Internet of Things 19 (2022) 100578

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Over-the-air firmware update for IoT devices on the wild[☆]

Maria Júlia Berriel de Sousa[a,*], Luis Fernando Gomez Gonzalez[a,b], Erick Mascagni Ferdinando[b], Juliana Freitag Borin[a]

[a] *Universidade Estadual de Campinas (UNICAMP), Campinas, Brazil*
[b] *Konker Labs, São Paulo, Brazil*

ARTICLE INFO

Keywords:
Internet of Things
IoT
Update over the air
OTA
RFC 9019

ABSTRACT

Internet of Things (IoT) has been gaining a lot of attention in the last few years and despite the large amount of investments, there is still much to be developed and refined. Software and firmware updates over-the-air (OTA) are a crucial part for IoT solutions longevity. This paper presents the design and implementation of an OTA firmware update method based on the standard architecture recently proposed by the Internet Engineering Task Force (IETF). The proposed solution is evaluated through a testbed using 20 constrained IoT devices and an open source IoT cloud platform, as well as on a set of devices deployed in a real world application. Results show that the proposed solution is suitable for constrained devices and has little impact on the network traffic.

1. Introduction

Internet of Things (IoT) is becoming a more common and more established field of information technologies as it is being applied in many areas, such as at home, in the industry, in cities, in agriculture and more [1]. Meanwhile, there is a vast amount of research devoted to IoT and its components, which include different parts of the whole IoT infrastructure: from the smart devices collecting data or interacting with the environment to the service that allows the data to be interacted with or even the method for transmitting that data.

Considering the IoT devices, a very important feature is the ability to update the software or firmware running on them. Updating allows for correction of bugs (specially security bugs) and to add new features to a device.

The possible large number of devices deployed and their, sometimes, hard to access locations, make the update process by physically reaching each one of them a costly and time-consuming task. To avoid that cost, the solution is to use *over-the-air* (OTA) updates, that can be performed remotely, leveraging the network to distribute the new code to the devices. Being such an important part of a device life cycle, software update brings its own challenges. Much of its hardships were described in previous works [2–4]. To advance in the direction of having a standardized firmware update method for devices with resource constraints, the IETF published the Request for Comment (RFC) 9019 defining a firmware update architecture for IoT [5].

In order to be accepted as a standard, RFC 9019 needs to be implemented and tested beyond proof of concept. This work presents the design and implementation of a solution for OTA firmware update based on the IETF architecture. The proposed solution is integrated to an open source IoT cloud platform and is evaluated in two different scenarios: (*i*) a testbed with up to 20 constrained

☆ This work is funded by Konker Labs, Brazil grant #34-P-04058-2018 and by the Brazilian National Council for Scientific and Technological Development (CNPq) grant #308840/2020-8.

* Corresponding author.

E-mail addresses: m117964@g.unicamp.br (M.J.B. de Sousa), luis@konkerlabs.com (L.F.G. Gonzalez), erick.ferdinando@konkerlabs.com (E.M. Ferdinando), jufborin@unicamp.br (J.F. Borin).

https://doi.org/10.1016/j.iot.2022.100578
Received 17 March 2022; Received in revised form 5 July 2022; Accepted 7 July 2022
Available online 11 July 2022

IoT devices in a Local Area Network (LAN) setting, aiming to assess the architecture applicability by testing it with real devices close to a scenario companies might encounter when implementing IoT solutions in their respective areas, and (*ii*) with devices already deployed and working as gateways to IoT devices in a Wide Area Network (WAN) setting for a restaurant chain. Results showed no significant impact of the update process on the network traffic or on the application running on the devices. They also highlight the difficulties of testing and deploying IoT solutions in a professional setting. To the best of the authors' knowledge, this is the first work to implement and test a firmware update solution based on the IETF architecture in multiple IoT devices in two different settings, one of them being a real world scenario.

The contributions of this paper are organized as follows: Section 2 discusses previous work on OTA software and firmware update; Section 4 presents the proposed solution for OTA firmware update; Section 5 describes the developed testbed and the methodology used to evaluate the solution; Section 6 assesses the performance of the proposed solution; and Section 7 presents conclusions and future work.

2. Related work

The key elements that constitute what is now called IoT were already in place when this technology started to become popular. For instance, Wireless Sensor Networks (WSN) similarly connected devices with sensors to a network. It was the coupling of cheaper manufacturing of devices and the spread of cloud computing that allowed for IoT to happen.

Brown and Sreenan [3] identified four main characteristics to classify the focus of the frameworks and protocols for software updates on WSNs: (i) dissemination of new code to the network; (ii) traffic reduction; (iii) the execution environment of an update in the device; (iv) fault detection and recovery. All of which are still relevant as research topics since they address common challenges in IoT environments such as limited processing power and storage, power efficiency, limited bandwidth, and security.

Wang et al. [2] discusses the main concerns of updating sensor nodes in a network. The complexity of the algorithms used to perform an update must be small enough to fit the device's memory and their limited processing capacity, they must also be energy efficient, guarantee the full delivery and correctness of the firmware, and must scale for number of nodes, as well as for node density.

After devices got connected to the Internet, it became possible to manage (which includes updating) an entire network remotely, usually from a platform, where data collected by devices is aggregated and the information about each device is kept. Management itself became an object of study. A solution based on the Simple Network Management Protocol (SNMP) is given by Silva et al. [6]. A standard for device management was developed by Open Mobile Alliance (OMA), called Lightweight Machine to Machine (LwM2M) [7], generating works studying its applicability [4,8]. Hernández-Ramos et al. [4] focus on device version control and managing dependencies to avoid mismatching versions. In Kolehmainen's work [8], the challenges presented stem from devices limitations and their connectivity and reachability. Both identify standardization as an important goal and propose solutions, the former based on the LwM2M standard and the latter using blockchain. Both also present a solution based on the standard proposed by IETF [5], which will be explained in more details in the next section.

The update process can be adapted to different needs, which resulted in several existing solutions, each making a different set of requirements a priority. Some are closed source, while others are open source. A comparison of several projects and their support for OTA updates is made by Villegas et al. [9]. For instance, a model based on the Constrained Application Protocol (CoAP) and on the Message Queuing Telemetry Transport (MQTT) protocol [10] recommends one or the other protocol, or a combination of both, depending on the type of update (i.e. software or security) and its severity.

