Reprogramming Low-end IoT Devices from the Cloud

Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Padilla, Kaspar Schleiser, Ian Thomas

Reprogramming Low-end IoT Devices from the Cloud

Emmanuel Baccelli*, Joerg Doerr§, Ons Jallouli‡, Shinji Kikuchi†, Andreas Morgenstern§,
Francisco Acosta Padilla*, Kaspar Schleiser*, Ian Thomas‡
*INRIA †Fujitsu Laboratories ‡RunMyProcess §Fraunhofer IESE

***Abstract*—The Internet of Things (IoT) consists in a variety of smart connected objects, among which a category of low-end devices based on micro-controllers. The orchestration of low-end IoT devices is not straightforward because of the lack of generic and holistic solutions articulating cloud-based tools on one hand, and low-end IoT device software on the other hand. In this paper, we describe such a solution, combining a cloud-based IDE, graphical programming, and automatic JavaScript generation. Scripts are pushed over the Internet and over-the-air for the last hop, updating runtime containers hosted on heterogeneous low-end IoT devices running RIOT. We demonstrate a prototype working on common off-the-shelf low-end IoT hardware with as little as 32kB of memory.**

I. Introduction

The Internet of Things has been the subject of much research and development with respect to networks, protocols, gateways and devices. This research has focused on making it easy to connect sensors to the Internet using IP-based protocols and on collecting the raw data made available by such sensors for analysis and onward processing. But this means that most of today's research treats connected things as 'dumb' objects with little intelligence or configurability, limiting their ability to adapt and wasting their potential to create faster, more intelligent and more efficiently networked digital ecosystems. Despite very small memory capacity (a few kBytes [15]) and limited energy consumption (1000 times less than a RaspberryPi), low-end IoT devices can indeed execute quite complex logic, and interact with the rest of the network – just like other computers on the Internet.

In this paper we therefore investigated opportunities to model and deploy functionality directly to devices, making them more intelligent and reactive in pursuit of distributed, hyperconnected and autonomous systems. Our goal was to bring speed and agility of web- and cloud-based development to low-end IoT devices. To this end we created and tested a platform articulating of a suite of components: a generic IoT device operating system (RIOT), communicating via standard IoT application protocols with a cloud-based development tool enabling quick business-logic deployment, from the cloud to a fleet of managed IoT devices. The cloud component automatically translates graphically-programmed BPMN business logic into standard Javascript which can be deployed and executed on managed IoT devices. While some recent solutions such as NodeRED [7] provide a subset of these functionalities, we are not aware of a platform which bundles the full set of features described above.

The remainder of this paper is organized as such: first, we overview the architecture of the platform, from cloud-based IoT device behavior modeling to JavaScript logic deployment on the IoT devices. We then describe a specific use-case for which we build a prototype and describe aspects of hosting and deploying Javascript runtime logic on heterogeneous low-end IoT devices. Next, we evaluate memory and throughput requirements of our prototype on the IoT device side, which is the bottleneck. We finally review related work before we conclude.

II. Architecture

The functional architecture we consider is the interplay of three major domains, as depicted in Fig. 1:

- *M*odel concepts : the modelling domain focuses on graphical modelling concepts and corresponding domain specific languages (DSL);
- *C*loud aspects : the cloud platform domain focuses on the development and deployment of tools necessary to concretize and use the models; and
- *I*oT device aspects : the device platform domain focuses on device registration, business-logic deployment and execution.

Note that including IoT device aspects in the architecture differs from typical cloud-based or fog-based approaches, in which low-end IoT devices are not reprogrammable on the fly, and are not delegate any significant intelligence to pre-process data etc.

A. Sequential Overview

The approach we propose articulates the following building blocks, sequentially.

Automatic IoT Device Registration – Upon booting, each device from the managed fleet of IoT devices registers to a server, which feeds registered device information to the cloud platform providing an Integrated Development Environment (IDE) as described below.

Cloud-based IDE – Using the online IDE, a user can graphically program the business-logic to be deployed and executed on any IoT device managed by the platform. The IDE provides a code-free environment to define smart device behavior. In our prototype, we used the BPMN 2.0 (Business Process Model and Notation) modelling standard.

