
Universidade Estadual de Campinas
Instituto de Computação


Maria Júlia Berriel de Sousa

# Design and Evaluation of a Method for Over-The-Air Firmware Updates for IoT Devices

# Projeto e Avaliação de um Método para Atualizações *Over-the-Air* para Dispositivos *IoT*

CAMPINAS
2022

**Maria Júlia Berriel de Sousa**

# Design and Evaluation of a Method for Over-The-Air Firmware Updates for IoT Devices

# Projeto e Avaliação de um Método para Atualizações *Over-the-Air* para Dispositivos *IoT*

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

**Supervisor/Orientadora: Profa. Dra. Juliana Freitag Borin**

Este exemplar corresponde à versão final da Dissertação defendida por Maria Júlia Berriel de Sousa e orientada pela Profa. Dra. Juliana Freitag Borin.

CAMPINAS
2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Sousa, Maria Júlia Berriel de, 1991-
M337d Design and evaluation of a method for Over-the-Air firmware updates for IoT devices / Maria Júlia Berriel de Sousa. – Campinas, SP : [s.n.], 2022.

Orientador: Juliana Freitag Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Internet das coisas. 2. Software embarcado. 3. Dispositivos de internet embarcados. I. Borin, Juliana Freitag, 1978-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações Complementares

**Título em outro idioma:** Projeto e avaliação de um método para atualizações Over-the-Air para dispositivos IoT
**Palavras-chave em inglês:**
Internet of things
Embedded software
Embedded Internet devices
**Área de concentração:** Ciência da Computação
**Titulação:** Mestra em Ciência da Computação
**Banca examinadora:**
Juliana Freitag Borin [Orientador]
Hana Karina Salles Rubinsztejn
Islene Calciolari Garcia
**Data de defesa:** 12-12-2022
**Programa de Pós-Graduação:** Ciência da Computação

**Identificação e informações acadêmicas do(a) aluno(a)**
- ORCID do autor: https://orcid.org/0000-0001-5259-7281
- Currículo Lattes do autor: http://lattes.cnpq.br/5454342231475151


**Universidade Estadual de Campinas**
**Instituto de Computação**


**Maria Júlia Berriel de Sousa**

# Design and Evaluation of a Method for Over-The-Air Firmware Updates for IoT Devices

# Projeto e Avaliação de um Método para Atualizações *Over-the-Air* para Dispositivos *IoT*

**Banca Examinadora:**

- Profa. Dra. Juliana Freitag Borin
  Universidade Estadual de Campinas
- Prof. Dra. Hana Karina Salles Rubinsztejn
  Universidade Federal do Mato Grosso do Sul
- Profa. Dra. Islene Calciolari Garcia
  Universidade Estadual de Campinas

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 12 de dezembro de 2022

# Acknowledgments

First, I would like to thank my family, Luiza, Jadyr and Elisa, for always supporting me and providing the best education they could, that allowed me to get where I am today. I would also like to thank my girlfriend, Beatriz, for the years of companionship while I researched and wrote this dissertation.

Next, I would to thank my supervisor, Prof. Dr. Juliana Borin, for constantly providing guidance in my research and having a healthy environment for her students. My thanks extend to Luis Gonzalez and Erick Ferdinando for the help with data collection and the weekly feedback.

I am grateful to the many friends and colleagues I met during my years at Unicamp, but specially my friends in CEM and LAU futsal teams. They certainly made all these years a lot more enjoyable.

Finally, I am thankful to Konker (Inmetrics S.A., grant #34-P-04058-2018) for providing financial and material support to my research. Special thanks to Alexandre Junqueira for also helping with software development.

# Resumo

A área de Internet das Coisas (IoT, em inglês) vem ganhando atenção nos últimos anos, e apesar da grande quantidade de investimentos, ainda há muitos desafios a serem superados em todas as partes da arquitetura. Um desafio é desenvolver uma aplicação duradoura. Com relação aos dispositivos, um dos fatores que mais contribui para a longevidade é a habilidade de atualizar o firmware dos dispositivos. Em termos práticos, atualização de firmware ou software *over-the-air* (OTA) é uma parte crucial da arquitetura de IoT, junto com a capacidade de gerenciá-los remotamente.

Essa dissertação explica a arquitetura de IoT através de uma visualização em camadas e como atualizações OTA são parte dela. Como uma solução para esse problema, um método para atualização de firmware OTA, e a sua implementação é apresentado. Esse método é baseado no padrão recentemente proposto pela Internet Engineering Task Force (IETF), na forma do RFC 9019. A solução proposta é avaliada através de dois cenários de teste: um com 20 dispositivos IoT restritos conectados a uma plataforma IoT na nuvem e de código aberto, através de uma rede WiFi, o outro cenário consiste em um conjunto de 75 dispositivos implantados espalhados por uma grande área geográfica como parte de uma aplicação real. Resultados mostram que a solução apresentada é adequada para dispositivos restritos a tem pouco impacto no tráfego de rede.

# 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 are still many challenges to be overcome within all parts of IoT architecture. One challenge is how to develop a long-lasting application. With regard to devices, one of the main contributors to longevity is the ability to update devices firmware. In practical terms, software and firmware updates over-the-air (OTA) are a crucial part of IoT architecture, along with the capacity to manage them remotely.

This dissertation explains the IoT architecture with a layered approach and how updates OTA are part of it. As a solution to this problem, the design and implementation of an OTA firmware update method is presented. The method is based on the standard architecture recently proposed by the Internet Engineering Task Force (IETF), in the form of RFC 9019. The proposed solution is evaluated through two testbeds: one with 20 constrained IoT devices connected to an open source IoT cloud platform through a WiFi network and the other one with a set of 75 devices spread in a large geographic area as part of a real world application. Results show that the proposed solution is suitable for constrained devices and has little impact on the network traffic.

# List of Figures

# List of Tables

# List of Abbreviations and Acronyms

| | |
|---|---|
| 3GPP | 3rd generation Partnership Program |
| AMQP | Advanced Message Queuing Protocol |
| AP | Access Point |
| API | Application Programming Interface |
| BLE | Bluetooth Low Energy |
| CBOR | Concise Binary Object Representation |
| CCTV | Closed Circuit Television |
| CoAP | Constrained Application Protocol |
| CPU | Central Processing Unit |
| DDoS | Distributed Denial of Service |
| DTLS | Datagram Transport Layer Security |
| EEPROM | Electrically Erasable Programmable Read-only Memory |
| ECU | Electronic Control Units |
| FW | Firmware |
| GSM | Global System for Mobile Communications |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| IoT | Internet of Things |
| IP | Internet Protocol |
| ISP | Internet Service Provider |
| JSON | JavaScript Object Notation |
| LAN | Local Area Network |
| LTE | Long-Term Evolution |
| LPWAN | Low Power Wide Area Network |
| M2M | Machine to Machine |
| NTP | Network Time Protocol |
| NB-IoT | Narrow Band Internet of Things |
| OEM | Original Equipment Manufacturer |
| OMA | Open Mobile Alliance |
| OS | Operating System |
| OSI | Open Systems Interconnection |
| OTA | Over the Air |
| PAN | Personal Area Network |
| RAM | Random Access Memory |
| REST | Representational State Transfer |
| RFC | Request for Comment |

| | |
|---|---|
| RFID | Radio Frequency Identification |
| SNMP | Simple Network Management Protocol |
| SRAM | Static Random Access Memory |
| SSL | Secure Sockets Layer |
| SUIT | Software Updates for Internet of Things |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| UDP | User Datagram Protocol |
| URI | Universal Resource Identifier |
| WAN | Wide Area Network |
| WPAN | Wireless Personal Area Network |
| WSN | Wireless Sensor Network |
| WWAN | Wireless Wide Area Networks |
| XML | Extensible Markup Language |
| XMPP | Extensible Messaging and Presence Protocol |

# Contents

# Chapter 1

# Introduction

The term *Internet of Things* (IoT) was coined by Kevin Ashton[1] in 1999, but it only started to gain popularity in the last decade. Since then, it has become a more established area of computing and much has been researched and developed in order to overcome the challenges of the field.

There isn't a single adopted definition for IoT, but most of them include the existence of sensors and actuators (the *Things*) connected to the Internet or to each other, forming large networks of devices. And some form of intelligence - most of the time computers, rather than human - extracting information from the data collected and acting on it. This means that everyday objects now become *smart* and are capable of doing more than their original purpose.

The increased interest in the IoT topic is exemplified in Figure 1.1, a notable rise from 2014 onward. The number of devices is projected to grow exponentially in the next years. According to Ericsson Mobility Report on IoT connections outlook [2], the number of connections is expected to go from 12.6 billion in 2020 to 26.9 billion in 2026.

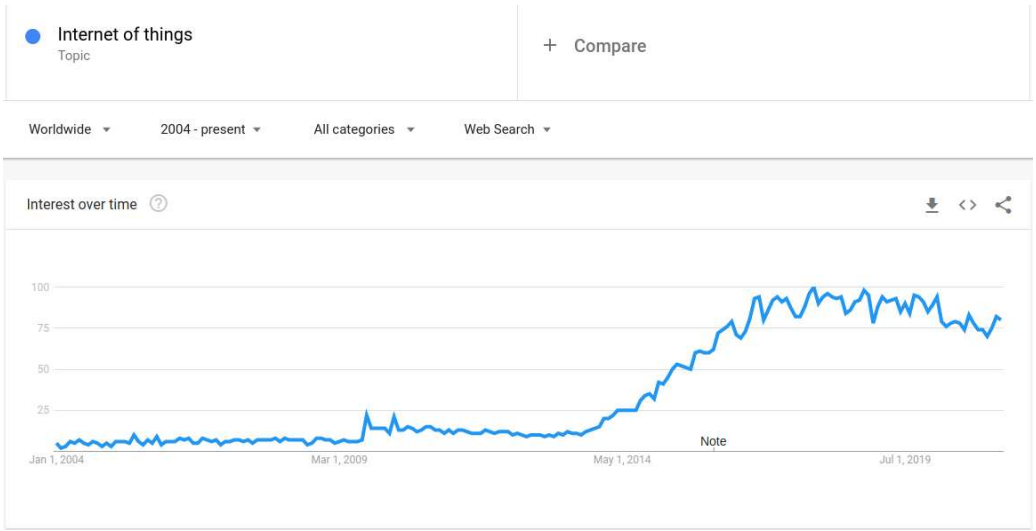



Figure 1.1: Interest in the topic IoT over time, from Google Trends

This kind of technology has the potential to be applied to a variety of areas [3, 4, 5],

[1] https://iot-analytics.com/internet-of-things-definition/

some of which are:

i. *Smart cities:* monitoring traffic, air and water quality, efficient management of city lights, planning city structure maintenance;
ii. *Smart grid:* monitoring power lines, buildings, areas subject to volcanic activity, flooding or landsliding in order to prevent disasters due to natural conditions;
iii. *Smart transportation:* route planning inside a city or between cities, fleet control, public transport passenger counting and ticketing;
iv. *Smart health:* monitoring health parameters, to prevent diseases or monitor them, improve quality of life, and provide better elderly care;
v. *Smart industry:* monitoring and controlling factories. To avoid accidents, improve productivity, reduce energy consumption, and control people access;
vi. *Smart agriculture:* monitor the soil and air conditions, more fine-grained productivity control;
vii. *Smart home:* reduce energy consumption, home safety, monitoring and controlling electrical appliances;

All of these applications require numerous devices to be deployed that will operate without human intervention for a long time. In the same way as computers and smartphones, these devices will need to have their software updated to maintain its functionality. The benefits of updating device's firmware are summarized in:

- correcting functionality bugs or security ones;
- adding new features that augment the device functionality;
- improving performance by optimization;
- extending device's useful life;
- saving time and costs of replacing one or more entire device;
- environmental friendly, by avoiding generating *e-waste*.

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, with minimal human interaction, 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 [6, 7, 8] with possible solutions discussed by other authors [9, 10, 11]. Most of these solutions are proposed or implemented using technologies developed for constrained devices that are incorporated with well established Internet technologies.

These works test the impact of updating the device, but very few consider how the network performance is affected by an update, e.g., if data is lost, how much, if any, downtime. Furthermore, each work that proposes a solution creates its own framework that satisfies the authors priorities, and compatibility with other works is not usually one of those priorities, generating many siloed solutions. Add to those the many others from

companies that are compatible only within each company products and are often closed source, incompatibility appears as a large problem for the IoT industry. The straight forward solution to this problem is standardization, and a proposal by a recognized standard agency is a strong candidate for such standard. 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 [1].

The incompatibility is one of the reasons commercial IoT solutions do not always make use of available update mechanisms, with a few exceptions that do not even support OTA updates. The other one being the financial cost, it has an initial cost of implementation, but most importantly, maintaining multiple firmware versions for multiple hardware platforms and its revisions is a constant source of spending that could be used for innovation. This is a similar problem faced by the Android smartphone industry.

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, adding to the body of work to make the RFC a standard. By using the RFC and making the code open source, we aim to ease the adoption of OTA procedures by the industry in a manner that can be easily replicated.

Experiments were designed to assess the impact of the OTA procedure. 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 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. Both had WiFi as the wireless transmission method. 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.

Thus, the main contributions of this work can be summarized by the following:

- design and implementation of an OTA software update solution based on the IETF specification;
- open-source library containing the implemented code;
- evaluation of such solution in both a LAN scenario with constrained devices and a real world WAN scenario with non-constrained devices.

## 1.1 Organization

This thesis has the following structure: Chapter 2 goes over IoT architecture and how to visualize it with a layered perspective. Chapter 3 discusses previous works about OTA software and firmware update, and how they compare to each other. Chapter 4 presents the proposed OTA solution and an overview of its implementation, as well as a qualitative evaluation of the solution. Chapter 5 explains the experiments which were performed to evaluate the proposed solution and show the results obtained, analyzing them. Finally, Chapter 6 presents conclusions, lessons learned, limitations, and future work.