There have been a few works implementing and testing different aspects of the IETF proposal. Besides the studies already mentioned [4,8,11], Ruckebusch et al. [12] show experimentally the intuition that the radio usage during updates is the most energy consuming task in a constrained device, even more in low bandwidth technologies. Cryptographic algorithms used for key management and decryption are evaluated in [11,13] and the conclusion is that even very constrained devices were capable of handling them. The work from Carlson [14] was entirely built upon the IETF draft architecture, showcasing its feasibility and flexibility. It also implements the same architecture, albeit from a version before it became an RFC, and their implementation uses Contiki-NG as the embedded operating system and perform tests only on a single board.

This paper will expand on this body of work, also building upon the IETF architecture, but using a more widely used protocol, Hypertext Transfer Protocol (HTTP). The reasoning being that it is easier to integrate it to the industry, that is accustomed to the Transmission Control Protocol/Internet Protocol (TCP/IP) stack. A message flow for distributing a firmware image to IoT devices is designed, implemented and integrated to an IoT cloud platform. As well, the developed solution is tested on a local network and on a wide area network, both with several devices operating.

3. IETF SUIT RFC 9019

The IETF Software Updates for Internet of Things (SUIT) working group published an internet draft proposing a standard architecture for the process of updating devices, as well as its requirements [5]. That draft became RFC 9019.

The architecture is based on the use of a manifest sent to devices with information about a new update. The information is used for authentication and authorization, where and how to obtain the new firmware, validation of the same, and several other requirements the process might have. This information can be considered update metadata. By focusing on the manifest, the proposed architecture can be flexible enough to be applied in many situations, but at the same time, robust with regard to functionality and security.

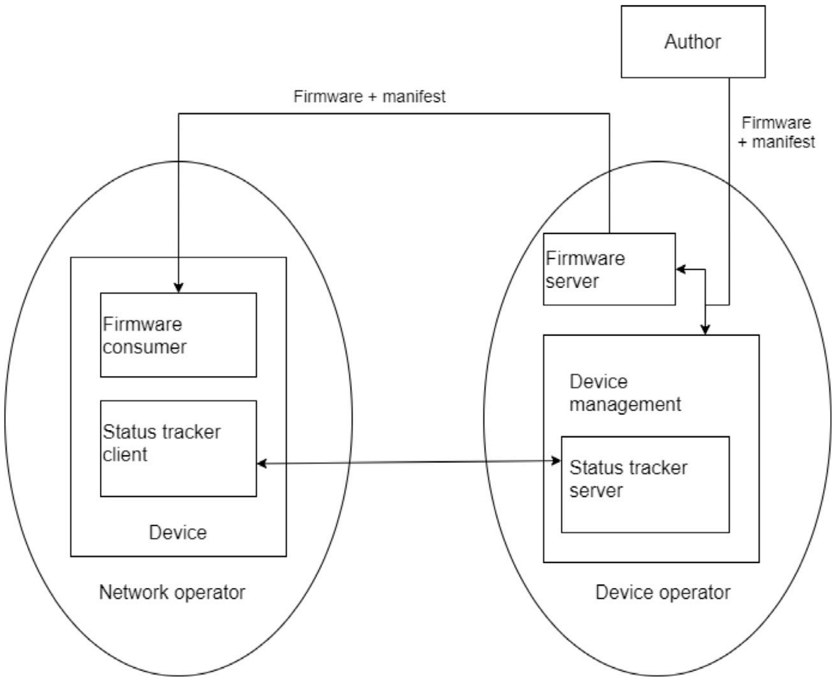


Fig. 1. OTA architecture.
Source: Recreated from [5].

The general architecture described by the RFC can be seen in Fig. 1. It consists of different entities that can interact with each other: the *Author* is responsible for creating a new firmware, uploading and notifying the device management platform; the *Device Operator* manages the IoT devices and can approve a new firmware, triggering the update using the status tracker; and the *Network Operator* is responsible for the devices network and can also interact with the status tracker.

The *Status tracker server* keeps software and hardware information about each device on the network and makes it available to the *Status tracker client*. Once the update is triggered by the status tracker, the firmware is uploaded to the *Firmware server*, making it accessible to the *Firmware consumer*. The update can be initiated by the status tracker client, be pushed by the status tracker server, or a hybrid of the two. Once initiated, the *Firmware consumer* downloads the manifest and the firmware, either separated or bundled together. After going through the steps of validating the firmware, it can be installed in the device by its bootloader.

The manifest is composed of a set of mandatory and optional fields. The mandatory fields are listed as follows:

Version ID Uniquely identifies the firmware version

Monotonic sequence number A monotonically increasing number

Payload format Describes the payload format for decoding

Storage location The memory location where to storage the new firmware

Payload digests One or more digests to ensure payload authenticity and integrity

Size The payload size in bytes

Signature A digital signature to verify the contents of the manifest

Dependencies A list of manifests required by the new one

Encryption wrapper A way for the device to obtain the key that decrypts the firmware

The optional fields allow for flexibility in implementation, in addition, some satisfy security requirements that can be desirable in a number of solutions.

4. IETF-based OTA firmware update design and implementation

This section presents the design and implementation of an OTA update process based on the RFC 9019 specifications and suitable for devices with constrained resources. The work includes the definition of the messages exchanged between the entities involved in the update process, as well as the design and implementation of the firmware for the IoT devices.

From the architecture presented in Fig. 1, the *Firmware Server* and the *Device Management* (which contains the *Status tracker server*) are both hosted in an IoT cloud platform, the same used to gather the data collected by the IoT devices, and from where the updates are sent. The IoT devices are the *Firmware consumers*, they track their own firmware information locally, acting as their own *Status tracker client*.

The implemented firmware provides functionality for communicating with the IoT cloud platform. All the communication related to the update process is performed using HTTP. The advantage of using HTTP for the update is that it allows for more bytes per packet, making the update process faster when compared to IoT protocols such as MQTT, especially for the firmware download. Additionally, for the purpose of easing the adoption of OTA updates by the industry at large, using a protocol network administrators are familiar with is a facilitator. As an example, HTTP is usually allowed through a company firewall, while MQTT or CoAP might not be.