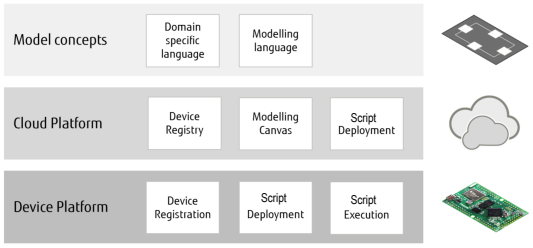

Fig. 1: Interplay of domains.

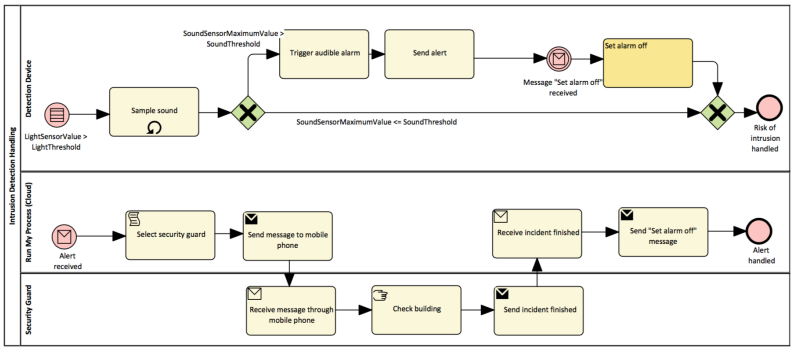

Fig. 2: Overall intrusion detection process.

From diagrams in cloud to code – The platform feeds the logic defined graphically in the IDE to a Domain Specific Language (DSL) server, using an XML-based format describing the interaction specified by the BPMN diagrams, including a description for the execution of its elements (graphical notation, process elements organization, interchange format etc.).

From code to embedded code – The engine running on the DSL server automates the translation of the provided XML into standard JavaScript, leveraging an API provided by RIOT, described further in Section V.

Embedded code deployment – The JavaScript file produced is transferred through an inbound gateway which can proxy HTTP and CoAP on one hand, and on the other hand IPv6 and IPv4, in order to cover most connectivity scenarios. The whole sequence from cloud IDE graphical programming, to JavaScript generation and deployment is described in Fig. 3.

Embedded code execution – The JavaScript logic is then executed on the IoT device and can (i) interact locally with the sensors/actuators, (ii) pre-process data extracted from sensors, and (iii) interact remotely with the cloud component to signal alarms or receive messages. More details on a specific interaction use-case are given on Section III.

III. Building Automation IoT Scenario

In the following, we focus on a concrete building automation use-case, for which we built a prototype described in the remainder of this paper. Note that we chose this use-case only for practical, proof-of-concept purposes. The approach we present is generic in that it applies to a wide variety of other IoT use cases.

Low-end IoT devices are managed from a remote ”cloud-based” component, to which they connect via a gateway and register to at boot time. IoT devices monitor light and sound level in their physical vicinity, for surveillance purposes. Specifically, if a device detects an abnormal level of light, it monitors the sound level. If the device detects an abnormal sound level in addition to abnormal light, it both triggers an audible alarm and signals the incident via the network, to the cloud component. Upon such signaling, a security guard is alerted on his mobile phone. After dealing on-site with the alarm, the guard confirms the incident is resolved, upon which the cloud component triggers the IoT device to switch off the alarm. The process modelling this scenario is depicted in Fig. 2.

Note how, in such scenarios, the logic on the IoT devices cannot be entirely determined in advance. On one hand, the fine-tuning of the light and sound level thresholds on each device typically need to be calibrated *after* the deployment, at commissioning time. On the other hand, the processing of raw sensor data may need to be moved from the devices to the gateway, or to the cloud component. Furthermore, devices may need to be removed/added to an existing system, and logic on legacy devices must be updated accordingly.

IV. Dynamically Scripting IoT Devices

We assume that each IoT device runs a small operating system (see survey [19]) providing basic services such as scheduling, hardware abstraction to access sensors/actuators peripherals, basic crypto and network connectivity up to an equivalent of BSD socket.

Middleware – We first provide glue code binding a lightweight script engine (such as [17] [6]) supporting a standard script language. This middleware binds the key APIs of the OS with the script interpreter via a simple library added to the script engine. Based on this middleware, IoT device can execute scripts in standard language interacting with the IoT hardware, e.g. setting timers, reading sensor values, setting actuators values, and communicating over the network.