## 1.2 Publications

The following paper was written and published as a result of this research:

- Maria Júlia B. de Sousa, Luis Fernando Gonzales, Erick Ferdinando, Juliana F. Borin. Over-The-Air Firmware Update for IoT Devices on the Wild. *Internet of Things*, 19-C:100578, August 2022.

# Chapter 2

# Background

Most of the technologies that constitute what is now called Internet of Things already existed by the time the term was coined, the innovation came from bringing it all together to create new use cases or expand existing ones. For instance, Wireless Sensor Networks (WSN) similarly connected devices with sensors to a network, though not necessarily to the Internet. Cloud computing and advancements in machine learning allowed for the data collected by the devices to be stored and processed in a timely manner, extracting useful information from it. Besides that, the possibility to use the cloud to combine data of multiple devices in real time opened new possibilities that were not feasible with WSN. In summary, it was the coupling of cheaper device manufacturing and widely accessible cloud computing that allowed for IoT to happen.

IoT hardware and software development for IoT has unique requirements, and for that reason, new technologies were developed specifically for constrained devices. Those can be better understood by layers, parallel to the TCP/IP stack used for the regular Internet, which serves the same purpose of integrating these devices to the regular Internet. Section 2.1 describes what are these requirements and some of the solutions, Section 2.2 presents a form of understanding the IoT architecture with layers, while Section 2.3 focus on how they have been used for OTA updates.

## 2.1 Architecture considerations

To enable the Internet of Things to happen, its architecture was developed around its unique characteristics. At a top-down perspective, it is not that different from the usual network of personal devices (computers, notebooks and smartphones) we interact with daily. The "Internet" is the same, cloud platforms that communicate with devices, but the devices are the source of most of the specificities of IoT, due to their limited nature. These limitations can exist in different parts and puts restrictions on how to develop software for them. The most common are:

i. limited processing capacity;

ii. limited storage;

iii. restricted access to a power supply; and

iv. low bandwidth available.

In addition, IoT applications are different and applied in more diverse domains than those for desktop computers, as was presented in Chapter 1. This led to research into what are the requirements for IoT applications and how to best fulfill them. Many of these requirements are informed by the technology available at the time of development, not just because of the aforementioned limitations, but by what is actually accessible, such as hardware that supports certain protocols, or are regulated in the area it will be deployed, how it operates with different hardware (that might already be deployed in some cases). On the other hand, the purpose of the application also informs the requirements, specially regarding data, such as what type of data to collect and how to treat said data. Integrating an update mechanism for the device's firmware from the moment the architecture is planned also influences the requirements, since it is an important component of applications, and it has its own specific requisites.

One way to elucidate which requirements are more important when designing an application is asking relevant questions about that particular IoT application. As an example, Xu et al. [12] presents some of them, relating to energy consumption, latency and throughput, scalability, topology, and security. More questions can be about standards to be used, technology accessible, and fault tolerance.

The last one is an important aspect to consider, since failure handling can have consequences for the entire architecture. Its main goal is to ensure the application remains operational. For this purpose, it is necessary to "... identify, isolate, correct and record failures." [13]. Firstly, identify that a failure happened, which implies monitoring the entire application and searching for abnormal behavior. After a failure is detected and is communicated to the relevant party, it is necessary to identify where it happened: devices, network or cloud platform. In the case of devices, failures may be caused by a software or hardware problem, be it a natural performance degradation, or caused by the external environment where the device is located. It also may be localized to a single device or a group, each having its own level of severity. For the network, losing the connection between devices and platform, besides the possible data loss, can cause the devices to be unmanageable remotely, so detecting if there is a network configuration problem or a hardware failure (a gateway, for instance) is critical. And for the platform, its complexity can span from a simple database to store the data to a very complex system, maybe connected to other systems, which would need its own research to list the many possible causes of failures.

Once the problem is isolated, it needs to be corrected. This can happen automatically, if it is a known problem with a predictable solution, or it may require human intervention to determine the cause and its solution. Lastly, recording the failure, so it can be used as a reference in case it happens again in the future, and as a way to measure how well the application is performing in the long term.

## 2.2 Layered IoT architecture

One way to understand the IoT architecture is through a layered perspective of its parts. It provides a way for visualizing its elements on a higher level that gives a better understanding of the architecture when planning a solution. There are works in the literature that take this perspective as well. Some of them present a simplified version, with layers as generic as possible [13, 14], while others are more practical, focusing on the protocols and technologies and how they interact [15, 16, 17], or directly based on the hardware used [18]. The disadvantage of a simpler approach is that they sometimes leave out important elements, such as data or device management. While more specific approaches are less generic and dependent on technology.

Focusing on the purpose of each layer yields an approach that balances abstraction with inclusiveness. Once the layers and their purposes are defined, more details can be specified when necessary. There is not an established way to divide the layers, that is why the number of layers and their distribution is different in each work it appears. Despite these differences, the bottom two layers are the only constant, being reserved for the physical devices and their communication. This leaves many functionalities for a single layer in works with a 3 layer model. These functionalities are better divided in Khan et al. [5], with a total of 5 layers, management is part of the application layer, where the data collected and processed is applied for smart health, agriculture, smart city, among other verticals. Security is only mentioned for transmission. The authors in [12] consider only 4 layers, but go one step deeper in how they interact, they also consider security an aspect that is important in all layers. In Marco and Mirisola work [19] the architecture is very similar to Figure 2.1, but added an extra vertical layer, "Identity and Access Management" for device identification and control of what type of resources is accessible to whom.

The idea of separating the IoT architecture into layers can be traced to the Open Systems Interconnection (OSI) model or the TCP/IP stack for traditional networks. This led to some works creating a parallel stack for IoT, which is what Thantharate et al. [20] did, separating protocols and technologies into TCP/IP layers according to their traditional counterpart (CoAP and HTTP, for example). In this case, application and management are in the top layer, parallel to web applications. Similarly, the authors in [21] presented a generic IoT stack and compared this stack to different stacks based on specific communication technologies, such as BLE, Z-Wave, LoRaWAN, Sigfox and more.

The number of layers in other works varies between 3 and 7, depending on the level of abstraction. For this dissertation a model with 5 layers plus 2 adjacent layers, shown in Figure 2.1, will be considered, and each layer will be explained in more details in the following subsections.

The model was based on previous cited works with the goal of being all-encompassing, but at the same not so detailed as to compromise its comprehensive attribute. This way, each layer function has the flexibility to exist in separate hardware or even servers: devices connected to a gateway, that collect all the data and process it before sending to a server where data from different sources is aggregated and the result of the analysis is presented to the end user via a smartphone app. At the same time, some layers may blend together:

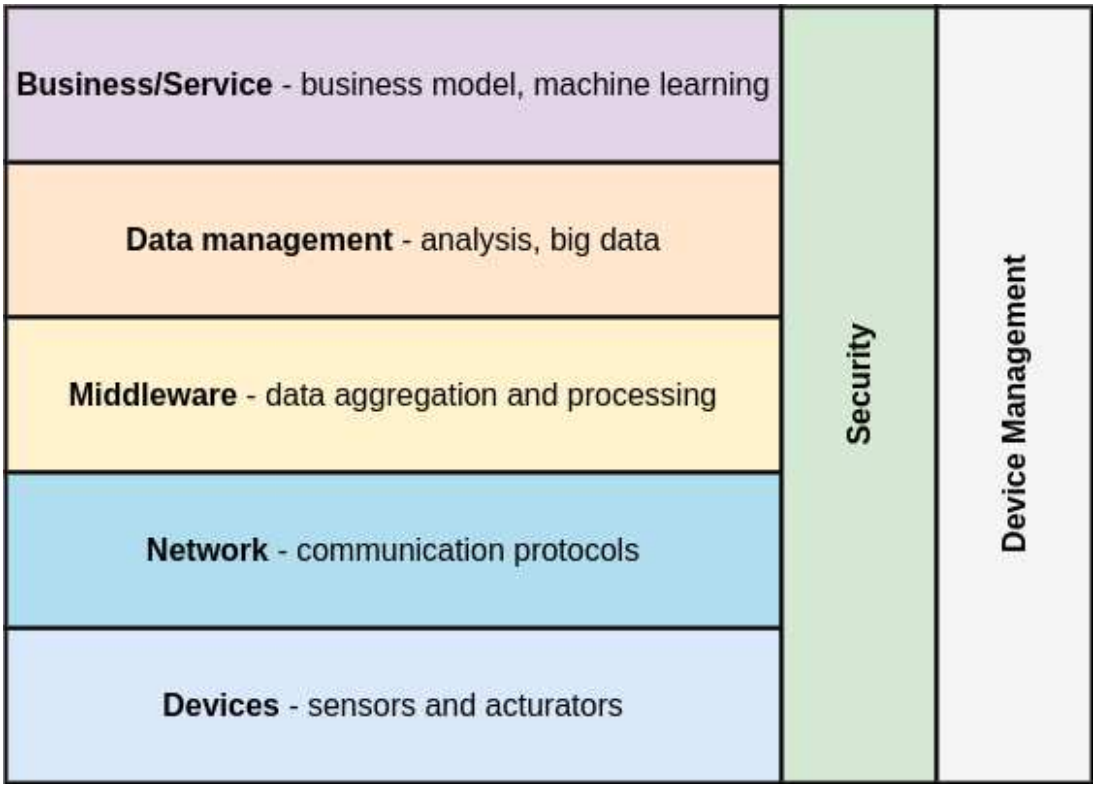



Figure 2.1: IoT architecture layers

the devices send the raw data to a server, where a platform is responsible for all operations from processing to presenting the data.

### 2.2.1 Devices

On the bottom layer of the architecture are the devices, the *Things*, that are used to either get information about the environment as *sensors*, or do something to the environment, as *actuators*.

As explained before, these are resources limited devices since their main function is to collect or receive data, they do not need to be complex. Their simplicity is an advantage in terms of manufacturing, making it cheaper, allowing for multiple devices to be deployed at a time.

A useful tool is to classify these devices according to their capacity. The RFC7228 [22] is one such classification, it defines constrained nodes and networks, and provides a classification based on data size (RAM), from less than 10KiB to 50KiB, and code size (flash), from less than 100KiB to 250KiB, dividing them into 3 classes: C0, C1 and C2. This RFC serves as the basis for other works that try to expand this classification. For Jaouhari [23], 3 more classes were added, to account for more powerful devices, with up to 512KB of RAM and 1MB of flash. They also divide each class in two according to the presence of security features or not. Another work from Gupta [24] is loosely based on the RFC, the features used for classification include clock frequency, power usage, OS and programming language supported, and more, but do not include flash size, and there is a total of 5 classes.

Considering the device's limitations, their limited processing and storage, they require efficient algorithms for data transmission, minimizing it as much as possible, as well as for encryption and key management, which tend to be resource intense algorithms. This also ties in energy efficiency with low bandwidth, seeing that network communication is very power consuming [25] and having large payloads in low bandwidth might extend the communication process. Meanwhile, having reliable algorithms that minimizes data retransmission collaborates to energy efficiency as well, specially in situations where the connection might be unreliable.

Among the companies that made it easier and cheaper to develop new devices are Arduino[1] and Espressif[2]. The first started focused on education, but soon grew into a larger community, with boards and software built to run on them. The Arduino Programming Language allowed embedded software to be more easily developed, in higher level languages such as C++, and the Arduino IDE made it simple to upload the software to the devices using a USB interface. The second, develops wireless chips for embedded devices, including the ESP8266 and ESP32. Another company that contributed to the devices ecosystem is the Raspberry Pi Foundation[3], from constrained devices with the Raspberry Pi Pico, to a single board computer with the Raspberry Pi 4 (its more recent iteration). These are mostly used for education purposes, by tinkerers and for prototyping.

Since the Arduino platform is open source, the community adapted the Arduino libraries for the ESP8266, which is usually programmed in Lua, providing the same utilities to program them using the Arduino Core framework[4].

### 2.2.2 Network

The network layer is where most of the innovation has happened for IoT purposes, rather than repurposing existing technologies. New protocols were created spanning the entire communication stack, from radio transmission to data encoding.

Amid the requirements of an IoT network, there are two focus points that can be considered the main points to take into consideration. One is its range, that can go from a few meters (Personal Area Networks - PAN) to hundreds of meters or even kilometers (Wide Area Networks - WAN). The second is the bandwidth available, that similarly covers a wide range of speeds, from bytes per second to gigabytes per second.

The type of data in which the IoT application is concerned with has a large influence in which protocol to choose, for instance, a protocol with low bandwidth support is not well suited for video transmission, but for small amounts of text data, low bandwidth becomes an advantage in terms of energy consumption.

Another important point to consider is the network topology and how the devices in the network connect to the larger Internet. Most commonly, devices will connect to the Internet through a gateway, that often support more than one radio technology in order to communicate with the devices and the Internet, it can be a router or even a cellphone;

[1] `https://www.arduino.cc`
[2] `https://www.espressif.com`
[3] `https://www.raspberrypi.org`
[4] `https://github.com/esp8266/Arduino`

or they can connect directly to a cellphone tower. It may be possible for the devices to communicate between themselves in a mesh topology, as opposed to a star topology, where the devices communicate only with a gateway.

With regard to the protocols themselves, and considering the OSI network model as a base, at the physical and data link layers, the IEEE 802.15 working group defines wireless personal network (WPAN) standards. It serves as the basis for protocols and standards. One of the standards is IEEE 802.15.4, which defines the bottom two layers, where protocols build on top of. One of such protocols is Zigbee [26], a proprietary protocol used mainly for home automation. 6LoWPAN [27] as well, is an open protocol that works as an adaptation layer between 804.15.4 link layer and TCP/IP using IPv6. Due to being open, there are several implementations of 6LoWPAN that are not interoperable, so an attempt to create a standard is found in the Thread protocol [28]. The WirelessHART [29] protocol is based on 802.15.4 as well, but only the physical layer, it implements its own link layer and uses the Highway Addressable Remote Transducer (HART) for the application layer. These protocols all work in the 2.4GHz range, except for 6LoWPAN that works in both 2.4GHz and sub-1GHz.