Fig. 2 shows the proposed messages exchange. The IoT platform is capable of receiving and forwarding messages through a channel. This is the *_update* channel, which works as the communication interface between the platform and the devices for the update process.

The process starts after the *Author* has uploaded the firmware and the manifest to the IoT platform, with the manifest serving as the "new update" message. The device then polls the channel and retrieves the manifest. If it is correct, the device can request the firmware from the platform, following the hybrid approach to updating [5], mentioned in Section 3. After downloading the firmware, the device verifies it with the information from the manifest. If everything is correct, the device restarts and boots the new firmware. If any step fails, a message with the corresponding step is sent and the update process stops.

For this work, a subset of the required Manifest fields [15] was used, as can be seen in Fig. 3. The "version" field corresponds to *Version ID*, "sequence number", to *Monotonic Sequence Number*, and "checksum", to *Payload Digest*. The "device" field identifies the recipient of the manifest and works as a combination of *Vendor ID* and *Class ID* recommended fields in the IETF manifest draft.

Since the infrastructure to handle private and public keys was not implemented, the manifest is not signed, thus *Signature* and *Encryption Wrapper* were not needed. The *Size* field should be used to avoid denial of service attacks, since, at this time, we are not focusing on security issues, the field was omitted. The *Payload Format* and *Dependencies* fields do not influence this update, since the entire firmware is replaced. The *Storage Location* field is used by the bootloader, but given that the devices in our testbed had a preloaded bootloader which was not prepared to receive this data, it was not implemented.

Though it was not the focus of this work, some security problems are tackled. On the platform side, there is authentication and permissions handling for the *author*, only allowing an authorized person to initiate the process. On the device side, the device identification prevents the firmware to be downloaded by a device with a different hardware, or even a device with the same hardware, but for which the update is not intended. The *version* field prevents older (potentially insecure) versions from being installed, and the *sequence number* works in an analogous way, preventing older versions of the manifest (but for a newer firmware version) to be valid. Lastly, the *payload digest* stops an incorrect firmware to be installed when there is a valid manifest, this can happen by attacks where an insecure version of the firmware is sent, or by some communication error, the firmware is received incorrectly.

While the RFC was still a draft, it included a list of requirements for the OTA process. Despite being removed, the list provides a good way to analyze the implementation proposed in this work. Table 1 explains each item in the list, its description and how it was implemented.

5. Experimental evaluation

To validate the proposed solution based on the IETF architecture, including the manifest, and to evaluate its performance, two experiments were devised. For the first one, a testbed with constrained IoT devices was implemented on a LAN. The second one used a real scenario with a set of IoT devices distributed across a large geographic area including multiple cities and connected through a WAN. The main goals with these two different scenarios are threefold: (1) to evaluate the suitability of the proposed OTA solution for constrained IoT devices; (2) to evaluate the performance of the proposed OTA solution in different network settings; and (3) to evaluate the impact of performing the OTA update on devices running a real-world deployed application.

5.1. Environment and configuration

5.1.1. LAN evaluation

The experiments performed in a Local Area Network used WiFi communication provided by a TP-Link N750 router running OpenWrt, version 19.07.7.

The IoT devices were implemented using the NodeMCU 1.0 development board, with an ESP8266 WiFi chip (80 MHz, 4 MB of flash memory, and 64 kB of SRAM). In order to simulate an IoT application collecting sensor data, the IoT devices were configured to send a piece of data to an IoT cloud platform every 5 s using the MQTT protocol. The firmware, containing the OTA implementation,