Container & Over-The-Air Scripting – Simultaneously we configure the OS to provision memory for a Web resource (a CoAP resource [28]) which hosts and exposes a placeholder for text, on which typical RESTful operations are possible.

Locally, on the IoT device, this Web resource contains the scripted logic of the application to be executed. Per construction, the script engine offers sandboxing properties for the application logic, thus providing some equivalent of a runtime software container.

Remotely, from the cloud component point of view (see Section III), this Web resource is then viewed as a container for the application logic to be deployed on the IoT device, with read/write access through standard CoAP messaging (PUT and GET requests).

Security – On one hand, the communication channel between the cloud component and the device (used to discover, read or write the container Web resource) is secured with

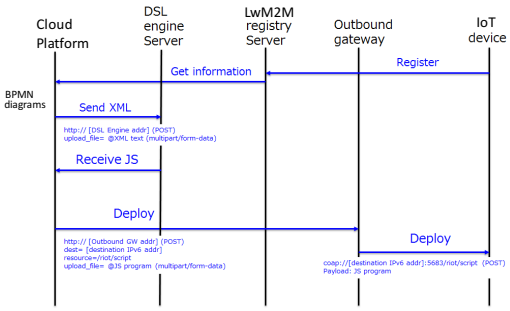

Fig. 3: Sequence towards automatic deployment of IoT business logic.

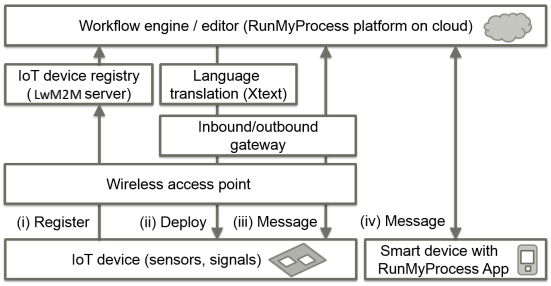

Fig. 4: Prototype components.

transport layer security and standard DTLS, which prevent eavesdropping, tampering, or message forgery.

On the other hand, the boiler-plate of the script includes a comment with the hash and cryptographic signature on the whole script (excluding this hash/signature comment). This hash/signature tuple is used to authenticate and authorize overwriting the current script with the new script.

Bootstrap & Device Registration – To bootstrap, we assume the simplest case of pre-shared keys. More advanced bootstrap alternatives beyond pre-shared keys are possible.

At commissioning time, basic registration of a container Web resource can be achieved via standard CoAP resource discovery at the cloud component. Optionally, advanced registration (e.g. detailed description of the capabilities of the device) is possible on top of CoAP signaling. For example, LWM2M registration [27] is a natural extension of the basic CoAP registration. Note however that alternatives for advanced registration are not limited to LWM2M.

A. Applicability & Trade-off

With the above architecture, from the vantage point provided by the cloud component, an operator can (i) discover the IoT devices which have registered and are currently available, (ii) access their container via the standard Web protocols, and (iii) push arbitrary scripts to containers, which are then start executing on the IoT device(s) which host these containers.

Generally, the architecture we described applies to most cases of remotely-managed fleets of low-end IoT devices – which include our target scenario (see Section III).The approach is agnostic w.r.t. communication technology used below IPv6 (6LoWPAN). Moreover, the approach is agnostic w.r.t. IoT device hardware, beyond basic support provided by the OS. Finally, the approach can achieve sandboxing of application logic on the IoT device, independently of the memory protection capabilities of the IoT hardware. For instance, we demonstrate in Section V this approach running on a CPU which does not have a Memory Protection Unit (MPU).

The key trade-off with this approach is the memory penalty incurred by the script engine, which is significant relatively to the total memory available on typical low-end IoT devices. One can however afford this penalty on many low-end IoT devices, as we show next in Section V. Moreover, for the smallest IoT devices that cannot afford this penalty, the advantage of this approach is that an OS aiming at very low memory footprint could freely (and happily) co-evolve with the middleware bundling we proposed, without compromising on memory overhead for these devices.

V. Prototype Implementation

In this section we present a prototype implementation of the architecture described in Section II. The components and their articulation are shown in Fig. 4.