Another standard is IEEE 802.15.1, which was a Bluetooth specification, but is now kept by the Bluetooth Special Interest Group (SIG). Bluetooth version 4.0 introduced Bluetooth Low Energy (BLE) [30], that is less power consuming, more suitable for constrained devices. Bluetooth classic and BLE are not compatible, tough many modern devices, such as smartphones, are capable of connecting to both. The most recent version is 5.3.

Another IEEE working group of interest is the IEEE 802.11, that defines wireless local area network (WLAN). It includes the definitions for several WiFi standards. The most used ones are in the 2.4GHz or 5GHz frequencies, 802.11n and 802.11ac respectively, can be found in most routers nowadays. But for IoT, sub-1GHz standards are of interest too, due to long range, low power and low data rates. For example, 802.11ah works around the 900MHz frequency, features lower power consumption than WiFi, and up to several megabits of throughput.

Most of the standards and protocols cited above work on relatively small area networks, but IoT applications may also cover wireless wide area networks (WWAN). They include cellular technologies defined by 3rd Generation Partnership Program (3GPP), those include GSM, LTE, and specially 5G, with technologies such as LTE-M and Narrow Band IoT (NB-IoT), that are focused on machine to machine communication. Among the features of both technologies are long battery life and support for one million devices per square kilometer [31].

The previously mentioned technologies are considered Low Power Wide Area Network (LPWAN). In the same category is LoRaWAN, its main characteristic is the long coverage range of hundreds of square kilometers [32], it has low bandwidth and is a low power network, it operates in the sub-1GHz frequency. Sigfox is another protocol in the sub-1GHz category, though not based on the IEEE standard, it is a network operator that requires a subscription to be used. More details and a comparison of each protocol is given by Tournier et al. [21].

In the top layers, from transport to application, protocols were created or adapted to

focus on IoT. The Extensible Messaging and Presence Protocol (XMPP) is an example of the latter, it was originally called Jabber, focusing on instant messaging. It is based on Extensible Markup Language (XML), which makes it on the heavier side, resource wise, of IoT protocols [13, 33]. Another protocol that originated outside of IoT is the Advanced Message Queuing Protocol (AMQP), mostly used in middleware applications. It uses TCP for transport and has two QoS levels for reliability [17]. Though HTTP is not commonly found in IoT applications due to its heavy resource usage and overhead, it is found in the literature [34]. It has the advantage of being widely used and supported.

It is possible to find these protocols being used in the literature and industry, but they are not the most popular. This is most likely reserved for Constrained Application Protocol (CoAP) and MQTT[5], two very lightweight protocols found in many works.

Corak *et al.* [33] compares CoAP, MQTT and XMPP in terms of packet creation and transmission time, in both instances MQTT performed better and XMPP was orders of magnitude slower, with CoAP performing slightly worse than MQTT. Meanwhile, Naik, N. [17] makes a qualitative comparison between MQTT, CoAP, HTTP and AMQP. Among them, HTTP had the higher bandwidth usage and latency and CoAP had the lowest, AMQP had the highest security and extra features, while MQTT had the lowest.

CoAP can be considered an HTTP equivalent for IoT. It is designed by the Constrained RESTful Environments (CoRE) IETF working group [20, 21], and as the name of the group suggests, it is based on a REST architecture with methods to GET, POST, PUT or DELETE resources identified by and Uniform Resource Identifier (URI). It works over UDP, making it generally coupled with DTLS for security.

Since MQTT is relevant for this work, a more detailed explanation is given below:

### Brief overview of MQTT

MQTT was created in 1999 by IBM, but is now part of OASIS open standards. Its most recent version is 5, but the previous version, 3.1.1 is still widely used. It works in a publish/subscribe model over TCP connections, which allows it to use TLS/SSL for security [20, 21].

Its basic elements are a client and a broker. The broker receives messages from clients that publish them to a topic, after that, any client that is subscribed to this topic will receive the message. In this manner, clients only talk to the broker and not between themselves. There are various types of messages that can be sent, among them are CONNECT, PUBLISH, SUBSCRIBE, and more. The protocol also offers three levels of Quality of Service which define the guarantee of delivery for a message: 0 - at most once, 1 - at least once, and 2 - exactly once.

Some of its advantages are the very lightweight and asynchronous communication, and the large availability of implementations for client and broker. Clients can be found in many programming languages, the Eclipse Foundation provides the Paho library in C/C++, Python, Java, JavaScript and more. A comprehensive list can be found in MQTT website [35]. While Mosquitto and RabbitMQ are well known open-source broker

[5]It was originally Message Queuing Telemetry Transport, but the last two versions of the specification [35] refer to it exclusively as MQTT

implementations. MQTT is also supported by many commercial platforms such as Google, Amazon, Canonical and Microsoft.

### 2.2.3 Middleware

The middleware layer is responsible for aggregating data from the network layer. This data comes from different sources, as a variety of hardware and protocols generates data in diverse formats. The main function of this layer is to start processing the data, filtering anything that got corrupted during transmission, doing any conversion necessary for the upper layers, or aggregating before forwarding or storing it. Mirisola, M. [19] divides this tasks in three phases: Extraction, Transformation and Loading.

Since data is transformed in this layer, it is important to keep it up to date with any changes in other parts of the application that can affect the data to ensure it remains constant and reliable.

In terms of hardware, the operation of the middleware layer can occur in different elements of an application, it can be an edge computing device, for instance, in solutions involving fog computing [36]. It can also be gateways, aggregating data from devices in its LAN before sending to a platform. Or the data might arrive in the cloud or platform unaltered and all processing is done there, facilitating the communication with the upper layers.

### 2.2.4 Data Management

In the data management layer, more complex data handling is done. This is the layer responsible for storing the data collected from the sensors and processed by the middleware layer. This includes keeping databases and its backups.

Beyond that, it is responsible for preparing the data to be served in a more friendly manner for clients, using data analysis techniques and big data algorithms, when applicable, though not machine learning.

In his work, Mirisola, M. [19] specifies three approaches for dealing with big data: cloud computing, real time computing and fog computing.

### 2.2.5 Business/Service

The top layer is what clients of IoT applications interact with. The client can be a company managing a factory, government managing a city, a person managing their house, or many more types. Usually there is an interface where the client can see the information from the data management layer, and, in some cases, have direct control over devices. This interface might be accessible through a browser, mobile application or both, having it accessible via the internet being its most important characteristic.

In such interface, the information can be presented in a dashboard of charts and graphs. The information can be the result of the processing in the data management layer, or needs more complex analysis with machine learning [5, 19]. In some applications, there might be no distinction between the data processing in this layer and the one in data management.

In turn, the information can be used for decision-making. It can be about changes in the application by whatever mechanisms are exposed in the business layer. The decision on the changes can be made by a person, such as the client or system manager, or automated using machine learning. These changes can go from policy updates about data collection, data rates, firmware updates, among others, to specifically towards single devices.

This feedback loop is strongly correlated to scalability, providing a sense of how much room for growth an IoT application might have, whether it is working as intended or it is necessary to scale up or down.

### 2.2.6 Security

Security is an extensive part of IoT, it is the entire focus of many works, such is its importance. The lack of security mechanisms can have great impact on the solution and in peoples lives, and there are examples that demonstrate so: the infection of CCTV cameras and home appliances, turning them into botnets, to launch Distributed Denial of Service (DDoS) attacks [37] or mine bitcoins [38]; two researchers were able to hack and control a Jeep, causing an immense recall of *Chrysler/Jeep* vehicles; a vulnerability in the ZigBee protocol allowed for a new malicious firmware to be updated to Philips Hue smart lamps [39].

These demonstrate concrete consequences of bad security practices. Besides avoiding them, having good security practices also have the benefit of increasing the solution longevity. However, implementing such mechanisms demand time, resources and knowledge of the best practices, and in an environment where shipping a product as fast as possible is a priority, security often goes ignored. It is important to note that despite the beforehand "costs" of implementation, dealing with the consequences of a hack (such as the aforementioned examples) may implicate in much higher costs, and correcting the problem by adding security mechanisms might involve a considerable amount of re-engineering. One effective way to diminish security "costs" is including security features from the planning of a new product or solution, i.e. built-in security, as opposed to add-on security, normally found in traditional IT [14].

The reason security spans all the layers in Figure 2.1 is that for a really secure solution, all layers should be taken into account. In many cases, each layer requires its own mechanisms, that when used in conjunction with the other layers, make the entire stack more secure.

This is specially true for the devices layers, where the heterogeneity of platforms combined with their constrained nature makes it difficult to reuse the knowledge developed for the World Wide Web. Many algorithms had to be adapted, or new ones created to work in such environment, making it difficult to standardize them. Devices characteristics leave them vulnerable to physical attacks, where an attacker may obtain confidential information or alter the device to carry further attacks. There are some aspects one can pay attention in order to mitigate these risks, such as not leaving open debug interfaces or storing sensitive information in protected memory whenever possible.

Whenever data moves between layers, the transport can happen through insecure channels. In the upper layers, most times regular end-to-end security mechanisms are available,

such as encryption, a public key infrastructure, secure authentication and more, which means previous knowledge can be reused to secure its transport. However, the communication between the devices and the Internet (through a gateway or not) is much harder to be secured, for the same reasons as it is hard to secure the devices themselves. The protocols described in Section 2.2.2 each have their own security features and weaknesses. In their survey, Tournier *et al.* [21] argue that besides the protocols being incompatible, the security studies based on them are also incompatible, but even then it is possible to find common abstract characteristics by virtue of their common IoT architecture.

Since the data management and business layers deal with all the data of the system, it is very important to keep it safely stored, with proper access control and authentication, to avoid data leakage. This has serious implications for privacy, for instance, smart health applications contain patient confidential data that can not be shared with unauthorized entities.

For the purpose of this work, IoT security goals can be summarized in three categories: confidentiality, integrity and availability [14, 21]. Confidentiality is having information available only for authorized parties, for example, an attack that breaks this is a man-in-the-middle attack between a device and its gateway, because it could expose confidential data. Integrity is certifying that the stored data is not altered, or data in transit is the same from source to destination, for example, a spoofing attack, where a device is impersonated and can inject bad data into the system. Lastly, availability is making the information readily available for any authorized party, an example of an attack are DoS attacks, targeting devices can remove them from the network, losing the data they collect, or targeting the service layer causing inaccessibility for clients.

### 2.2.7 Device Management

To ensure devices are working and continue to work, device management is an important part that comes with its own challenges. Managing numerous and heterogeneous devices individually becomes an impractical task. For this reason, a device management scheme is vital for IoT applications. Managing devices includes:

i. monitor device status;
ii. perform troubleshooting procedures on devices;
iii. firmware update control;
iv. change working parameters (for example: the frequency which data is sent).

It is mostly done remotely, since physical access to every device deployed is not always simple. Manually managing a deployed device should be avoided, except when other options have been exhausted, for instance when a device breaks and needs to be replaced.

Monitoring the performance of the entire system is essential for its long term operation, from devices, communication to application. Monitoring device's resources includes metrics such as memory, CPU, and energy consumption. Monitoring the network involves measuring online and sleeping times, data rate, and latency. At higher layers, it is important to monitor data processing metrics, such as speed, correctness and more.

In terms of reliability of the application, there are parameters that indicate the quality of the data collected. For example, how often it needs to be collected to get the useful insight from it while balancing device power consumption and network usage. This can be adjusted during the application lifetime, using previous data as feedback. Ensuring the data arrives correctly and values remain consistent between collection and processing will also guarantee the information extracted is correct and useful.

In order to maintain devices working correctly, the device manager should keep information about each individual device, such as device state of operation (it can be sending data, sleeping to save energy, updating, faulty operation) and information for troubleshooting whenever needed. Since management usually involves interacting with different hardware platforms that may support different data rates, protocols or data formats, keeping information about what software is running on each device, alongside its version and dependencies, if any, is essential for the firmware updating part of the system as it aids the control of the update process.

Besides updating the device's firmware, it might be possible to change operating parameters of devices (for example, the frequency data is sent). Its delivery is similar to an update, but it is handled differently by the devices.

A description on how to manage the network is given by [13], the authors consider using Simple Network Management Protocol (SNMP) for IoT, and Internet of Things Platforms Infrastructure for Configurations (IoT-PIC), which is based on SNMP. They also compare two device management platforms. Paper [40] is very focused on management, it extends the Open Mobile Alliance (OMA) LwM2M into oneM2M architecture and embed into the architecture a way to manage legacy devices that might not have the same capabilities.

An important characteristic of a device manager is the ability to address each device independently. Some authors [5, 12] highlight the importance of Radio Frequency Identification (RFID) for this purpose, but URI and IPv6 are options as well. Besides the implications in the overall system operation, it also has security implications in the form of access management, where a device has its own scope of accessible information, and where each system operator may only be able to access certain groups of devices.

## 2.3 Updates Over the Air

Given the importance of having devices running up-to-date software, as described in Chapter 1, any IoT architecture should have an over the air updating mechanism, furthermore, it should be integrated with the architecture from the beginning.

Updates over the air is the whole process from developing a new firmware version, introducing it to the architecture via the mechanisms available, until it reaches the bottom layer of the architecture, the devices, with it installed and running in all devices it is pertinent to. Despite all it envelops, the focus of this work is in the later part. Considering that, there are two key points to the process: dissemination and activation [41, 42].

Dissemination is how the new firmware is transported across the network to the end devices. As expected, the most relevant layer is the network. One aspect is its topology, which has a big influence in how the dissemination happens. In a mesh network, sending

large amounts of data over multiple hops is a big challenge, for this reason mesh networks are not a common topology, tough there is research into a reliable broadcast algorithm for file transmission [43]. Another possibility is a star topology, where devices communicate with a gateway, in this case, the gateway can be leveraged to coordinate the update in its own local network. Finally, devices might connect directly to a base station, fetching the update directly from the update server.

Once the update is ready to go, an important architectural decision that influences how it is disseminated is the mode in which it is initiated. The update server might initiate it by notifying the target devices, or the devices might query the server for a new update, or it can be a hybrid of the two modes [1, 23, 24].