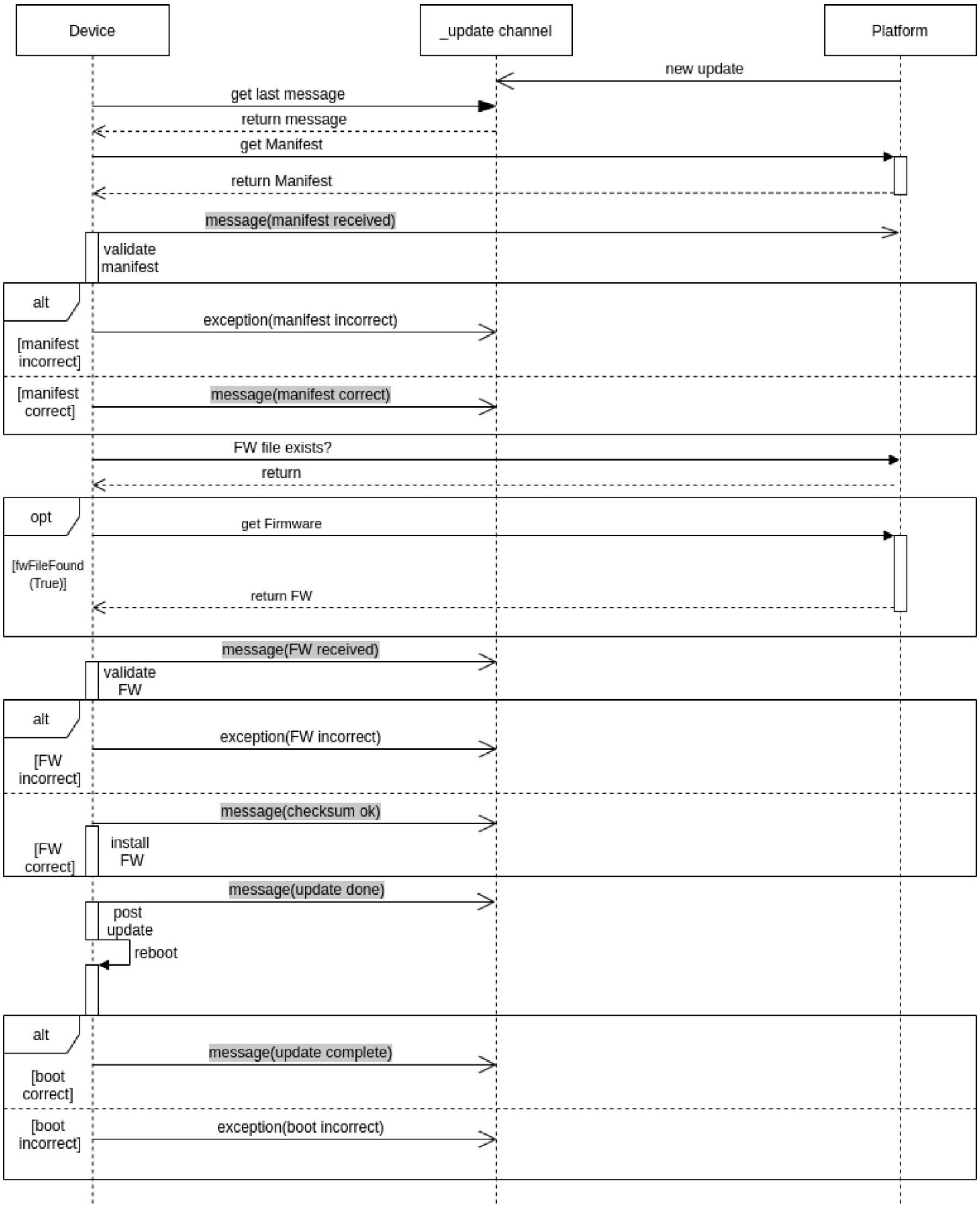


Fig. 2. Sequence diagram of messages exchanged during the OTA update process.

```
{
   "version": "0.0.0",
   "sequence_number": "9999999999999",
   "device": "node00",
   "checksum": "digest",
}
```

Fig. 3. Minimal manifest example.

Table 1
Comparing requirements proposed by IETF to current implementation.

IEFT Draft		Implementation
Requirement	Description	
Agnostic to how firmware images are distributed	The protocol or the way used for transmission should not prohibit the solution from working. This is both for firmware transmission and for the manifest.	HTTP over WiFi, but could be replaced
Friendly to broadcast delivery	In cases where broadcast is desired, it should cause the least disruption to the network distributing firmware and manifests. For this to happen, security must be applied to the manifest and firmware instead of transport layer or below. Also, devices should only accept manifests destined for them, even if it can receive manifests for other devices.	Devices are individually addressed, and the manifest has a field identifying the target device. Security implementation is listed in the next item.
Use state-of-the-art security mechanisms	Authentication for firmware and manifest author(s), integrity protection and confidentiality. Support for different cryptographic algorithms.	Device authenticates with user and password. FW checksum checks to platform and to device. Cryptography and digital signature remain for future work.
Rollback attacks must be prevented	The installation of an older firmware must be prevented, even with a valid manifest.	Device does not rollback firmware and uses a manifest field to validate this.
High reliability	Failure to validate any part of the update, or a power failure should not cause the device to be not operational	If state change messages are lost, functionality resumes as normal. Bootloader responsible for checking firmware integrity on boot.
Operate with a small bootloader	Bootloader should verify firmware before running it, and be able to rollback update in case it is not valid. It should not be updated, as this is a security risk.	Device bootloader is not updated by OTA. Does a CRC check at boot.
Small Parsers	The manifest parser should be minimal, to avoid possible bugs.	Manifest uses JSON format (described in Listing 3). *ArduinoJson* library used as parser.
Minimal impact on existing firmware formats	The firmware update mechanism must not require changes to existing firmware formats.	Proposed mechanism can be used independently of firmware format.
Robust permissions	Different entities (firmware author, device operator) should have their own permissions. A device can perform permission calculations, or trust a single entity to do so.	Only platform can perform update, by an authorized person.
Operating modes	Updates can be client initiated, server initiated or a hybrid, where each update step has a initiator.	Hybrid model, device polls server for new manifest, and installs after going through verification steps.
Suitability to software and personalization data	The manifest format should be extensible enough to allow for other forms of payloads.	Since manifest validation and firmware download are separate steps, the second could be changed to receive other forms of payloads.

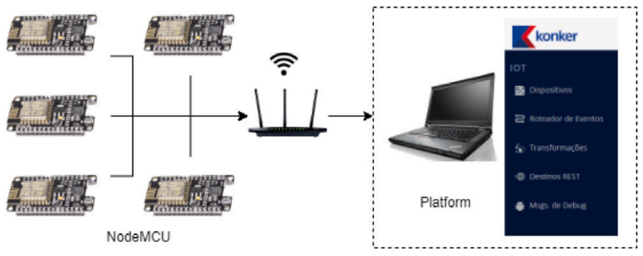


Fig. 4. LAN test setup.

was developed using the *Arduino* framework[1] for PlatformIO, in C++, and can be found in a GitHub repository.[2] The *Arduino* framework includes a library that implements an OTA update process, therefore, it was possible to reuse the implementation for the firmware download ("get Firmware" from Fig. 2) and for writing it to memory ("install FW"). The steps for checks and validations, and the messages exchanged were thought of, and implemented to comply with the RFC. The size of the new firmware was 390,288 bytes and the size of the manifest was 216 bytes. The RFC gives special attention to the bootloader. In the case of this implementation, the bootloader is provided by the *Arduino* framework, that already does the work of selecting the correct address to write the new firmware and boot correctly.

1 Version 2.6.2.

2 https://github.com/lmcad-unicamp/libKonkerESP.

Table 2
Distribution of devices by city.

City	# of devices
São Paulo	32
Guarulhos	7
Osasco	5
Santo André	4
Jundiai, Barueri, Sorocaba	3
Carapicuiba, São Jose dos Campos, São Bernardo do Campo	2
Itapevi, Diadema, Cajamar, Caraguatatuba, São Roque, Taboão da Serra, Franco da Rocha, Votorantim, Bragança Paulista, Itatiba, Mogi das Cruzes, Mauá	1

The IoT platform was hosted locally in a computer. Every element was in the same LAN, making it more suitable to evaluate the results. The experiment was performed using Konker IoT open source platform.[3] Fig. 4 shows an illustration of the implemented testbed.

5.1.2. *WAN evaluation*

In the WAN experiments, the devices used were Raspberry Pi 3, 1.2 GHz Quad Core processor, 1 GB of RAM and the firmware[4] was implemented using Python 3.[5] Of a total of 75 devices, most are spread across the city of São Paulo and its neighboring cities with a few across the state of São Paulo, in establishments from a restaurant chain. Table 2 presents the complete list of devices by city. The IoT platform used is the same, but in this case, it was hosted in a public cloud.

