Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things

Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Acosta Padilla, Kaspar Schleiser, Ian Thomas

Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things

Emmanuel Baccelli*, Joerg Doerr§, Shinji Kikuchi†, Francisco Acosta Padilla*, Kaspar Schleiser*, Ian Thomas‡

*Inria
†FUJITSU LABORATORIES LTD.
‡RunMyProcess
§Fraunhofer IESE

***Abstract*—The Internet of Things (IoT) connects a variety of small devices, via gateways, to the cloud. Use-cases often require IoT devices to run logic that is not pre-determined before deployment, and that must be updated during the life-time of the device. In this paper, we explore the potential of over-the-air scripting and updatable runtime containers hosting application logic on heterogeneous low-end IoT devices. Based on RIOT and Javascript, we provide a proof-of-concept implementation of this approach for a building automation IoT scenario. A preliminary evaluation shows our prototype runs on common off-the-shelf low-end IoT hardware with as little as 32kB of memory.**

I. Introduction

A large part of the Internet of Things (IoT) will consist of interconnecting low-end devices, whose characteristics include very small memory capacity (a few kBytes [7]) and limited energy consumption (1000 times less than a RaspberryPi). IoT use-cases require the orchestration of different pieces of logic running concurrently on low-end IoT devices and elsewhere on the network (e.g. in the cloud) and communicating with one another.

In a number of use-cases, the logic that needs to run on low-end IoT devices is not known upfront, before deploying the device(s). For instance, some part of the logic (e.g. pre-processing of some data) may need to be transferred on demand, from the cloud to the device, for privacy or performance reasons. Another example is the fine-tuning of some parameters of the logic running on some device, which can only be done after the deployment (e.g. the sensitivity of a distributed alarm system on-site).

In such context, this paper presents a generic approach to host, run and update IoT application logic on heterogeneous low-end devices, using over-the-air scripting and small containers. Based on RIOT and Javascript, we provide a proof-of-concept implementation of this approach for a building automation IoT scenario, as well as a preliminary evaluation of this implementation running on common off-the-shelf low-end IoT hardware.

A. 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 [4]. Another category of solution is partial firmware update, which include approaches such as dynamic loading of binary modules [8], or differential binary patching [12]. 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).

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" [19]). Examples of more dynamic and more distributed approaches are swarmlets using actors programming for IoT scenarios [11] [16], or techniques leveraging an information-centric paradigm to dynamically distribute IoT logic via named function networking [15]. Recent prior work in this domain also proposed Actinium [14], 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 has seen recent advances with the availability of very small script interpreter engines such as JerryScript [9], or MicroPython [3]. 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.

II. Building Automation IoT Scenario

In the following, we focus on a concrete building automation use-case. 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.

The use-case we consider is shown in Fig. 1. 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.

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.

III. Scripting Over-The-Air Architecture

We assume that each IoT device runs a small operating system (see survey [10]) 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 [9] [3]) 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 [18]) 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 II), 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 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.

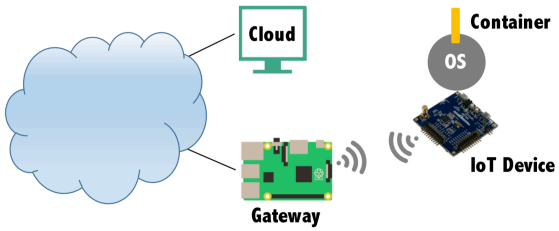

Fig. 1: Gateway connecting the cloud to a low-end IoT device, running an operating system hosting a container.

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 [17] 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 II).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 IV 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 IV. 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.

IV. Proof-of-Concept Implementation

In this section we present a prototype implementation of the architecture described in Section III.

A. Setup & Cloud Component Aspects

We connected low-end IoT devices to the Internet via a RaspberryPi with a IEEE 802.15.4 radio module (Openlabs rpi802154-r1), configured in a standard fashion to act as a border router between plain IPv6 and the LoWPAN. We emulated the cloud component with Copper [13], an add-on to the Firefox browser providing a CoAP client, from which we could conveniently push JavaScript logic to containers.