Meanwhile, activation regards the devices and how they handle receiving an update. Depending on the device's capabilities, an update might happen in an "update mode" where its other functionalities stop until the update is complete, or it may be capable of handling the update in parallel with its other functions. Device behavior might also change depending on the type of update, that is, which part of the firmware is being updated. It can simply replace the entire firmware, or it can be partial [25]. A partial update can be a differential patch, which is usually complicated for constrained devices to handle, or it can be modular, where a specific part of the firmware is replaced, such as the network stack, an application. Replacing the bootloader is also a possibility, but it is less common, since a failure may cause the device to be unable to boot. In some cases, no code is replaced, only configurations, for example, frequency which data is collected, which server to send the data, etc. It also involves fault tolerance and security measures, to ensure the update will not disrupt any device that is working normally.

For an update to happen, first a new version of the firmware needs to be developed and made available to devices. This process can include multiple parties, from the device manufacturer, distributor, original equipment manufacturers (OEM), ISPs [23], to end users, all can have a stake in the creation and distribution of a new firmware. This factor sometimes is the main reason updating IoT devices is not a more common practice, since companies may go out of business or simply decide to stop supporting an older device in favor of a newer one. Most developers do not account for ownership changes when developing a new product, even though there are solutions for this problem [24].

The update process can also follow one or many policies that are decided by an authorized operator. For instance, device selection for the update, by a specific type of hardware, or geographic location, by client, or many other criteria. Policies might refer to how the update is applied, for example the activation of a new firmware version might be deferred to a specific moment in time, as to not interrupt a service during working hours. They may also refer to the severity of the update, from trivial to urgent [20].

In order to coordinate, enforce policies and make the update happen, an essential part of the architecture is the device management. It is the element responsible for keeping track of devices information relevant to updating, such as firmware version, in addition to log and report any feedback regarding successes or failures.

Beyond using the device management infrastructure, security mechanisms are also very important, as the consequences of lack of security can be devices compromised at the firmware level. Moreover, the most used layers from Figure 3.1 are the bottom two,

network for transmitting the firmware, and devices, where the firmware is installed. In some cases, clients may be able to have some control over the process or visualize the feedback through the service layer.

# Chapter 3

# Related work

The concepts explained in the previous chapter are the basis for understanding IoT and OTA updates. This chapter focus on works that implement and evaluate them, with special focus on works about updates over the air.

As explained in Chapter 2, previously, the research on devices was focused on WSNs. From this body of work, the surveys conducted by Brown and Sreenan [44], later updated in [7] are a good summary of the state of the art up to that moment in time. They present a good overview of how those networks worked, with a focus on software update. They identified four main characteristics to divide the frameworks and protocols involved in the OTA update process:

i. dissemination of the new code through 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 in more recent research, as they address issues like power efficiency, performance and security. According to the authors, dissemination is the most researched area and fault detection and recovery is the area that needs most improvement.

From the surveys, it is notable that many solutions were built to be used with TinyOS [45], an operating system made for embedded devices that was discontinued in 2013. The default propagation protocol for TinyOS is called Deluge [46] (based on Trickle [47]), both based on broadcasting one or several packages containing the software upgrade to the network. Another embedded OS that was used in several studies was Contiki [48] that, after being discontinued, was forked into Contiki-NG [49]. MOAP, a Multihop Over-the-Air Programming [50] mechanism focused on being memory and energy efficient, was frequently used for update dissemination as well. In terms of newer embedded operating systems, RIOT OS [51] and ARM mbed [52] are often cited in surveys [21, 23, 41, 53, 54] and used by solutions [11, 55].

The authors in [6] lists the main concerns of updating a WSN. According to them, the complexity of the algorithms used to perform an update must be low enough to fit the device's memory and small processing capacity, they must also be energy efficient,

this implies efficiency in communication and in writing data to EEPROM, since these are the most energy consuming operations. It should also guarantee the full delivery and correctness of the firmware in an unreliable network, and must scale for number of nodes, as well as for node density. These can be compared to the survey conducted by [7]. In the survey, they catalog the updates features of a WSN in update dissemination and activation, which encompasses the first concern about complexity, fault detection and recovery, which includes guaranteeing the firmware delivery, and management.

After devices got connected to the Internet, it became possible to manage an entire network remotely, usually from a platform built for it. An IoT platform, besides aggregating the data collected by the devices, can act as a device management tool, as described in Section 2.1. One of the management tasks is firmware update control. Being a non-trivial task, management itself became an object of study. A solution based on the Simple Network Management Protocol (SNMP) is given by Silva *et al.* [13]. A standard for device management was developed by OMA, called Lightweight Machine to Machine (LwM2M) [56], generating works studying its applicability [8, 55, 57]. Hernández-Ramos *et al.* [8] focus on device version control and managing dependencies to avoid mismatching versions.

IoT is an environment very restricted and, at the same time, heterogeneous for developing solutions for. For these reasons, updating these devices requires a different process than updating software in a computer or a smartphone. Most of the problems faced by Internet of Things devices were already being explored by research in Wireless Sensor Networks, those remain mostly the same, with the additional problems and easiness brought in by connecting everything to the Internet.

Again, Hernández-Ramos *et al.* [8] presents an extensive list of challenges to be overcome by updating IoT devices. Some of them are version and dependencies control by the device, to ensure the device will continue working after updating; integrity and confidentiality of the firmware image, alongside efficient cryptography and security. They also explore the idea of having devices digital twins in order to test new software and assessing its impact, which is not very well explored in the literature. In [57], the challenges presented stem from the constrained nature of devices and their connectivity and reachability. Both works identify standardization as an important goal, and proposes solutions, the former based on the LwM2M standard, and the latter using blockchain. Both also present a solution based on the SUIT standard proposed by an IETF [1] working group, that later became RFC 9019.

In order to overcome these challenges, there are many works that propose an OTA solution. A model based on CoAP and MQTT is given by [20], which based on experiments, recommends one or the other, or a combination of both, depending on the type of update (i.e. software or security) and its severity. The authors in [10] provide an extension of IoT architectures to enable application and network level upgrades, as well as implementing and evaluating it using Contiki. The work from Carlson [15] 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.

There are solutions for more specific domains as well, one of them being connected cars, which is already a reality, with "infotainment" systems and an array of connect electronic control units (ECU) that control the car functions, such as steering, acceleration, breaks and more. Khurram *et al.* [58] discuss threats to these components and suggests that OTA system should be integrated to a vulnerabilities' detector, so any problem can be detected and corrected as fast as possible. In turn, Nilsson *et al.* [59] proposes an architecture of portals that manage vehicles that join and leave the network, and a lightweight protocol for secure OTA [59] and for firmware verification after download [9].

A useful tool for IoT development are testbeds. Testbeds are environments with resources available for testing new developments. Resources can include hardware, test software or networks, in the case of IoT. The SmartSantander [18] is a Smart City testbed designed to be used by developers, who can run experiments using the IoT infrastructure. It contains temperature, humidity, $CO_2$ and parking sensors, that can be reprogrammed to suit the experiments. While TinySDR [60] is a hardware platform for endpoint devices development, it supports several radio protocols and has a dedicated LoRAWAN chip for OTA programming.

## Fault tolerance

In the previous chapter, it was mentioned that failure handling is divided in four steps [13]: identify, isolate, correct and record. It is no different when focused on device updating. Having a feedback mechanism is crucial to identifying any failure, and the same mechanism can be used to isolate the failure with regard to which moment the failure happened, be it during firmware transmission, storing or flashing. There are different techniques that ensure the firmware is received correctly, hash functions can be used to verify the integrity after download and after flashing it to permanent storage [9]. Splitting the firmware in segments and applying an error correction technique [42] can be a way to avoid retransmission of the entire firmware, since this adds redundancy without the overhead of sending acknowledgment for each segment.

The correction of the failure depends on which part of the process it happened and its cause. The most frequent causes are usually power loss for the device or loss of connection during the update. A software problem, such as segmentation faults or infinite loops are ideally detected before updating the device, by robust firmware testing, but may happen after deployment regardless. In order to fix the failure, it might be necessary to restart the entire process, or just the firmware download. If the new firmware is already installed, a rollback might be needed. Many works argue the importance of keeping the last functioning version of the firmware in a separate storage partition for this purpose [7, 11, 42, 55]. The rollback process must be supported by the bootloader.

As for recording of the failure, it can be kept in the device in the form of internal logs, or in whatever platform it connects to. If a feedback mechanism already exists, recording is done as part of said mechanism.

## Security

In Chapter 2, the importance of security for IoT in general and its consequences was discussed, plus a few ways applications can be secured. In the context of updates over-the-air, many of the discussed mechanisms are still relevant, but there are specific OTA mechanisms that will be discussed in this section, and the solutions found in the literature.

Since many times the whole firmware is replaced with an update, entire devices can become compromised, which in turn may disrupt the entire system. A compromised system can cause a breach in one or more of the security goals: of confidentiality, integrity or availability of the system, additionally, it puts consumer privacy at risk.

In terms of the three security goals of confidentiality, integrity and availability, the first can be broken when an attacker has access to the firmware, either by having physical access to a device or by intercepting unencrypted communication; or to information about the firmware in a device that would enable an attacker to forge an update, such as cryptographic keys. The second is broken precisely by forging an update and installing a malicious firmware that gives control of one or more devices to an attacker. Lastly, availability can be broken by disrupting a device normal operation via the update infrastructure.

The development of the firmware and its transport up to the server from where the devices will download it, can make use of any state-of-the-art security mechanism, and its weaknesses are beyond the scope of this discussion, which will focus on the path from update server to device.

Assuming the firmware in the update server is correct and properly authenticated, the path forward, is where the most evident attack surfaces exist, the over the air communication and the devices themselves. Regarding the protocols focused on IoT, there are works presenting security features and what types of attacks they are vulnerable to [14, 21]. Besides having a similar discussion, Mohanta *et al.* [16] also discuss how machine learning and blockchain can be used to improve security in IoT systems. Since each solution has different levels of security, standardization could help IoT overall security. It is this that is defended by Keoh *et al.* [61] in their work. They mainly discuss the use of CoAP together with DTLS, copying the HTTPS/TLS dynamic used by websites. This is the strategy used or tested in a number of updating solutions [15, 33, 41].

In a different approach, the authors in [34] used HTTPS over WiFi for the extra security provided by TLS. They found that it adds several bytes to the packet headers, which is acceptable for WiFi connections, but might not be for low bandwidth networks. That is one reason UDP and DTLS may be preferable, at the cost of delivery guarantee.

To ensure confidentiality and integrity of the firmware, two works [59, 62] suggest a method where the firmware is split into blocks used for hash chaining and each hash is encrypted before being sent to a device, where it is decrypted and reassembled. Schmidt *et al.* [11] propose different types of firmware packages with combinations of security mechanisms, balancing how secure or fast the update should be. A technique used in many solutions to guarantee integrity is signing of the package [11, 15, 55, 63].

Preventing a device to be infected with malicious code that would give its control to the attacker is one way to maintain availability, but an attacker does not need to control

the device to make it unavailable, a DoS attack with repeated false updates can achieve the same end. The Host Identity Protocol (HIP) [61] provides protection against it.

One difficulty of properly securing devices is the toll cryptographic takes on them. Zandberg *et al.* [55] measured the impact of OTA and the impact of cryptography in the update process. In their testing scenario, crypto-related tasks took 68% of the total time spent updating, and represented a significant percentage of flash usage, but less on RAM, where network code was much more significant. Another work from Schmidt *et al.* [64] tested the impact of tasks such as decryption and signature generation and verification in terms of time, memory usage and power consumption. They tested their own OTA solution, as opposed to the previous work, where different strategies were tested. In their findings, decryption was the most power consuming task.

More generally, in the surveyed OTA solutions, security was usually hit or miss. It either was the main focus of the solution, or was only mentioned as important, but not implemented. Where it was present, the main mechanisms to ensure security were key exchange for authentication, some form of encryption for the transmission, which depends on the device being capable of decrypting the messages, and digital signatures to guarantee firmware integrity.

## 3.1 Comparison of OTA solutions

The context in which an IoT solution will be applied influences which requirements are to be fulfilled and which policies and strategies are better suited to fulfill them. Taking into account this and the challenges discussed before, there are several existing solutions, each making a different set of requirements priority. Table 3.1 makes a qualitative analysis of some of the main solutions for updates over the air focused on IoT. For each solution, the following aspects are described:

**Communication** Protocols and radio technologies used for connecting devices

**Security** Mechanisms employed, from firmware generation to booting the new firmware. The specific method is not detailed, only its usage. Authentication may include firmware author authentication in order to perform the update. Signing refers to digitally sign any payload that is transferred to the devices, which can be used for authentication and firmware integrity verification after download. And encryption refers to payload encrypting.