The devices used in this test are used as gateways for LoRAWAN end nodes. The end nodes are not updated in this experiment, nonetheless, their presence highlight the importance of having the update work correctly, as a faulty update would leave more devices affected beyond the ones involved in the OTA update process.

5.2. *Methodology of the experiments*

5.2.1. *LAN evaluation*

In total, 4 test scenarios were performed in the LAN experiment, each scenario with a different number of devices, starting with 5 devices and adding another 5 for each scenario, up to a total of 20 devices. All devices were running the same firmware and received the same update, but each one received a unique manifest. The maximum of 20 devices connected to the same Access Point (AP) does not overflow the network, but it is a number close to what can be found in real world LAN implementations of IoT solutions using WiFi communication. Even though an IoT solution can have many more devices, a single AP has a maximum number of devices it can handle.

Before and after the update, the devices were left running normally and collecting data, to simulate a device collecting data such as temperature or noise. This is so that the latency of the network could be measured, as well as any effects updating might have on data collection and transmission. While running, the devices collected health information, such as memory usage and WiFi signal strength.

These experiments combine data collected from the perspective of the network [16] and the device [11] to measure the update impact on the network and how it might influence the devices. The update process, from receiving the manifest to finish rebooting a new firmware, was divided in 7 steps. During each step, the device collected information about its resources usage and stored everything in memory to send to the platform after it boots to a new firmware.

The duration of each step was also registered. To do that and to keep time measurements precise, the computer running the platform was also hosting a Network Time Protocol (NTP) server, from where devices synchronized their internal clocks. Time starts counting when the manifests are sent (a "new update" from Fig. 2) and ends when all devices have sent the message ("update complete") confirming they restarted correctly. It is important to note that the timestamp for the "new update" message is the same for all devices, however, the devices may poll the *_update* channel in different instants of time, since this request is sent every 30 s and the devices are not synchronized. The messages used for timing the update steps are highlighted in Fig. 2 and correspond to the following:

i. *Polled* when the device requests the manifest from the platform;
ii. *Manifest received* when it is downloaded;
iii. *Manifest correct* after it is validated;
iv. *FW received* after firmware download;
v. *Cksum OK* when firmware is validated;
vi. *Done* after any post processing necessary;

[3] https://www.konkerlabs.com.
[4] https://github.com/lmcad-unicamp/update_rpi3.
[5] Linux kernel version 4.19 and Python version 3.5.3.

Table 3
Information sent during update process for 75 devices.

Step	Step	Resources usage
Polled	✓	✗
Manifest received	✗	✗
Manifest correct	✓	✓
FW received	✗	✗
Checksum ok	✓	✗
Update done	✗	✓
Update correct	✓	✓

vii. *Correct* after a device restart.

Another point to gather information was the computer running the platform. Wireshark[6] was left running to collect statistics from the network.

5.2.2. *WAN evaluation*

The WAN experiment also has different scenarios. In the first scenario, all 75 devices were updated, while in the second only 10 were updated. The most important difference is the firmware, the first scenario had an earlier version, where the messages from Fig. 2 were not in their final iteration. The steps are the same, but there are fewer messages. This caused the information collected to be different between both scenarios. There are two types of information for each step, a message with the *step* and a message with the *resource usage* of the device. Table 3 shows the difference between both, while the new firmware sent all of them, the older version sent the ones indicated in it with a check mark. In the second scenario, all the status messages for each step in Fig. 2 are sent, plus the device logs are also collected.

The difference in firmware version happened because even though this new version is tested, it could still cause a problem, and the procedure for getting the update approved involves more than just the *Author*, only a few devices had the newer version of the firmware updated. It is important to notice the update performed for these experiments were not in the main software, again, to avoid breaking anything. Another reason was enabling logging for a long time could damage the memory card, so this extra information was only enabled during testing and for a few devices. It is important to remember that these devices were running on a real world scenario and any problem could interfere on the service of one or more restaurants.

Each Raspberry Pi is powered through a wall socket, so energy saving is not the main concern. They can be connected to the internet via an Ethernet cable or WiFi, though it is not possible to identify which one at each point, and since each device is at a different location, with different connection conditions, collecting information about the impact in the network is not possible in the same way it was done in the LAN experiment. But it is possible to collect information about the devices and their connectivity conditions and gather how the later influence the first.

As part of the normal operation of these devices, they send statistics about their status every hour. These statistics include ping (from platform to device), device uptime, and the size of device data queue. This queue is for the data collected by the sensors, where data is sent from periodically, if sending fails for any reason, it is attempted again until it is sent successfully, so if it fails repeatedly, the queue size increases, for this reason this information can be used as a proxy for connection quality, as the bigger the size of the queue, the worse the quality of the connection is.

The goal of this experiment is, besides testing the feasibility of the implementation, to gather extra information about the impact of the update process in the operation of IoT devices. From the device perspective, the information collected in a similar vein as the nodeMCU, at each stage of the update. The information collected is: CPU utilization, free memory, the time passed since the last collection (in ms), CPU temperature and top 5 processes sorted by CPU utilization. These type of devices are not constrained as a nodeMCU, for example, they can continue collecting data in parallel with the update, and capable of running cryptographic algorithms that are not optimized for constrained devices.

6. Results and analysis

6.1. *LAN results*

Fig. 5 shows the average duration to execute each step of the update process for all devices in each test scenario. Since each device requests the manifest at a different time, but the starting moment for the process in the platform is the same for all devices, the first step is the longest and has the most variability, while the steps corresponding to downloading the firmware and rebooting are the next longer steps, as expected. The shortest steps only involve local processing in the device and no network communication.

Except from the first step, the other step's duration remains similar for all scenarios. It only increases with the number of devices during firmware download, as the amount of bytes sent increases linearly with the number of devices.

[6] https://www.wireshark.org/.