B. Low-end IoT Hardware Aspects

We assembled a low-end IoT device fulfilling the sensors/actuators requirements for the scenario targeted in Section II. From the hardware perspective, based the prototype on a commercially available kit from Microchip, the SAMR21-xpro [2], 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. 2, connecting via GPIO a light sensor, a sound level sensor, and a revolving light (total cost was about $5 using cheap components).

C. Embedded IoT Software Aspects

From the software perspective, we based our prototype on the open source operating system RIOT [6], because of its low memory footprint, its modularity and its matching with the prerequisites identified in Section III. For the script engine, we chose to use JerryScript, a lightweight Javascript interpreter [1]. 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.

```
// 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

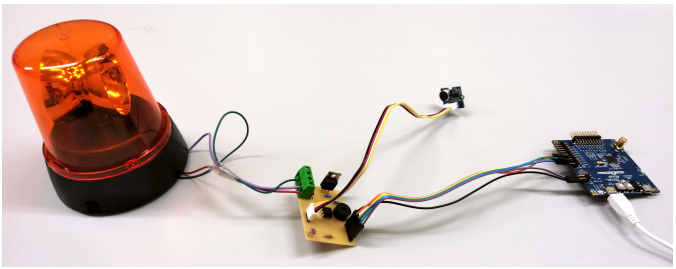

Fig. 2: SAMR21-xpro with custom break-out board connecting light sensor, sound level sensor, buzzer and revolving light.

Following the approach depicted in Section III, 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 below code (see Listing 2 in Appendix) implements the behavior necessary to realize the alarm scenario described in Section II.

D. Preliminary Evaluation & Discussion

First we verified the basic functionalities of our middleware on an M3 Open Node [5] available remotely on the IoT-lab testbed[1]. Then, we verified the full functionalities of the alarm scenario defined in Section II on a SAMR21 in our office. 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.

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 overhead 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

[1]Code and instructions to reproduce the test are available online at https://github.com/emmanuelsearch/RIOT/blob/riot.js.demo.iotlab/examples/javascript/

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 II, 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.

V. Conclusions, On-Going & Future Work

In this paper, we have demonstrated how the combination of scripting-over-the-air and 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, and (ii) saves bits-over-the-air compared to firmware updates. On one hand, this approach offers flexibility at runtime: one can transfer intelligence on-demand from the cloud, or the gateway, to the IoT device. On the other hand, programming the low-end IoT devices in Javascript significantly lowers the bar for programmers without strong embedded skills.

As an anecdote, while working on this paper, we actually used scripting-over-the-air and container logic to debug low-end IoT hardware: by pushing simple scripts in a container, we could quickly and accurately detect a faulty light sensor on one of our custom breakout boards. As a point of reference, on this category of hardware, LED-based debugging of software was the norm just a few years ago...

Our on-going work focuses on the cloud component side, where we hook a tool for graphical programming of business logic based on BPMN, with the automatic generation (via a DSL) and transfer (via HTTP and CoAP) of the corresponding Javascript to the IoT devices.

Future work on the IoT device side should include additional mechanisms for sandboxing containers, and support for multiple containers running simultaneously on a single IoT device.

References

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

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

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

[4] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix

```
this.brightness = saul.get_by_name("brightness");
this.sound = saul.get_by_name("sound");
this.buzzer = saul.get_by_name("buzzer");
this.sound_level;

var monitor_light = function () {
       brightness.on_threshold(800.0, monitor_sound
           );
   };

var monitor_sound = function () {
       sound_level = sound.sample(5000);
       if (sound_level.max > 800.0) {
           alarm_on();
       } else {
           false_alarm();
       }
   };

var false_alarm = function () {
   print("Light, but no sound");
   };

var alarm_on = function () {
       buzzer.set_value(30.0);
       coap.request("coap://[2a05:d014:XXXX:YYYY
           :9786:f713:6820:e17f]/coap", coap.method
           .POST, "ALARM!");
   };

var alarm_off = function () {
       print("Alarm turned off (via CoAP)");
       buzzer.set_value(0.0);
       return false;
       };

this.handler = coap.register_handler("/alarm", coap
    .method.PUT, alarm_off);

this.monitor_light();
```

Listing 2: Application script for the alarm scenario