**Node Addressing** If it is possible to address devices individually via *unicast*, and if it supports *broadcasting*

**Fault Tolerance** If any fault tolerance mechanism is discussed in the work

**Power Consumption** If there is any discussion of power consumption, be it in terms of saving energy or measuring usage during the update process

**Management** How devices are managed, if there is any platform or framework

**Multi Hardware** For what type of devices the solution should work. If it is not specified in text that the solution is generic, it is assumed that it can work in devices with a characteristic relevant to the solution, such as compatible with an OS or used in a specific setting

**Device Class** Which class of devices the solution was tested on, the classification is the same from [23] with the addition of a "SoC" (System on a Chip) class for the devices that are much more capable than what is included in the reference classification

**Experiment Scenario** Scenario where the solution was tested, if any, specific board(s) and OS used

Fields with a − means that the authors do not discuss those aspects. A ✓ means that it is either supported or it is discussed, depending on the column. Finally, a ✗ means that it is not supported. The last row of the table refers to the OTA solution proposed in this dissertation.

| Paper | Communication | Security | Node Addressing | | Fault Tolerance | Power Consumption | Management | Multi Hardware | Device Class | Experiment Scenario |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Broadcast | Unicast | | | | | | |
| [9] & [65] | – | Signing and encryption | ✗ | ✓ | ✓ | – | Portal entity manages nearby vehicles | Automotive Eletronic Control Units (ECU) | C0-2 | – |
| [11] | – | Signing and encryption | – | – | ✓ | – | – | – | C2 | Atmel SAMR21 running RIOT-OS |
| [15] | CoAP over 802.15.4 | Signing and end-to-end encryption | ✓ | ✓ | – | ✓ | – | – | C2 | Firefly (ARM Cortex M3) running Contiki-NG |
| [19] | CoAP over 6LoWPAN | Encryption | – | – | ✓ | ✓ | OMA LwM2M | STMicroeletronics boards | C4-5 | STM32F401RE, STM32L476RG running Contiki |
| [20] | CoAP and MQTT | Encryption | ✓ | ✓ | – | – | – | Devices capable of running CoAP/MQTT | – | Simulation |
| [24] | Bluetooth | Signing and encryption | ✗ | ✓ | ✓ | – | Generic Software Update Provider | Generic | C0-5 | ATMega2560 |
| [25] | – | – | ✓ | ✓ | – | ✓ | – | Contiki compatible boards | C0-2 | TelosB, Zolertia z1, RM090 |
| [34] | HTTPS over WiFi | Signing and encryption | – | ✓ | – | – | REST API | – | SoC | Own Linux distribution for Raspberry Pi (ARMv6 and ARMv7) |

| Paper | Communication | Security | Node Addressing | | Fault Tolerance | Power Consumption | Management | Multi Hardware | Device Class | Experiment Scenario |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Broadcast | Unicast | | | | | | |
| [41] | – | Authentication and encryption | – | ✓ | ✓ | ✓ | – | Generic | C2-3 | Zolertia Re-Mote (ARM Cortex M3) |
| [42] | 802.15.4 | – | ✓ | ✓ | ✓ | – | Network manager | 802.4 compliant hardware | C1 | 100 ATmega128RFA1 running Contiki |
| [50] | – | – | ✓ | ✓ | ✓ | ✓ | – | – | C1 | 15 Mica-2 motes running TinyOS + emulation |
| [55] | CoAP over 6LoWPAN | Signing | ✗ | ✓ | – | ✓ | LwM2M | – | C2-4 | Atmel SAMR21, STM32103REY, Nordic nfr52840, all running RIOT-OS |
| [62] | LAN technologies - WiFi, NFC, Bluetooth | Authentication, encryption and signing | – | ✓ | – | – | Gateway for LAN management | Devices for home automation | – | – |
| [63] | Ethernet | Signing and encryption | – | ✓ | ✓ | – | Hawkbit | SWUpdate compatible boards | SoC | SiFive Freedom U540 |
| [64] | – | Secure bootloader and signing | – | – | ✓ | ✓ | – | – | C2 | Atmel SAMR21 running RIOT-OS |

| Paper | Communication | Security | Node Addressing | | Fault Tolerance | Power Consumption | Management | Multi Hardware | Device Class | Experiment Scenario |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Broadcast | Unicast | | | | | | |
| [66] | – | Authentication, encryption, attestation | – | – | ✓ | ✓ | TUF | Generic | C0-3, SoC | I.MX6-SabreLite and ARM Cortex-M23 running HYDRA |
| [67] | Lightweight Mesh - 802.15.4 for Link Layer - over P2PMesh | – | ✓ | ✓ | ✓ | ✓ | RESTful API | – | C2 | Gateway: Raspberry Pi. Mesh nodes: Atmel SAMR21. |
| This work | HTTP over WiFi | Authentication | – | ✓ | ✓ | – | Konker IoT platform | | C4-5, SoC | 20 ESP8266 and 75 Raspberry Pi 3 |

Table 3.1: Comparing OTA solutions and our proposal

From the communication column, many solutions are using technologies focused on IoT for communication, CoAP seems to be the choice to replace HTTP in combination with 802.15.4 based radio. With this choice, most solutions are tested in a LAN or PAN setting, including solutions using different communication methods. The only works that included a testbed over a wide area were [42] and ours.

With regard to security, the main strategy is signing the payload, while authentication and encryption follow closely. Many works were focused on one or more of these aspects of secure updates ([9, 11, 15, 20, 34, 55, 62, 66]). From those, some measured the energy [15, 25, 34, 64] used during the process, while others focused on time measurements [20, 55]. It is noteworthy that implementing security mechanisms tends to demand more effort and specialized knowledge, which could explain the trend of either having security as the basis for a work, or not implementing anything or almost no mechanism. For our work, it is more deeply discussed in Section 4.3.

Addressing nodes individually is always possible on the works where it is specified, whereas broadcasting is less considered, but when it is, it is supported, except for [9, 65, 24, 55]. For fault tolerance, the most used mechanism was partitioning memory in two banks to allow a rollback in case of a faulty update, meanwhile the feedback mechanism is not always specified. Similarly to security, this aspect of our solution is more deeply discussed in Section 4.3. Power consumption was less considered than the previous column, but tended to be a main concern when present, specially for very constrained devices. The main focus points were with energy usage during communication and encryption processes.

Almost half of the considered works did not specify how to manage the IoT solution, and from the ones that did, about half used a simple solution for the update process specifically, either implementing their own update server [34, 67], or using an existing framework. The Update Framework (TUF) [68] appears somewhat often in the literature. A newer framework is Hawkbit [69], an open source backend developed by the Eclipse Foundation.

While the other half used a more complete solution that can handle operations beyond updating, as described in Chapter 2. OMA LwM2M [56] is the only one used more than once, and it is often considered in the literature. The Konker Platform is also a full-fledged platform, which is also open source. The platform is capable of individually addressing devices.

Almost none of the works specify the solution as generic for any type of hardware. As microcontrollers get more powerful, the need for super constrained solutions decreases, as seen by the fact that few solutions were tested in classes 0 and 1 devices. Our solution was implemented and tested in slightly more powerful devices, but the RFC which it is based on, the feedback mechanism, and IoT platform, all are hardware-agnostic, so it could theoretically be implemented in more constrained devices.

Lastly, most of the experiments are proof of concept, tested on a single board (SAMR21 being the most popular one), or in different board models, but only one of each. Almost no experiments happened in a testbed with multiple devices ([42, 50] and ours). Similarly, simulation of the network [20], or emulation of devices [50] was not a common way of testing.

Other authors made a similar type of comparative analysis, for instance an interesting

comparison of protocols and some studies involving them is done by Corak *et al.* [33], that makes a qualitative and quantitative analysis of CoAP, MQTT and XMPP, and evaluates other studies with similar purposes. Brown and Sreenan [7] have a thorough comparison of their surveyed solutions, they divided into dissemination, activation, detection, recovery and management, and each is subdivided into several features that can be present in each solution. For Jaouhari and Bouvet [23], the comparison is focused on security features and hardware support, which is the focus of their work.

A comparison of both open source and private projects and their support for OTA updates is made by Villegas *et al.* [54], the authors found that many solutions did not specify anything related to OTA in their documentation, and suggest separating the logical structure of the updates from the technologies used, this way the structure can become a standard, allowing companies to build on the implementation. One such attempt at standardization was made by IETF and will be explained in more detail in the next section.

## 3.2 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 [1]. That draft was turned into 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. By focusing on the manifest, the proposed architecture can be flexible enough to be applied in many situations, but at the same time, robust in regard to functionality and security.

There have been a few works implementing and testing different aspects of the proposal. Ruckebusch *et al.* [25] shows 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 [24, 55] and the conclusion is that even very constrained devices were capable of handling them. The architecture from the RFC is compared to others architectures of different complexities, in [55], from a baseline to combining the SUIT with LwM2M. Asokan *et al.* [66] uses elements of the RFC, such as stakeholders and objects, to design their own architecture. The work from Carlson [15] 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, their implementation uses Contiki-NG as the embedded operating system and perform tests only on a single board.

RFC 9019 is also cited in more recent survey-style works. Both [8, 23] cite it as an example of standardization of the firmware update process. It is often cited alongside LwM2M as an update approach [57]. Villegas and Astudillo [70] also considers it and extracts recommendations based on the architecture.

The general architecture described by the RFC can be seen in Figure 3.1. 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 managing 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 make 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* or 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 one at a time or bundled together. After going through the steps of validating the firmware, it can be installed in the devices by their bootloader.

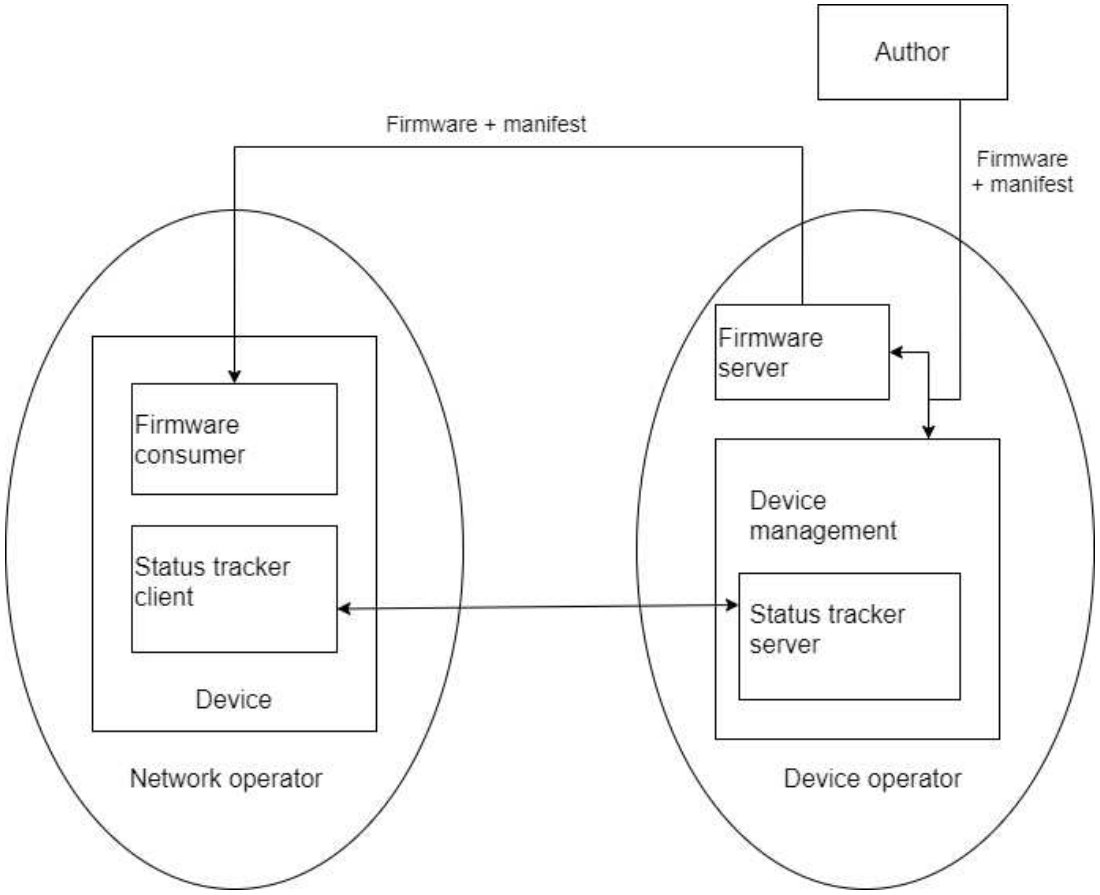



Figure 3.1: RFC 9019 Architecture [1]

This dissertation will expand on this body of work, also building upon the IETF draft architecture, but using regular internet protocols, such as HTTP, the reasoning being that it is easier to integrate it to the industry, that is accustomed to the 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 in a local network with several devices functioning.

### 3.2.1 The Manifest

The purpose of the manifest is to inform a device about a new update, giving it the means to acquire it and verify its authenticity and correctness. It works as metadata for the firmware and is secured via signing.

The RFC uses the manifest defined by another IETF RFC, 9124, [71], where its fields are detailed, and several security considerations are made about each field.

Taking into account the considerations made by [1, 71] and to enable its goals to be achieved, the manifest is composed of a set of fields, some of which are mandatory, and some are optional (though can be marked as recommended). The mandatory fields are listed below:

**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 device 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 (envelope element)

**Dependencies** A list of other manifests required by the new one

**Encryption wrapper** A way for the device to obtain the key that decrypt 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. *Vendor ID* and *Class ID* are both to identify identical hardware from different vendors, or classes of devices from the same vendor. As an extra step against rollback attacks, for devices that keep track of time, the manifest can have an *Expiration Time*. The payload server can be pointed by the *Payload Indicator* as a URI, and extra *Processing Steps* can be sent for firmware installation, such as indicating compression or encryption algorithms to be used, or scripts to run.

# Chapter 4

# Method for IETF-based Over the Air Updates in IoT Devices

This chapter features the designed IETF-based OTA firmware update and how it was implemented, as well as a qualitative evaluation.

## 4.1 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 included the design and implementation of the firmware for the IoT devices and the definition of the messages exchanged between the involved entities.

### 4.1.1 OTA process model

From the architecture presented in Figure 3.1, 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 where *Author(s)* can upload a new firmware to be 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, while the data collected is sent via MQTT. The advantage of using HTTP for the update is that it allows for more bytes per packet, making the update process faster if compared to IoT protocols such as MQTT, especially for the firmware download. Longer update duration might discourage application administrators from performing new updates, in order to avoid disrupting the application, leaving out-of-date devices operating instead.

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. Three of the largest technologies companies, Google, Amazon and

Microsoft, have a commercial IoT platform, and all three have support for MQTT and HTTP, further supporting the argument that familiarity is important for adoption.

The IoT cloud platform used for the experiments is the Konker platform. It 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. This design has the advantage of being similar to a MQTT broker, but with HTTP. This would allow for MQTT usage for updating, if desired, without having to change the architecture.

The messages exchanged during the update process can be seen in Figure 4.1, a state diagram of all the decisions a device make during the update.