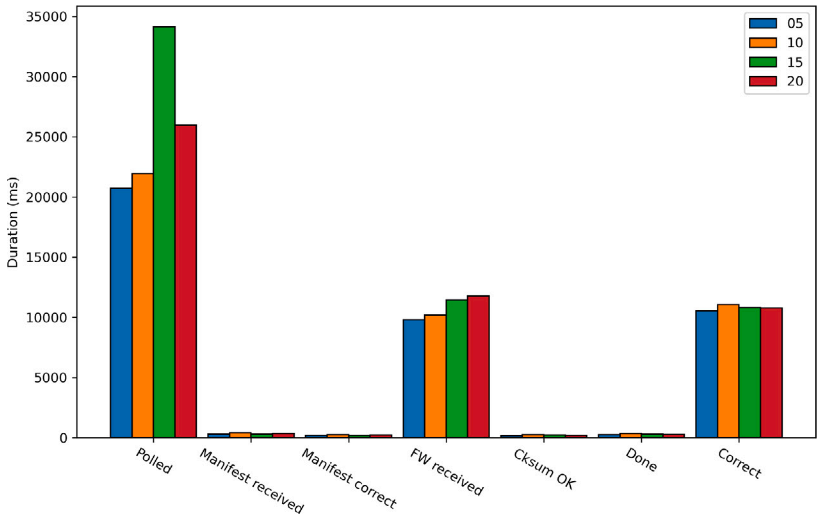


Fig. 5. Average duration of each update step for each scenario.

Table 4
Data collected from Wireshark — totals.

Metric	Test scenarios			
	05	10	15	20
Packets	13 248	26 693	38 525	51 074
Bytes	2 901 910	5 892 006	8 681 784	11 540 824
Test duration (s)	617.244	651.199	652.803	654.624

Table 5
Percentages relative to the total in Table 4.

	Scenario	Data type			
		Sensor data	Manifest	Firmware	Miscellaneous
Percent of packets	05	6.67	1.86	0.03	91.44
	10	7.66	2.03	0.03	90.28
	15	7.28	1.98	0.03	90.71
	20	7.33	1.95	0.03	90.69
Percent of bytes	05	2.16	0.91	67.25	29.68
	10	2.46	0.98	66.24	30.32
	15	2.28	0.95	67.43	29.34
	20	2.29	0.94	67.63	29.14

Test duration results presented in Table 4 show that the entire test duration for each scenario does not take longer than 11 min, but, as explained in Section 5.2, the devices were left running before and after the update, for 9 min, which means the update process, from sending the manifests to all devices to confirming everything is correct after rebooting, takes between 1 and 2 min for all scenarios of tests, that represents 12.5% of total test time for the first scenario and approximately 17% for the other three. There is only a small increment in that time, even when comparing 5 devices to 20, it increases only by 6.05%.

By the results presented in Table 5 it becomes clear that downloading the firmware is the most bandwidth consuming part of the test, being only 0.03% of the packets, but more than 65% of the transmitted bytes, compared to the collected "Sensor data" (MQTT) and "Manifest" (JavaScript Object Notation, JSON). Even in "Miscellaneous" data type, that includes data such as NTP requests, requests for the _*update* channel or devices sending health information, which were the vast majority of packets, it still represents fewer bytes than the firmware download.

The health information regarding the update (available memory and WiFi strength) was collected right before it began and right after it ends. Their available memory averages are presented in Fig. 6. Since all devices are running the same firmware, the memory available varies equally through test scenarios. After receiving a manifest, the free memory decreases a little after each step, but on aggregate, the memory needed to parse the manifest, download, and check a new firmware is no larger than 2 kB, making it efficient even for more constrained devices than the NodeMCU. The WiFi strength averages remained mostly the same for all test scenarios and at each step (the same in Fig. 6), varying between −45 dBm and −50 dBm. This shows that interference effects resulting from the increase in the number of devices could be dismissed when analyzing the results of these tests.

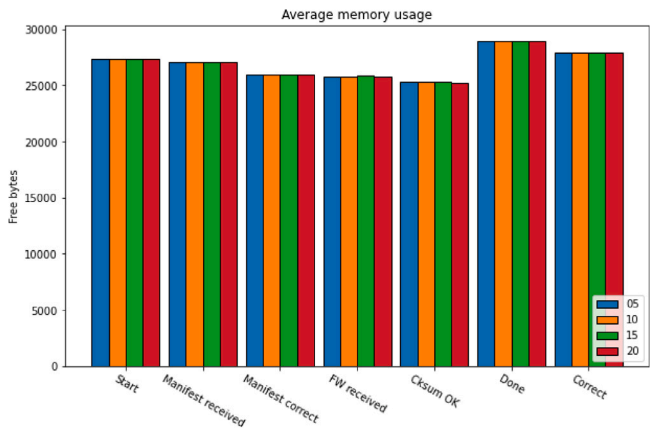


Fig. 6. Available memory during update.

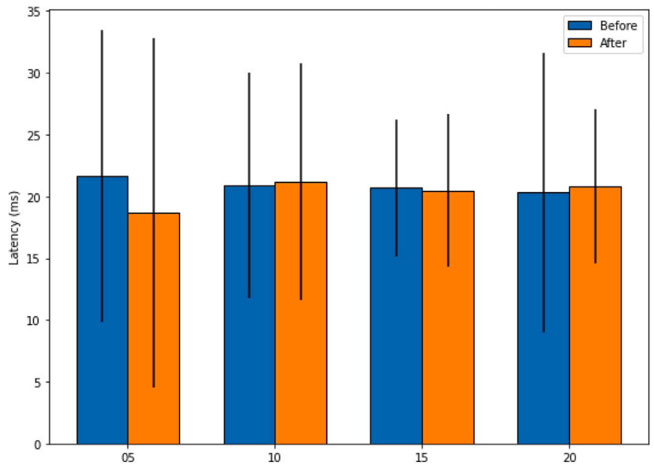


Fig. 7. Average message latency.

Considering the entire test (including the data collection), Fig. 7 shows the average latency for data transmission, which was measured for each MQTT message as the time it took for a message to be sent by the device and be received by the platform. The graph is separated by before and after the update, as they were not sent during the process. Except for the scenario with 5 devices, there is no significant difference caused by the update, which means that the impact of the process on the data transmission latency was not significant. In the 5 devices scenario, there is a decrease in the latency after the update that can be explained by the small number of devices, where a single device with a larger latency can influence the results and is evidenced by a larger standard deviation.