A. Cloud Component

In the first phase, we emulated the cloud-based IDE with Copper [22], an add-on to the Firefox browser providing a CoAP client, from which we could conveniently push JavaScript logic to containers, as described below. In a second phase we used the cloud-based IDE provided by RunMyProcess [10], which we used to automatically generate JavaScript based on simple graphical programming, as described in Section II.

B. Connectivity Setup

We connected low-end IoT devices to the Internet via a RaspberryPi with a IEEE 802.15.4 radio module, configured in a standard fashion to act as a border router between plain IPv6 and the LoWPAN. The set up is shown in Fig. 5.

To register IoT devices to the cloud component, minimalistic LWM2M client functionality was implemented on top of the CoAP API provided by RIOT. The LWM2M client then registers to the LWM2M server (the latter was based on the Leshan implementation [1]). To implement the targeted scenario (see Section III) the LWM2M client function in RIOT just registers at bootstrap phase the resources 3/0, 9/0, 3303/0, and 3315/0, which map respectively to the IoT device, the .js container, the light sensor and the sound-level sensor, as per the OMA standard [27].

C. Low-end IoT Hardware Aspects

We assembled a low-end IoT device fulfilling the sensors/actuators requirements for the scenario targeted in Section III. From the hardware perspective, based the prototype on

a commercially available kit from Microchip, the SAMR21-xpro [5], which features a 32-bit micro-controller, 32kBytes of RAM and 256 kBytes of Flash memory, and a IEEE 802.15.4 radio transceiver. Note that the SAMR21 has no memory protection unit (MPU) in hardware. We extended the SAMR21 with a custom break-out board shown in Fig. 6, connecting via GPIO a light sensor, a sound level sensor, and a revolving light (total cost was about $5 using cheap components).

D. Embedded IoT Software Aspects

From the software perspective, we based our prototype on the open source operating system RIOT [14], because of its low memory footprint, its modularity and its matching with the prerequisites identified in Section IV. For the script engine, we chose to use JerryScript, a lightweight Javascript interpreter [4]. We then developed the middleware necessary to map JerryScript with the APIs offered by RIOT providing timers, sensor/actuator interaction, event callbacks and high-level networking (CoAP messaging). The resulting prototype RIOT Javascript API provided by this library is shown below in Listing 1.

Following the approach depicted in Section IV, we then configured RIOT to expose a container CoAP resource, that can be discovered and accessed remotely over the Internet (using the standard IPv6 protocol suite). When hosted in a container on a low-end IoT device with the necessary sensors/actuators, the code shown in Listing 2 (see Appendix) implements the behavior necessary to realize the alarm scenario described in Section III.

```
// Sensor & actuator access API
sensor = saul.get_by_name("NAME");
sensor = saul.get_one(TYPE);

// Sensor & actuator manipulation API
sensor.on_threshold(LEVEL, callback, FLANK);
sensor.read();
actuator.write(VALUE);

// Network access API
coap.register_handler(resource_name, COAP_METHOD,
    callback);
coap.request(url, COAP_METHOD, payload);

// Timer API & snippets
t = timer.setInterval(callback,
    interval_length_in_usec);
t = timer.setTimeout(callback, timeout_in_usec);
```

Listing 1: Prototype RIOT Javascript API

E. Evaluation & Discussion

First we verified the basic functionalities of our middleware on an M3 Open Node [12] available remotely on the IoT-lab testbed. Then, we verified the full functionalities of the alarm scenario defined in Section III on a SAMR21 in our office, using JavaScript code written by hand, pushed via Copper [22]. Finally, we pushed and generated JavaScript automatically from the cloud-based graphical programming IDE described in Section II. At runtime, from the cloud component interface, we pushed variations of the Javascript logic shown in Listing 2 (see Appendix), adapting the light and sound threshold parameters appropriately, so as to calibrate the sensitivity of the intrusion detection to the environment it was deployed into. We could verify that updates of the container logic were indeed received, installed and correctly executed on the IoT device.

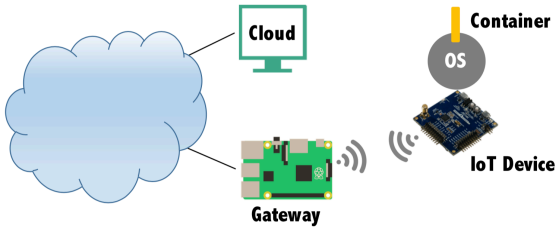

Fig. 5: Gateway connecting the cloud to a low-end IoT device, running an operating system hosting a container.