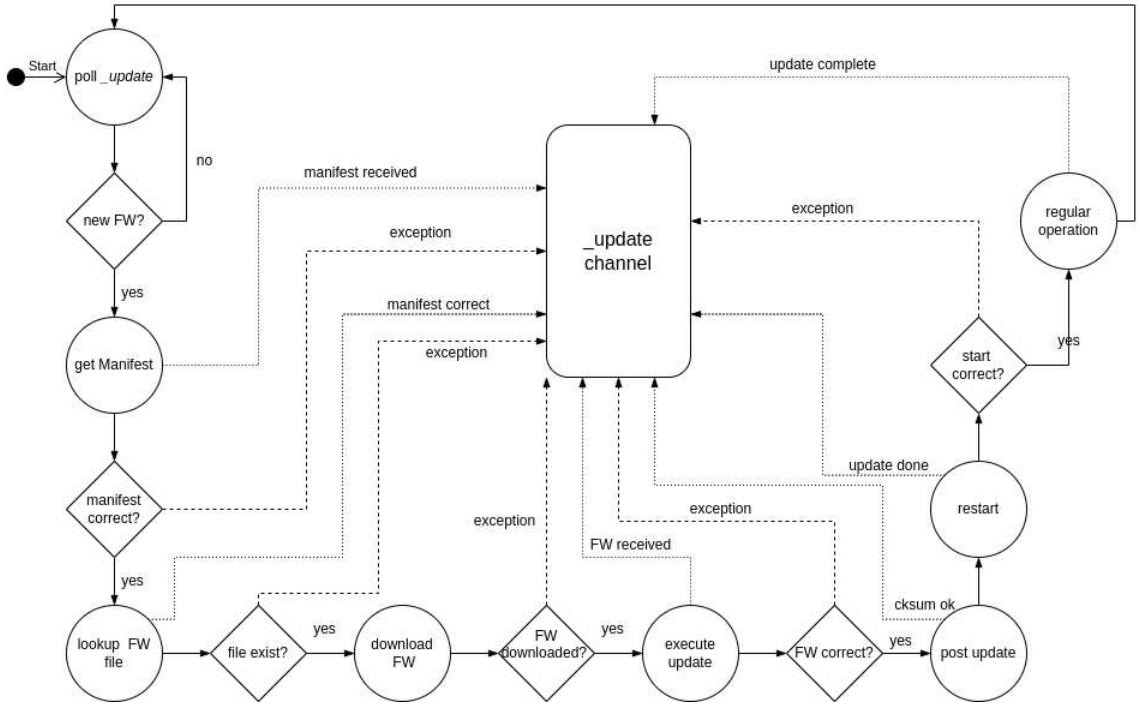



Figure 4.1: Update process state machine and messages sent by the device

The process starts after the *Author* has uploaded the firmware and the manifest to the IoT platform, with the manifest signifying the existence of a new update. If nothing is retrieved, the device will simply try again after a predetermined time interval. The manifest is retrieved from the channel by the device and parsed. If the manifest is correct, the device can request the firmware from the platform, following the hybrid approach to updating, mentioned in Section 3.2 and described in [1]. 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, an exception message with the corresponding step is sent and the update process stops. This way, both the platform and the device knows which steps of the process were performed.

Another way to describe the process is with a sequence diagram. Figure 4.2 is equivalent to Figure 4.1, but better highlight the messages, and make the results that will be presented easier to visualize.

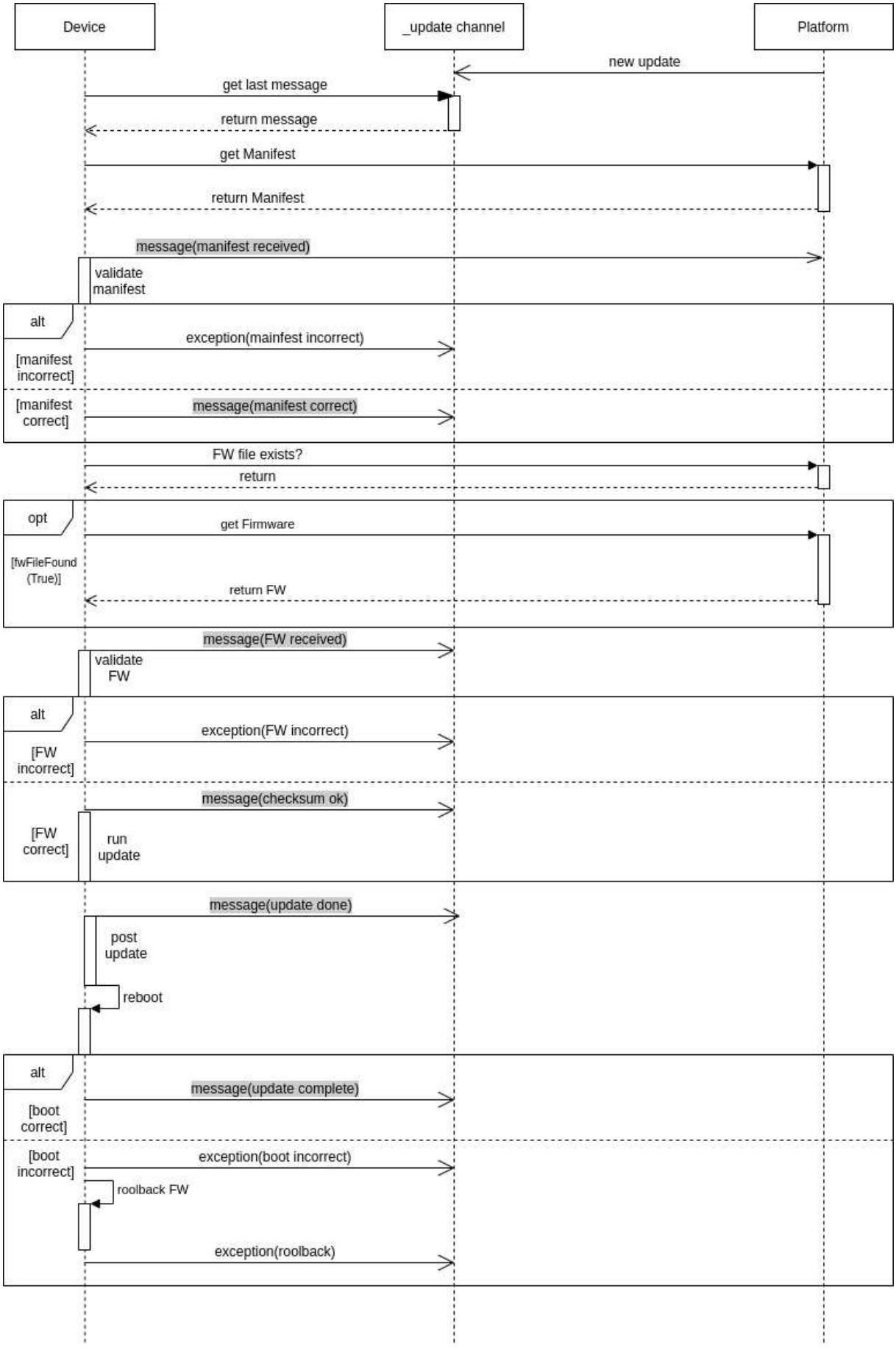



Figure 4.2: Sequence diagram of messages exchanged during update process

### 4.1.2 Firmware design

As a starting point, the existence of a new update is checked every predetermined interval of time, $\Delta T$. This interval depends on the application and the balance between resource usage to check for new update and frequency which they are sent. Algorithm 1 summarizes the algorithm entry point verification at a higher level.

**Algorithm 1** Update check verification

**Require:** $timeSinceLastCheck \geq \Delta T$

```
if there is a new update then
    validate and perform update
end if
```

In a more detailed manner, Algorithm 2 expands on each step. To check for a new update, the device polls for a new message in the *_update* channel, if there is one, and this message is the manifest, then a new firmware is available. From this point, the verification of the manifest can be done. The parsing and verification is detailed in Algorithm 3. It ensures that all the mandatory fields are present, and both mandatory and optional fields (that are present) are correct. If it is legitimate, recent and correct, the update can proceed.

**Algorithm 2** Main update function

```
function otaProcess
    rawManifest ← getManifestFromPlatform
    if rawManifest is not empty then
        send status message → "manifest received"
        manifest ← parseManifest(rawManifest)
        if manifest is valid then
            send status message → "manifest correct"
            if downloadVerifyFirmware(manifest) then
                if installFirmware then
                    send status message → "update done"
                    set firstBoot flag as true
                    restart application
                end if
            else
                send exception message → "update failed"
            end if
        else
            send exception message → "manifest incorrect"
        end if
    end if
end function
```

Once the manifest is determined to be correct, it can proceed with the firmware download and verification, as shown in Algorithm 4. For the NodeMCU, the firmware was transferred in a binary format, and the download was handled by the *ESP8266httpUpdate*

**Algorithm 3** Parsing and verification of the manifest

```
1: procedure parseManifest(rawManifest)
2:     decodedManifest ← decodeJSON(rawManifest)
3:     for field in requiredFields do
4:         if field is in decodedManifest then
5:             if field is valid then
6:                 continue
7:             else
8:                 return Null
9:             end if
10:        else
11:            return Null
12:        end if
13:    end for
14:    for field in optionalFields do
15:        if field is in decodedManifest then
16:            if field is valid then
17:                continue
18:            else
19:                return Null
20:            end if
21:        end if
22:    end for
23:    return decodedManifest
24: end procedure
```

library. The Raspberry Pi firmware is in a *zip* format, and the installation means extracting the files to the correct directory. For this device, it is possible to enable rollback to the previous firmware, in case there is a problem with the new one. For this purpose, a *zip* file is also kept in memory with the previous firmware.

After the download finishes, its integrity is checked against the checksum received in the manifest. If the checksum is correct, then the new firmware information received in the manifest can be stored in memory, both devices store them in the JavaScript Object Notation (JSON) format. The NodeMCU device does have a small file system where this information is stored.

**Algorithm 4** Firmware download and verification

```
1: procedure downloadVerifyFirmware(manifest)
2:     newFirmware ← downloadFirmware
3:     if newFirmware is not empty then
4:         send status message → "firmware received"
5:     else
6:         send exception message → "firmware not found"
7:         return false
8:     end if
9:     if firmware checksum is correct then
10:        send status message → "checksum ok"
11:        save firmware information to memory
12:        return true
13:    else
14:        send exception message → "firmware incorrect"
15:        return false
16:    end if
17: end procedure
```

The architecture allows for finalizing steps to be executed by including them in the manifest, which happens in Algorithm 5. These can be installation timing, any functionality the device might have to turn off before installing, or a script to run. If the finalizing steps terminate correctly, the device can restart and boot to a new firmware.

**Algorithm 5** Run processing steps, if any, and extract new firmware to persistent memory

```
1: procedure installFirmware
2:     if manifest has finalizing steps field then
3:         run instructions from manifest
4:     end if
5:     write new firmware to memory
6:     if new firmware not correctly extracted then
7:         return false
8:     end if
9:     set firstBoot flag as true
10:    return true
11: end procedure
```

**Algorithm 6** During setup procedure, this check is performed

```
1: procedure checkFirstBoot
2:     if firstBoot then
3:         set firstBoot flag as false
4:         send status message → "update correct"
5:     end if
6: end procedure
```

After booting, there is one last step, to inform the platform that all went correctly and the new firmware is installed, signalling the end of the process, as shown in Algorithm 6.

### 4.1.3 Manifest Implementation

For this work, a subset of the required Manifest fields [71] was used, as can be seen in Listing 1. 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 RFC 9124.

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 considering security issues, the field was omitted. The *Payload Format* and *Dependencies* fields do not influence this update, since the only one format is used for each device. 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, or had a proper file system, it was not implemented.

The format chosen for the manifest data is JSON, due to its simplicity to parse and availability of parser libraries for the devices used in the experiments. This format also brings some flexibility to be changed to a more efficient format, in terms of memory footprint, for more constrained devices, the Concise Binary Object Representation (CBOR). The conversion from one format to the other is very straight forward, since CBOR was based on JSON, and there are a number of implementations of such converters available.

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

Listing 1: Minimal manifest example

## 4.2 Implementation aspects

### 4.2.1 Devices

Given the differences between both devices used for experiments, the NodeMCU (ESP 8266) and Raspberry Pi 3, there are notable differences in the implementation, even though the OTA algorithm is the same expressed in Figure 4.1. One of the main difference is that the Raspberry Pi is capable of running an operating system, and has OTA running as an independent application, while for the NodeMCU OTA, it is part of the firmware together with its main application.

Provided that an existing operating system was not used, the entire NodeMCU firmware was implemented to perform the tests. The code uses the Arduino framework and some of its libraries. It works with object-oriented programming, and consists of a main class

that exposes methods for a programmer to use in the `setup()` and `loop()` functions, and several classes that implement the functionalities available.

For usage, the main class `KonkerDevice` is composed of several functions for configuration of platform credentials, server and connection type, and handling a data collection application. For the experiments, the data collected did not come from any sensors, but the structure for real usage is ready, with a buffer to store said data before sending.

**KonkerDevice** Main class.

`Buffer` A circular buffer where data collected is stored before being sent to the platform, this way some data is still kept in case of a connection drop

`WifiManager` Handles WiFi connection and access point

`Protocol` Which protocol to use for sending data from the buffer. More protocols can be added here, if the device supports it

- MQTT: This is the current protocol being used
- HTTP: Available, but not used for sending data at the moment

`File System` Handles read and write from permanent memory. Current stored information are WiFi and platform credentials, and firmware information from the manifest

`Health` Collects and sends information about the device operation. It is useful information for testing and most of it can be disabled when deployed to save resources

`Platform` Handles platform connections and credentials

`Helpers` Wrappers for libraries. So far it handles JSON parsing and NTP

`Update` Handles the entire update process. This module works according to the pseudocode from Algorithms 1 to 6.

For the Raspberry Pi devices, an application running on a Linux distribution was already functioning when OTA was implemented. This application was responsible for receiving the data collected by sensors and sent to the Raspberries via LoRAWAN, this data was then added to a queue to be then sent to the platform.