The most noticeable tendency in all the results is how increasing the number of devices did not cause an observable impact, be it on the network or on the devices themselves. With that, the proposed solution brings many advantages at a very little cost, in terms of infrastructure.

6.2. WAN results

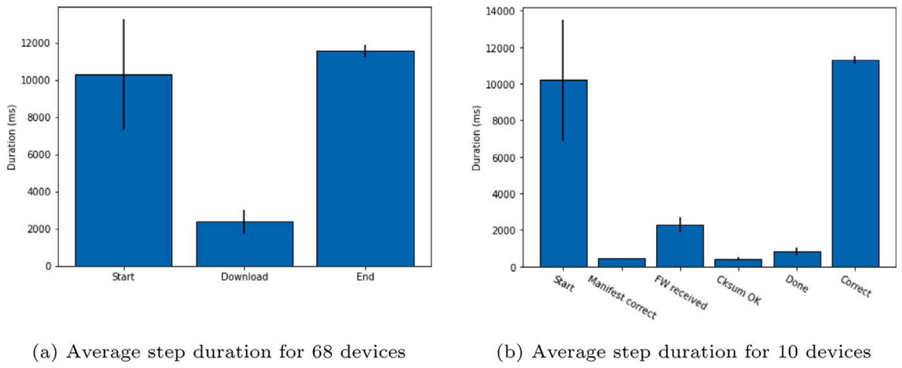
As explained in Section 5.2, in the first scenario, the update was sent to 75 devices. Since the *Network operator* has no control over whether the device is turned on or has connection to the internet, some of them have not responded to the manifest in the approximately 60 h that were given to the test. A few responded to the manifest, but failed somewhere during the process. The 10 devices for the second scenario were chosen in a manner that they all at least answered to the manifest.

Table 6 shows the number of devices in each outcome. In total, it was an 80% success rate for the first scenario and 70% for the second.

It was not possible to retrieve all the data described in Section 5.2 for all devices, of the 75, only 68 have complete data, and it is from them that the following results were extracted.

Table 6
Distribution of successes and failures.

Completed	No response	Download failure	Checksum failure	Other failure	Total
60	11	0	2	2	75
7	0	2	0	1	10



(a) Average step duration for 68 devices (b) Average step duration for 10 devices

Fig. 8. Average duration of each update step.

Similar to Figs. 5, 8(a) and 8(b) shows the average duration of each step. In the case of Fig. 8(a), since not all steps have a corresponding message, some steps were incorporated together. From Table 3, *Start* includes up until the manifest is correctly received, *Download* includes the firmware download and its verification, while *End* includes any post-processing and devices restart.

The step's duration distribution is also similar to Fig. 5, the biggest difference is the total duration, mostly due to the Raspberry devices querying the platform for the manifest every 10 s instead of 30, although the firmware download took about 5 times more time on average. The average total duration of the update for the first test was 24.24 s (standard deviation = 3.10) and for the second test was 25.45 s (standard deviation = 3.45).

From the scenario with 10 devices, the data about device behavior was captured. Regarding memory usage, it did not vary much through the update, which is expected since the update size is orders of magnitude smaller than the device memory. Meanwhile, the CPU utilization did show a significant increase after downloading the firmware, as seen in Fig. 9, during the process of extracting and copying the firmware to the correct location.

The two devices that failed to download the firmware are also shown. In both, there were two attempts at performing the update. While device 02 had a similar behavior in one attempt, up until it failed to retrieve it, device 03 had a higher CPU utilization in both tries. Despite that, the other data did not show many differences when compared to devices that finished the update. The availability of both on the day of the update was 28.6% added to the fact that they continued communicating with the platform after the failure, showed that the connection did not completely drop, but could be unstable. This could have caused the failure at the download step.

Device 09 also did not finish the update and failed in the same firmware download step, although in this case there was no exception message nor the subsequent data. For this reason, this device data is not shown. This indicates that what caused the update failure was a drop in the internet connection, even though its availability was 99.1% during the day.

Comparing to the first scenario, concerning these three devices, both 02 and 03 completed the update, but 09 failed in the checksum check.

The fact that in the experiments in a LAN controlled environment all devices updated without a problem, but the same did not happen in a more open one, shows the difficulties of deploying devices on the wild, where there is little or no control over the connection, or even if the device is turned on or off. Even collecting the information proved difficult when devices performed the update, but did not respond when requested for information later.

Despite this, the WAN experiments had a high success rate, and where it failed it was possible to identify the most likely cause with the messages (or lack thereof) from Fig. 2. This data can provide mitigation or solutions for these problems, helping improve the solution that is built on top of the RFC architecture.

7. Conclusion

This paper proposed an OTA firmware update solution for constrained IoT devices based on the IETF standard architecture. The proposal included the design of the message flow among the involved entities. The proposed solution was evaluated in two different scenarios: one testbed including up to 20 IoT devices in a LAN setting and another with 75 devices deployed in different cities as part of a real application, monitoring restaurants. Results show that it is feasible to implement this solution in real world scenarios and that it does not interfere with the application running on the infrastructure.

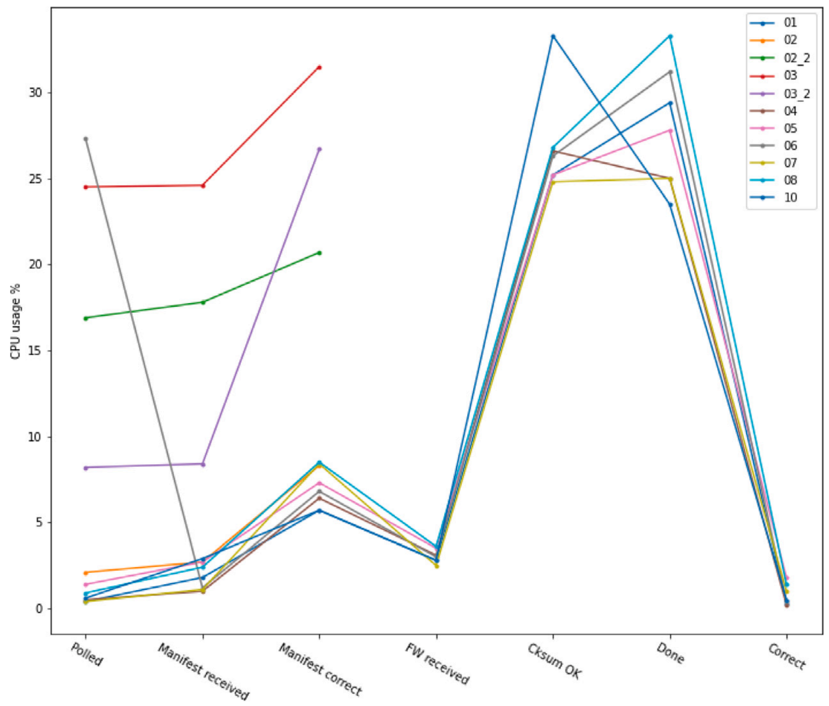


Fig. 9. CPU utilization during the update for 10 devices.