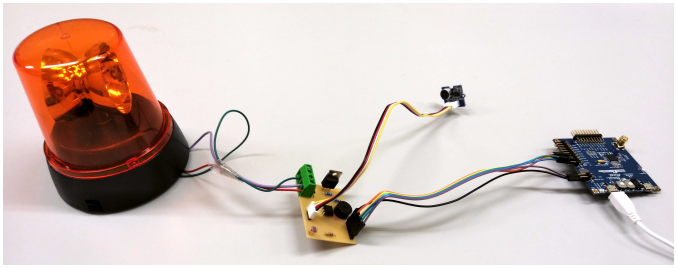

Fig. 6: SAMR21-xpro with custom break-out board connecting light sensor, sound level sensor, buzzer and revolving light.

We then coarsely measured both the memory necessary on the IoT device and the network traffic load incurred on the LoWPAN with this approach. On the SAMR21, the RAM usage was ≈27kB, split as follows: 8kB of heap and 4kB of stack for the Javascript engine, and 15kB for the rest of the OS including the network stack with CoAP and 6LoWPAN. The Flash usage was ≈220kB, roughly split as follows: 160kB for the Javascript engine, 60kB for the rest of the OS including the network stack.

We then assessed the generality of the approach: the code we developed for this prototype is not restricted to the SAMR21 board, but can be readily compiled and run on more than 84 different types of IoT devices, which corresponds to more than 80% of all the IoT hardware supported by RIOT.

Based on the above numbers, and compared to implementing in C similar functionality for the application logic, the memory overhead caused by JavaScript is thus ≈12kB of RAM and ≈160kB of Flash memory. However, in order to update the functionality dynamically on the device, a firmware update mechanism is necessary on top of the basic OS and the application logic. Taking the case of full firmware update, the RAM requirement would not be significantly impacted, but the required Flash memory increases significantly: typically twice the image size is needed (in our case 120kB for the OS including the network stack), with a small additional space for the bootloader (≈4kB is a conservative estimation). Hence, the total overhead is in fact ≈12kB of RAM and ≈96kB of

Flash memory, which is thus a significant penalty compared to an optimized C implementation. However, this is affordable if the IoT device has enough available memory: typical memory resources for such devices (e.g. 32kB of RAM and 256kB of Flash) are more than enough, as we have demonstrated.

On the other hand, in the case of the scenario defined in Section III, the number of bytes transmitted over-the-air to update the device with our container is 1kB (the size of the script shown in Listing 2 shown in Appendix). Compared to this, a full firmware update approach requires the transmission of an entire image, thus ≈60kB, which is significant penalty in comparison. However, with a partial firmware update technique, this penalty could be reduced.

F. Prototype Reproducibility

The simplest way to play with a prototype similar to the one described in the previous section is to remotely use the IoT-Lab testbed [12] as described in this guide [2]. To reproduce exactly the prototype including the hardware, three main components are needed (i) one or more IoT device(s), (ii) a border router, (iii) global IPv6 connectivity from the border router to the rest of the Internet, and (iv) the online IdDE.

IoT device – The IoT device hardware can be reproduced by assembling a SAMR21-xpro board (commercialized by Microchip [5]) and a breakout board connected via GPIO with the elements listed in Fig. 7. The IoT device software can be reproduced by using RIOT configured as in [9], and specifying the custom target LwM2M server address and port. This software provides local IPv6 connectivity to the IoT device and hosts the JavaScript container, accessible via CoAP on top of IPv6.

Border router – The border router hardware can be reproduced with a Raspberry Pi on which is plugged an IEEE 802.15.4 wireless communication module from Openlabs [8]. The border router software can be reproduced by using Linux (Raspbian with a Linux Kernel 4.9 or newer) which supports IEEE802.15.4 and 6LoWPAN. This software enables the border router to route IPv6 (and thus CoAP) traffic from the LoWPAN to the IPv6 backbone (the Internet) via its other network interfaces (typically Ethernet or Wifi).

Global IPv6 connectivity – In most networks, IPv4 connectivity is provided natively, but not IPv6. If IPv6 is not natively supported, the Raspberry Pi must somehow provide IPv6 tunneling through IPv4. This can be configured using a free tunnel broker such as Hurricane Electric [3].

