

[43] (2017). *Samsung ARTIK Modules*. [Online]. Available: https://www.artik.io/modules/

[44] (2017). *PowerJive USB Voltage/Amps Power Meter*. [Online]. Available: http://www.measuringsupply.com/artifact/1402679/

[45] (2017). *Coffee Maker Demo Video and Things IoT-DDL Files as Additional Online Materials on GitHub*. [Online]. Available: https://www.github.com/AEEldin/IoTDDL_CoffeeMakerDemo

[46] (2017). *Atlas IoT-DDL Builder Web Tool*. Accessed: 2017. [Online]. Available: https://www.cise.ufl.edu/~aekhaled/AtlasIoTDDL_Builder.html

[47] (2012). *Constrained RESTful Environments (CoRE) Link Format*. [Online]. Available: https://www.tools.ietf.org/html/rfc6690

[48] S. K. Datta and C. Bonnet, "Describing things in the Internet of Things: From CoRE link format to semantic based descriptions," in *Proc. IEEE Int. Conf. Consum. Electron.-Taiwan (ICCE-TW)*, May 2016, pp. 1–2.

[49] S. Käbisch and D. Anicic, "Thing description as enabler of semantic interoperability on the Web of Things," in *Proc. IoT Semantic Interoperability Workshop*, 2016, pp. 1–3.

[50] H. Hasemann, A. Kröller, and M. Pagel, "RDF provisioning for the Internet of Things," in *Proc. 3rd IEEE Int. Conf. Internet Things*, Wuxi, China, Oct. 2012, pp. 143–150.

[51] *C++ Micro Services*. [Online]. Available: http://cppmicroservices.org/

[52] *Open Services Gateway Initiative Alliance*. [Online]. Available: https://osgi.org/

AHMED E. KHALED received the B.Sc. and M.Sc. degrees in computer engineering from Cairo University, Egypt, in 2011 and 2013, respectively. He is currently pursuing the Ph.D. degree in computer engineering, with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA. His current research interests include Internet of Things, smart spaces, and ubiquitous computing.

ABDELSALAM (SUMI) HELAL (F'15) received the Ph.D. degree in computer sciences from Purdue University, West Lafayette, IN, USA. He is currently a Professor and the Chair in digital health with the School of Computing and Communications, and with the Division of Health Research, Lancaster University, U.K. Before joining Lancaster University, he was a Professor with the Department of Computer and Information Science and Engineering, University of Florida, USA, where he directed the Mobile and Pervasive Computing Laboratory and the Gator Tech Smart House. His research interests include pervasive systems, the Internet of Things, smart spaces, with applications to digital health and assistive technologies for successful aging and independence.

WYATT LINDQUIST received the B.Sc. degree in computer engineering from the University of Florida, Gainesville, FL, USA, in 2017. He is currently pursuing the Ph.D. degree in computer science with the School of Computing and Communications, University of Lancaster, U.K. His current research interests include Internet of Things, operating systems, and embedded systems, with applications to digital health.

CHOONHWA LEE received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1990 and 1992, respectively, and the Ph.D. degree in computer engineering from the University of Florida, Gainesville, FL, USA, in 2003. He is currently a Professor with the Division of Computer Science and Engineering, Hanyang University, Seoul. His research interests include cloud computing, peer-to-peer and mobile networking and computing, and services computing technology.