Even though the use of the RFC 9019 architecture greatly improved the updating process of the remote devices, which was previously done by reaching physically each one of them, there is still room for improvement. The most straightforward in terms of implementation is a retrial system for updates that responded to the manifest, but failed during the update. Doing so automatically would further automate the management of the devices.

One of the main aspects that could be improved in this work is the security features, for example, signing the manifest and encrypting the firmware or using HTTPS. Since the architecture from the RFC 9019 was designed with security in mind, including such features would not require any significant change in the implementation. Although more testing would be required to measure how it impacts the implementation, mainly compared to the results presented in this work.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Juliana Freitag Borin reports financial support and equipment, drugs, or supplies were provided by Inmetrics.

References

[1] E. Ahmed, I. Yaqoob, A. Gani, M. Imran, M. Guizani, Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges, IEEE Wirel. Commun. 23 (5) (2016) 10–16, http://dx.doi.org/10.1109/MWC.2016.7721736.

[2] Qiang Wang, Yaoyao Zhu, Liang Cheng, Reprogramming wireless sensor networks: challenges and approaches, IEEE Netw. 20 (3) (2006) 48–55, http://dx.doi.org/10.1109/MNET.2006.1637932, URL http://ieeexplore.ieee.org/document/1637932/.

[3] S. Brown, C. Sreenan, Software updating in wireless sensor networks: A survey and lacunae, J. Sens. Actuat. Netw. 2 (4) (2013) 717–760, http://dx.doi.org/10.3390/jsan2040717.

[4] J.L. Hernández-Ramos, G. Baldini, S.N. Matheu, A. Skarmeta, Updating IoT devices: challenges and potential approaches, in: 2020 Global Internet of Things Summit (GIoTS), 2020, pp. 1–5, http://dx.doi.org/10.1109/GIOTS49054.2020.9119514, URL https://ieeexplore.ieee.org/abstract/document/9119514.

[5] Brendan Moran, D. Brown, M. Meriac, H. Tschofenig, A Firmware Update Architecture for Internet of Things, Request for Comment 9019, 2021, http://dx.doi.org/10.17487/RFC9019, URL https://www.rfc-editor.org/rfc/rfc9019.html.

[6] J.d.C. Silva, M.L. Proença Jr., J.J.P.C. Rodrigues, IoT network management: Content and analysis, 2017, XXXV SIMPÓSIO BRASILEIRO de TELECOMUNICAÇÕES E PROCESSAMENTO de SINAIS.

[7] Open Mobile Alliance, Open Mobile Alliance - LightweightM2M Overview. URL http://www.openmobilealliance.org/wp/Overviews/lightweightm2m_overview.html.

[8] A. Kolehmainen, Secure firmware updates for IoT: A survey, in: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, Halifax, NS, Canada, 2018, pp. 112–117, http://dx.doi.org/10.1109/Cybermatics_2018.2018.00051, URL https://ieeexplore.ieee.org/document/8726545/.

[9] M.M. Villegas, C. Orellana, H. Astudillo, A study of over-the-air (OTA) update systems for CPS and IoT operating systems, in: Proceedings of the 13th European Conference on Software Architecture - ECSA '19 - Vol. 2, ACM Press, Paris, France, 2019, pp. 269–272, http://dx.doi.org/10.1145/3344948.3344972, URL http://dl.acm.org/citation.cfm?doid=3344948.3344972.

[10] A. Thantharate, C. Beard, P. Kankariya, CoAP and MQTT based models to deliver software and security updates to IoT devices over the air, in: 2019 International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, Atlanta, GA, USA, 2019, pp. 1065–1070, http://dx.doi.org/10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00183, URL https://ieeexplore.ieee.org/document/8875289/.

[11] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, E. Baccelli, Secure firmware updates for constrained IoT devices using open standards: A reality check, IEEE Access 7 (2019) 71907–71920, http://dx.doi.org/10.1109/ACCESS.2019.2919760, URL https://www.researchgate.net/publication/333472928_Secure_Firmware_Updates_for_Constrained_IoT_Devices_Using_Open_Standards_A_Reality_Check. Conference Name: IEEE Access.

[12] P. Ruckebusch, S. Giannoulis, I. Moerman, J. Hoebeke, E. De Poorter, Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g, Internet of Things 3–4 (2018) 104–119, http://dx.doi.org/10.1016/j.iot.2018.09.010, URL https://linkinghub.elsevier.com/retrieve/pii/S2542660518300362.

[13] H. Gupta, P.C. van Oorschot, Onboarding and software update architecture for IoT devices, in: 2019 17th International Conference on Privacy, Security and Trust (PST), 2019, pp. 1–11, http://dx.doi.org/10.1109/PST47121.2019.8949023, URL https://ieeexplore.ieee.org/document/8949023.

[14] S. Carlson, An Internet of Things Software and Firmware Update Architecture Based on the SUIT Specification (Master's thesis), KTH Royal Institute of Technology, Stockholm, Sweden, 2019, URL http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1330148&dswid=-1503.

[15] B. Moran, H. Birkholz, H. Tschofenig, An information model for firmware updates in IoT devices, 2020, URL https://tools.ietf.org/html/draft-ietf-suit-information-model-04. Library Catalog: tools.ietf.org.

[16] S. Unterschutz, V. Turau, Fail-safe over-the-air programming and error recovery in wireless networks, Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems (2012) 27–32.