Online IDE – The cloud-based IDE we used to enable graphical programming and automatic Javascript code generation is provided by RunMyProcess [10]. A simplified version of the prototype (not using graphical programming) can be based on a simple browser client such as Copper [22].

Reference	Value	Footprint
J1	IO_Line	Pin_Headers:Pin_Header_Straight_1x06_Pitch2.54mm
R1	LDR03	Opto-Devices:Resistor_LDR_5x4.1_RM3
R2	10K	Resistors_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P10.16mm_Horizontal
J2	SEN12945P	Pin_Headers:Pin_Header_Straight_1x04_Pitch2.00mm
BZ1	Buzzer	Buzzers_Beepers:Buzzer_12x9.5RM7.6
U1	4N28	Housings_DIP:DIP-6_W7.62mm
R3	56	Resistors_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P10.16mm_Horizontal
R4	10K	Resistors_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P10.16mm_Horizontal
Q1	BUZ11	TO_SOT_Packages_THT:TO-220_Vertical
D1	1N4001	Diodes_THT:D_5W_P10.16mm_Horizontal
J3	1A Load	Connectors:bornier3

Fig. 7: Break-out board hardware elements.

VI. Related work

Updating software on a deployed IoT device is typically done via over-the-air firmware update. With this approach, logic is updated and recompiled remotely, to produce a whole new firmware image, which is then downloaded and booted by the device [11]. Another category of solution is partial firmware update, which include approaches such as dynamic loading of binary modules [16], or differential binary patching [21]. If the difference in the binary is small, such approaches bring significant savings can be achieved in terms of bits-over-the-air (desirable on low-power, low-throughput networks).

Modelling and Orchestration of IoT logic is evolving from early approaches based on static, centralized schemes, to more dynamic and more distributed techniques. Early static, centralized approaches include trigger-action programming service IFTTT ("If This, Then That" [29]). NodeRED [7] extends the concept of IFTTT with more advanced modelling and graphical programming, but still does so in a centralized fashion. Examples of more dynamic and more distributed approaches are swarmlets using actors programming for IoT scenarios [20] [25], tasklets distributed at compile-time across nodes running TinyOS [18], or techniques leveraging an information-centric paradigm to dynamically distribute IoT logic via named function networking [24]. Recent prior work in this domain also proposed Actinium [23], an approach using small, distributed runtime containers on computers proxying for low-end IoT devices, accessible as Web resources, and hosting JavaScript logic.

Small memory-footprint embedded programming with bytecode interpreters was explored in early work such as Maté capsules [26] on TinyOS. Recent advances have brought the availability of very small script interpreter engines such as JerryScript [17], or MicroPython [6]. In this paper, we thus explore the potential of orchestrating runtime containers of scripted logic on low-end IoT devices. To the best of our knowledge, the closest prior work is Actinium. Compared to Actinium, we eliminate the need for Web resource proxying, as runtime containers are running *directly* on the low-end IoT devices. From the modelling tools perspective, the closest work is NodeRED. Compared to NodeRED our platform is more holistic in the sense that it encompasses both embedded containers, deployment and modelling in the cloud. The javascript container described in this paper was previously presented in [13]. Contrary to [13], this paper (i) encompasses the description of the full architecture also encompassing IoT device behavior modelling, the cloud-based IDE and automatic JavaScript code generation, and (ii) includes full

details on how to reproduce the prototype.

VII. Conclusion

In this paper, we have demonstrated how the combination cloud-based IDE and scripting-over-the-air with runtime containers hosted on low-end IoT devices (i) provides a generic solution for dynamic deployment of application logic on such devices, (ii) requires less than 32kB of memory on the devices, and (ii) saves bits-over-the-air compared to firmware updates. On one hand, such an approach can bring DevOps to low-end IoT devices. On the other hand, IoT system designers can decide after deployment which part of the intelligence should remain in the cloud, and which part should be embedded in the low-end IoT devices themselves. Last but not least, programming low-end IoT devices with an online graphical IDE significantly lowers the bar for programmers without strong embedded skills.

References

[1] Eclipse Leshan. https://www.eclipse.org/leshan.