Previously, these devices had to be updated manually, accessing each individually via SSH, transferring a new application and activating it via a bash script. This new implementation made the process a lot faster and less error-prone.

The update process was implemented as a Python script that periodically ran to check for a new update. The process that runs the script is controlled by the *supervisor* program. Since the script is not responsible for all other functions, it is a more straightforward implementation than the NodeMCU. The biggest difference is that the update process does not necessarily update itself, it can be limited to updating the application. This is the case for the tests performed in this dissertation, only the application was replaced.

With regard to the activation of the new firmware, the NodeMCU bootloader is responsible for handling memory banks where the old and new firmware are stored, and

booting to the correct one. To start the new firmware, the Raspberry Pi device does not need to restart, it only needs to restart the application that was updated.

For the NodeMCU, the verification of the new firmware activation is done once in the `setup()` function. For the Raspberry Pi, this last step happens the next time the OTA process is run. Both use a flag in memory to signal that an update happened.

### 4.2.2 Platform

As mentioned before, the platform used for device management is the Konker IoT platform. From the platform web interface it is possible to register devices, manage credentials and monitor them. It also has other functionalities for dealing with devices, but are not relevant for this work.

In it, there is an interface to supervise all the messages received from each device, alongside their timestamp and the MQTT channel it was sent to. Figure 4.3 shows a sample of messages, specifically update steps messages.

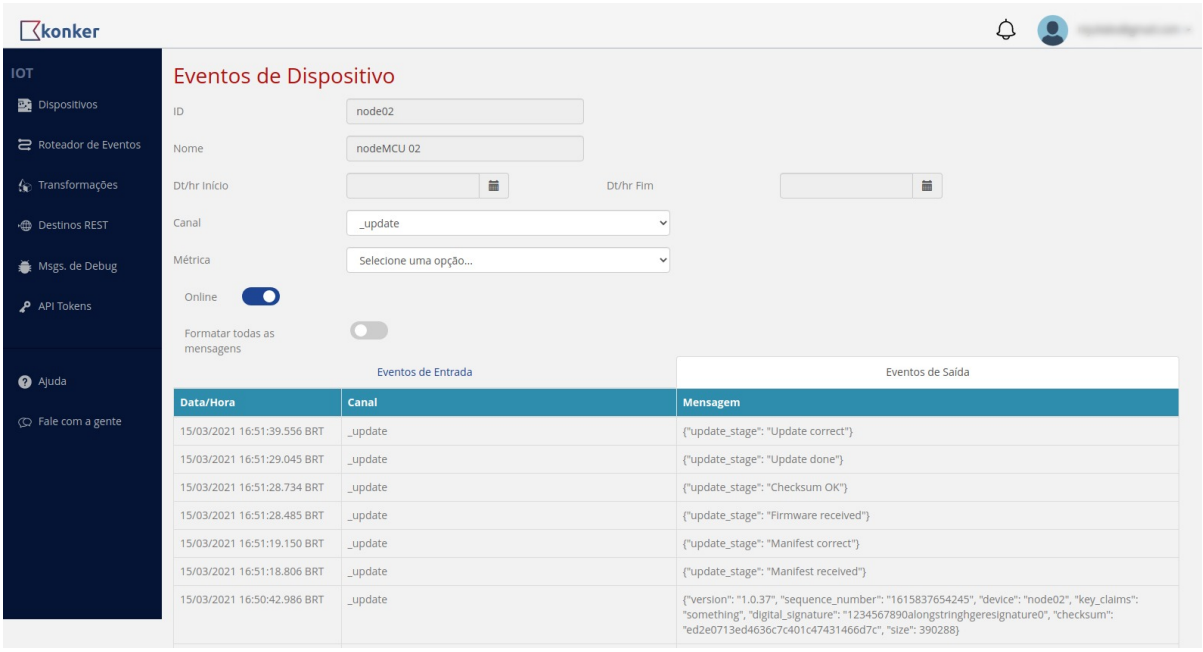



Figure 4.3: Capture of Konker platform web interface

Some device related functionalities are available through an API. It has a web interface as well, but its main intent is to be used by scripts to automate tasks. For example, instead of registering devices manually, it is possible to have a script that automates the HTTP requests. Figure 4.4 shows which categories of operations are available, and the specific *device firmwares* endpoints.

There are two steps to creating a new update. Firstly, */application/firmwares/deviceModelName* is where the firmware is uploaded, with a checksum and its version. The checksum is used by the platform to verify the firmware integrity. Once the new firmware exists within the platform, individual devices that will receive it are designated in the */application/firmwareupdates/* endpoint.

These steps do not signify to the devices that a new update is available, this happens after the manifest is sent to the platform by publishing it to the specified channel, from which the devices will check for a new manifest, and consequently a new update.

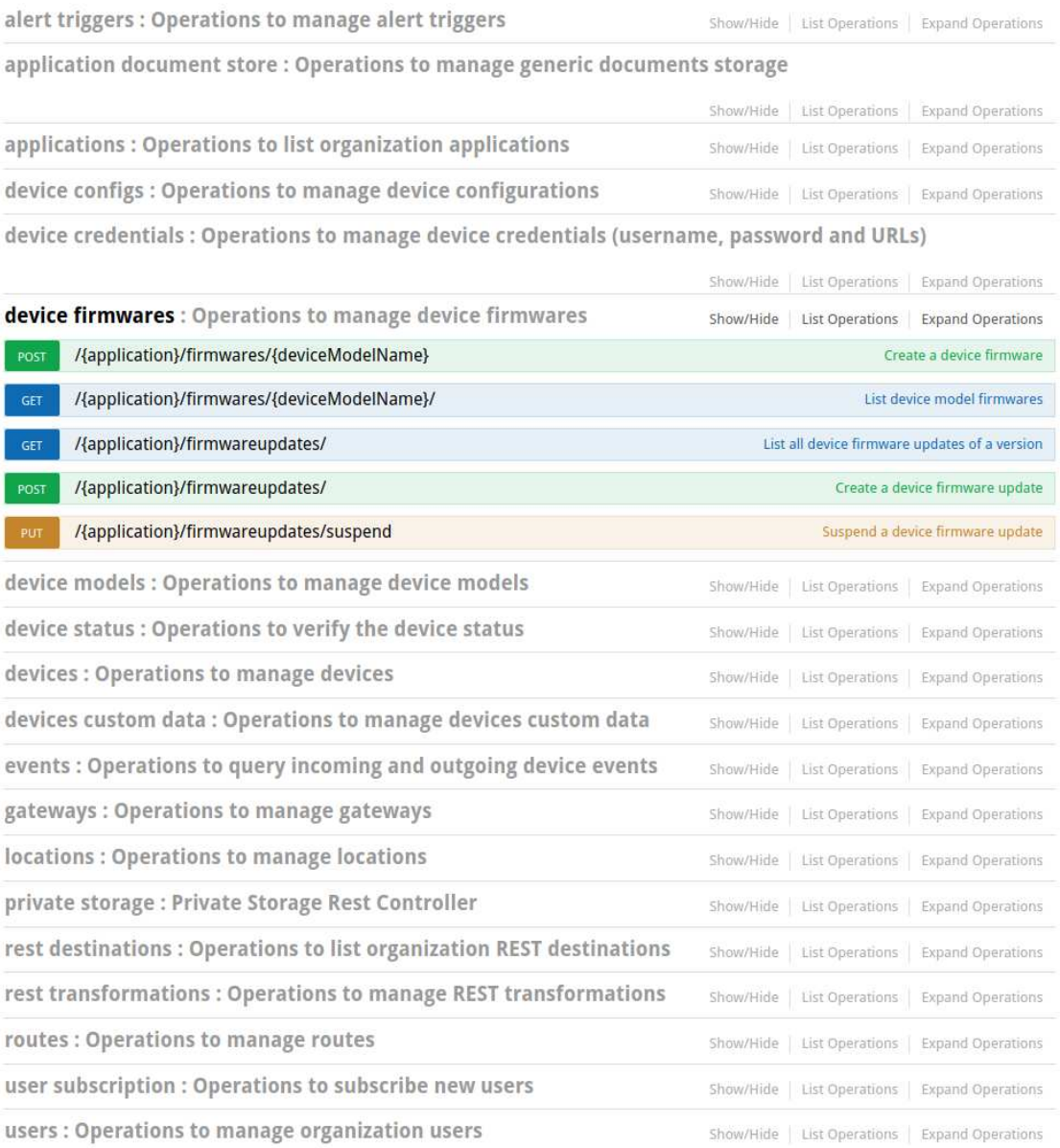



Figure 4.4: Capture of Konker API web interface

## 4.3 Qualitative evaluation

Earlier versions of the SUIT draft (up until version 12) contained 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 4.1 explains each item in the list, its description and how it was implemented.

| 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's 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 1). *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 an 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 adapted to receive other forms of payloads. |

Table 4.1: Comparing requirements proposed by IETF to current implementation

Besides these requirements, it is interesting to analyze in more details how the process handles problems (that incurs in the *high reliability* requirement). During the update process, many failures can happen, and having mechanisms to deal with them is essential to keep the application running. Some mitigation is already in place, either being already provided by the RFC architecture, or by implementing it when necessary. But there are improvements that can be done, specially on the side of the platform.

With problems that are not caused by loss of connection, the status messages can be

used to detect problems (the exceptions in Figure 4.1). At the moment, the platform only serves the firmware and receive the messages, but does not take any action based on them. The platform can be improved to respond to the status messages that are already being sent. For a list of failures, their causes and possible solutions, see below:

**Manifest not received** Happens when the platform responds with something unexpected (not a manifest or status message). The solution is to resend the manifest.

**Manifest incorrect** This can have more than one cause, it can be a malformed JSON that cannot be parsed, it can be missing a mandatory field, or a field might not be valid, for example, the version of the manifest is older than the current one in the device. Whatever the case, the solution is to correct and resend the manifest. Correcting it might need manual intervention.

**Firmware not found** Happens when the firmware endpoint does not respond. It can be solved by sending a new manifest with a new endpoint, in case it has changed. If the problem is with the endpoint, the correction is on the side of the platform.

**Firmware incorrect** When the checksum is incorrect. One cause can be an incorrect download, in which case downloading again is the solution, but if the problem was with the manifest the entire process needs to restart. Since this can have two origins, detecting the proper solution is the bigger challenge.

**Update failed** This is a problem with the finalizing steps, when they exist, and has the most implementation specific causes and solutions. In the case of this work, only the Raspberry Pi devices received any instructions, and detection of errors and solutions depend on the instruction itself.

**Started incorrectly** Happens when there is a problem with the new firmware, and it couldn't start correctly. The solution is a firmware rollback, if the device has that capacity, otherwise it may necessary to manually update a working firmware to the device.

In the case of a device disconnection that still leaves the device on, due to a bug in the firmware or something else, the messages indicating the update steps stops arriving at the platform. If the device is still sending other types of messages, such as data or health messages, it means that it is still on and communication is working. In this case, the solution can be based on the last update step message that arrived, from restarting the update process by sending a new manifest, if the failure happened before the firmware download, or resuming the process from the download if it happened after.

If the disconnection was caused by a power failure, several things can happen to the device: if it can turn on again and functions normally, the update process start from the beginning or resume from the point it stopped. But if the device can not turn on again, or the power failure caused the device to brick (for example, with corrupted memory), then manual intervention might be the only solution, by plugging a new battery, manually restoring the firmware, or replacing the entire device. In this case, all types of messages will stop arriving, which is a strong indicator of a more serious failure.

Another source of failures can be a network problem, an indication of this being the source is the amount of devices experiencing the same problem and their geographic location. Network problems can cause messages to delay, or be lost, which is effectively a disconnection, though it might be hard to differentiate between both. In the case of delays, simply waiting for the network to return to normal and the delayed messages can be enough, the challenge is to determine how long to wait. This depends on which step of the update was the last, or which type of application the device runs, where its cruciality determines how long it is acceptable to have no communication.

The importance of security together with some solutions from the literature were already discussed in Chapters 2 and 3. It is also important to discuss this implementation in terms of security. Though it was not the focus of this work, some security problems are tackled. The security mechanisms and possible attacks mitigation are present in the implementation will be discussed next.

The main security goals discussed previously were confidentiality, integrity and availability. On the side of the platform, there is authentication and permissions handling for the *author*, only allowing an authorized person to initiate the update process, which contribute to improve confidentiality by not providing information about devices to unauthorized parties. It contributes to integrity by not allowing anyone to upload a potentially harmful firmware to the platform. Lastly, it contributes to availability by requiring authorization to manage devices, not allowing an attacker to remove devices from the network through the platform.

On the device side, the main concern is that the correct device (confidentiality) correctly installs the correct firmware (integrity). Having ensured that, the device remains available for the network. Most of the manifest fields serves these purposes: 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 altered firmware from being installed when there is a valid manifest. It can happen with attacks where an insecure version of the firmware is sent, or by some transmission error, the firmware is received incorrectly.

From the manifest description [71], a list of types of attacks is presented specifically for firmware transport, its related security requirements, and how to prevent it with one or more manifest fields that satisfies them. The threats the implementation is protected against were discussed in the previous paragraph, the remaining depends on infrastructure beyond the manifest, such as securing keys, which have a direct correlation to the threats this implementation is not protected against. They depend on signing the manifest and encrypting the payload, that protects against instance traffic interception and tampering.

Some threats' mitigation are simpler to handle, being a matter of implementing the related manifest field. For example, adding an expiration date (that is considered an optional field) to avoid installation of an old firmware on a device that has been offline for a long time.

A more indirect manner of securing is having the platform as the only trusted firmware

source coded into the firmware. Redirecting it would require the installation of a firmware that points to a malicious server, by which point the device is already vulnerable.

Despite the security issues, most can be corrected by adding fields to the manifest and implementing their verification, therefore adding few more possible exceptions to Figure 4.1, for example, an exception for lack of encryption keys. But there is no need to change the architecture, these features are already accounted for by virtue of being based on RFC 9019.