[2] Guide to RIOT.js on IoT-Lab. https://github.com/emmanuelsearch/RIOT/tree/riot.js.demo.iotlab/examples/javascript/.

[3] Hurricane Electric Tunnel Broker. https://tunnelbroker.net.

[4] JerryScript RIOT Package. https://github.com/RIOT-OS/RIOT/tree/master/pkg/jerryscript.

[5] Microchip Atmel ATSAMR21-xpro Board. http://www.atmel.com/tools/atsamr21-xpro.aspx.

[6] MicroPython. https://micropython.org/.

[7] NodeRED. https://nodered.org.

[8] OpenLabs IEEE 802.15.4 communication module. http://openlabs.co/OSHW/Raspberry-Pi-802.15.4-radio.

[9] RIOT.js on SAMR21. https://github.com/emmanuelsearch/RIOT/tree/riot.js.demo.ciot/examples/javascript.

[10] RunMyProcess. https://www.runmyprocess.com.

[11] F. J. Acosta Padilla et al. The Future of IoT Software Must be Updated. In *IAB Workshop on Internet of Things Software Update (IoTSU)*, 2016.

[12] C. Adjih et al. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *Proc. of IEEE WF-IoT*, December 2015.

[13] E. Baccelli et al. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE Percom, 2018*.

[14] E. Baccelli et al. RIOT OS: Towards an OS for the Internet of Things. In *32nd IEEE INFOCOM. Poster*, Turin, Italy, 2013. IEEE.

[15] C. Bormann et al. RFC 7228: Terminology for constrained node networks. IETF Request For Comments, May 2014.

[16] A. Dunkels et al. Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *ACM EWSN, 2006*.

[17] E. Gavrin et al. Ultra lightweight javascript engine for internet of things. In *ACM SIGPLAN, 2015*.

[18] O. Gnawali et al. The tenet architecture for tiered sensor networks. In *ACM SenSys*, pages 153–166. ACM, 2006.

[19] O. Hahm et al. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, 2016.

[20] R. Hiesgen et al. Embedded Actors-Towards distributed programming in the IoT. In *IEEE ICCE-Berlin, 2014*.

[21] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *IEEE SECON, 2004*.

[22] M. Kovatsch. Demo abstract: Human-coap interaction with copper. In *IEEE DCOSS, 2011*.

[23] M. Kovatsch et al. Actinium: A restful runtime container for scriptable internet of things applications. In *IEEE IoT, 2012*.

[24] M. Król and I. Psaras. NFaaS: Named function as a service. In *ACM ICN, 2017*.

[25] E. Latronico et al. A vision of swarmlets. *Internet Computing, 2015*.

[26] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

[27] S. Rao et al. Implementing LWM2M in constrained IoT devices. In *IEEE ICWiSe, 2015*.

[28] Z. Shelby et al. RFC 7252: Constrained Application Protocol (CoAP). *IETF Request For Comments, 2014*.

[29] B. Ur et al. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *ACM CHI, 2016*.

Appendix

```
var process_1 = new function() {

    this.brightness_1 = saul.get_by_name("brightness
        ");
    this.sound_1 = saul.get_by_name("sound");
    this.buzzer_1 = saul.get_by_name("buzzer");

    this.sound_level;

    var self =  this;

    this.start = function () {
        self.brightness_1.on_threshold(800.0, self.
            activity2);
    };

    this.activity2 = function () {
        self.sound_level = self.sound_1.sample(5000)
            ;
        self.split_xor3();
    };

    this.split_xor3 = function () {
        if (self.sound_level.max > 100.0) {
            self.activity5();
        } else {
            self.activity4();
        }
    };

    this.activity4 = function () {
    print('Light but no sound');
    };

    this.activity5 = function () {
        self.buzzer_1.write(100.0);
        self.activity7();
    };

    this.await7 = function () {
        var handler;
        var callback = function () {
            handler.cancel();
            self.activity8();
        }
        handler = coap.register_handler("/alarm",
            coap.method.PUT, callback);
        coap.request("coap://[2a05:d014:677:XXXX:
            YYYY:f713:6820:e17f]/coap", coap.method.
            POST, "ALARM!");
    };

    this.activity8 = function () {
        self.buzzer_1.write(0.0);
        print('Canceling alarm');
    };
}

process_1.start();
```

Listing 2: Auto-generated script for the alarm scenario