The implementation of signature verification, decrypting and more manifest fields improve security, but do take a toll on the devices, requiring more testing to measure the impact. Other authors have done so, and found that it does require a significant amount of time, memory [55] and energy [41], but were able to successfully implement in devices more constrained than the ones used for this work.

# Chapter 5

# Experiments and Results

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;
3. to evaluate the impact of performing the OTA update on devices running a real-world deployed application.

Both experiments were performed using the Konker IoT platform described in Section 4.2.2. It is an Open Source platform developed by Konker Labs [1]. The cloud hosted version of the platform is available[2], with its API[3].

The details of each experiment and their results are show in the following sections. First, the configuration and setup of the LAN and WAN experiments are delineated in Section 5.1. The results and conclusions are in Sections 5.2 and 5.3, respectively.

## 5.1 Experimental evaluation

### 5.1.1 LAN evaluation

#### Environment and configuration

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

[1] `https://www.konkerlabs.com`
[2] `https://prod.konkerlabs.net/registry`
[3] `https://api.prod.konkerlabs.net/v1/swagger-ui.html`

The IoT devices were implemented using the NodeMCU 1.0, with an ESP8266 WiFi chip (80MHz, 4MB of flash memory, and 64KB of SRAM). In order to simulate an IoT application collecting sensor data, the IoT devices are configured to send a piece of data to the IoT cloud platform every $5s$ using the MQTT protocol. The firmware, containing the OTA implementation, was developed using the *Arduino* framework for PlatformIO, version 2.6.2, in C++ and can be found in a GitHub repository [4].

The implementation relies heavily on messages exchanged during the update process. The messages used for timing the update steps displayed in the results from Sections 5.2 and 5.3 are in Figure 4.2, from the previous chapter.

The *Arduino* framework library that implements an OTA update process mentioned in Section 4.2, *ESP8266httpUpdate*, was used for the firmware download ("get Firmware" from Figure 4.2) and for writing it to memory ("install FW"), but the other steps for checks and validations, and the messages exchanged were implemented to comply with the RFC. The size of the new firmware received by the devices was 390,288 bytes, and the size of the manifest was 216 bytes. The RFC 9019 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.

In order to keep the entire architecture in the same LAN, instead of using the cloud-hosted platform, a computer was used to host it, making it a more controllable environment to evaluate the results. The computer the platform was running on is a Thinkpad T420, with an Intel i5-2520M 3.2 GHz, 12 GB of RAM and running Linux Mint 19.1 (kernel 4.15). Both the web interface and the API ware available locally and Figure 5.1 illustrates the testbed.

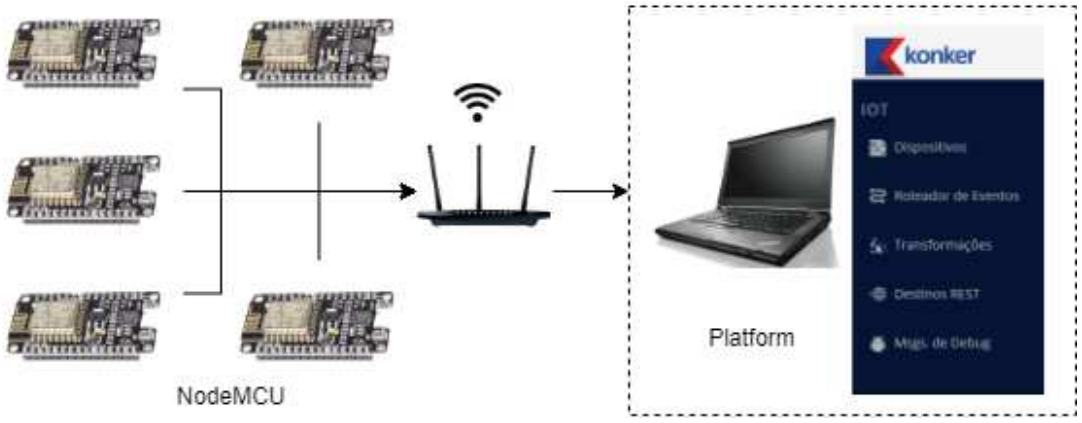



Figure 5.1: LAN experiments setup

### Methodology of the Experiments

In total, 4 test scenarios were performed, 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.

[4] `https://github.com/lmcad-unicamp/libKonkerESP`

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), is not enough to overflow the network, but it is a number closer to what can be found in real world implementations of IoT solutions using WiFi communication. Even though a network 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 by comparing them. 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 [42] and the device [55] to measure the update impact on the network and how it might influence the devices. The network measurements were made in the computer running the platform, where Wireshark[5] was left running and capturing the traffic.

The process, from receiving the manifest to finish rebooting a new firmware, was divided in 7 steps. During each step of the update, the device collected information about its resources usage and stored everything in memory. Once it boots to a new firmware, this information is sent to the platform. This way, the transmission of this information does not interfere with the update process.

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 an NTP (Network Time Protocol) server, from where devices synchronized their internal clocks. Time starts counting when the manifests are sent (a "new update" from Figure 4.2) 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 $\Delta T = 30s$ (the predetermined interval from Algorithm 1) and the devices are not synchronized in this manner. This interval worked well for experiments, but may not be suitable for real deployment due to resource usage.

The messages in between used for timing the update steps are highlighted in Figure 4.2 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;

[5] `https://www.wireshark.org`

| City | Number of devices |
|---|---|
| São Paulo | 32 |
| Guarulhos | 7 |
| Osasco | 5 |
| Santo André | 4 |
| Jundiaí, 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 |

Table 5.1: Distribution of devices by city

vii. *Correct* after a device restart.

### 5.1.2 WAN evaluation

#### Environment and configuration

In the WAN experiments, the devices used were Raspberry Pi 3, 1.2 GHz Quad Core processor, 1 GB of RAM and the firmware was implemented using Python 3[6]. The firmware code is available in a GitHub repository [7] together with the code to generate a new update and manifest. Out of a total of 75 devices, most are spread across the São Paulo and its neighboring cities with a few across the state of São Paulo, in establishments from a restaurant chain. The complete list of devices by city is in Table 5.1. The same IoT platform was used, but in this case, it was hosted remotely in the 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. In terms of the IoT architecture, the Raspberry Pies can be considered part of the middleware layer, since they aggregate the data from several devices.

#### Methodology of the Experiments

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

[6]Linux kernel version 4.19 and Python version 3.5.3

[7]`https://github.com/Majubs/update_rpi3`

firmware, the first scenario had an earlier version, where the messages from Figure 4.2 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 5.2 shows the difference between both, while the new firmware sent all of them, the older version sent the ones indicated with a check mark. In the second scenario, all the status messages for each step in Figure 4.2 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 people 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.

| **Step** | **Step** | **Resources Usage** |
|---|---|---|
| Polled | ✓ | ✗ |
| Manifest received | ✗ | ✗ |
| Manifest correct | ✓ | ✓ |
| FW received | ✗ | ✗ |
| Checksum ok | ✓ | ✗ |
| Update done | ✗ | ✓ |
| Update correct | ✓ | ✓ |

Table 5.2: Status messages and information sent during update process

Each Raspberry Pi is powered through a wall socket, so energy saving is not the main concern. They can connect 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 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, 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.

The difference in the number of messages presents an interesting discussion. On one hand, sending less messages avoids collisions in the network and uses less energy from the device. On the other hand, it provides less information for debugging when something is not working properly. We argue that, since updates are an occasional thing, the extra information is worth the extra bandwidth. Though, the choice still remains to whoever is implementing a similar solution. In our experiments, the devices have ample access to energy and the messages are small enough for WiFi (which might not be the case for other protocols), so having the extra information brings more benefits than disadvantages.

## 5.2 LAN results and analysis

Figure 5.2 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 steps' duration remains almost the same for all scenarios of test. It only increases with the number of devices during the firmware download step, as the number of bytes sent is multiplied by the number of devices.

Test duration results presented in Table 5.3 show that the entire test duration for each scenario does not take longer than 11 minutes, but, as explained in Section 5.1.1, the devices were left running before and after the update, for 9 minutes, which means the update process, from sending the manifests to all devices to confirming everything is correct after rebooting, takes between 1 and 2 minutes for all scenarios of tests, that represents 12.5% of total test time for the first scenario and approximately 17% for the other three, as is shown in the last row of the table. There is only a small increment in that time, even when comparing 5 devices to 20, it only increases by 6.05%.

By the results presented in Table 5.4 it becomes clear that downloading the firmware is

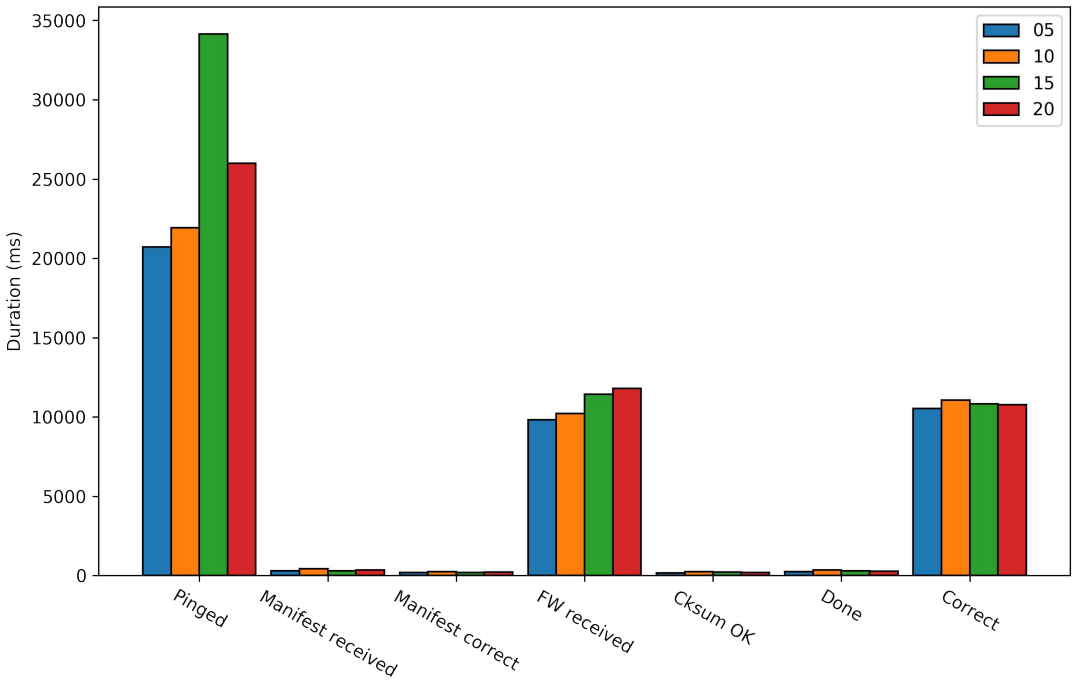



Figure 5.2: Average duration of each update step

| | **Test scenarios** | | | |
|---|---|---|---|---|
| **Metric** | **05** | **10** | **15** | **20** |
| Packets | 13,248 | 26,693 | 38,525 | 51,074 |
| Bytes | 2901910 | 5,892,006 | 8,681,784 | 11,540,824 |
| Test duration ($s$) | 617.244 | 651.199 | 652.803 | 654.624 |
| Update duration % | 12.51 | 17.08 | 17.28 | 17.51 |

Table 5.3: Data collected from Wireshark - totals

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" (JSON). Even in "Miscellaneous" data type, that includes data such as NTP requests, requests for the *_update* channel or devices sending health information, it still represents fewer bytes than the firmware download.

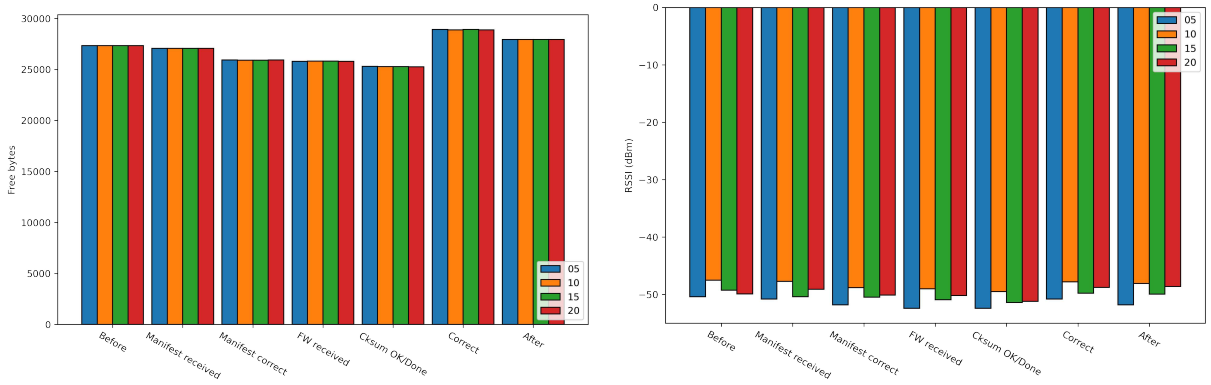
Right before the start of the update process, during each update step, and right after it finished, health information about the update (available memory and WiFi strength) were collected in each device and are presented in Figure 5.3. Since all devices are running the same firmware, the memory available varies equally through test scenarios (Figure 5.3a). 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

| | | Data type | | | |
|---|---|---|---|---|---|
| | **Scenario** | **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 |

Table 5.4: Percentages relative to the total in Table 5.3

is no larger than 2KB, making it efficient even for more constrained devices than the NodeMCU.

The WiFi strength, in Figure 5.3b, has a small variability between test scenarios as well, maintaining a good baseline of -50dBm. This shows that interference effects resulting from the increase in the number of devices could be dismissed when analyzing the results of these tests.




(a) Available memory

(b) WiFi strength

Figure 5.3: Average amount of available memory and WiFi strength

Considering the entire test (including the data collection), Figure 5.4 shows the average latency for data transmission, with its respective standard deviation, 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.

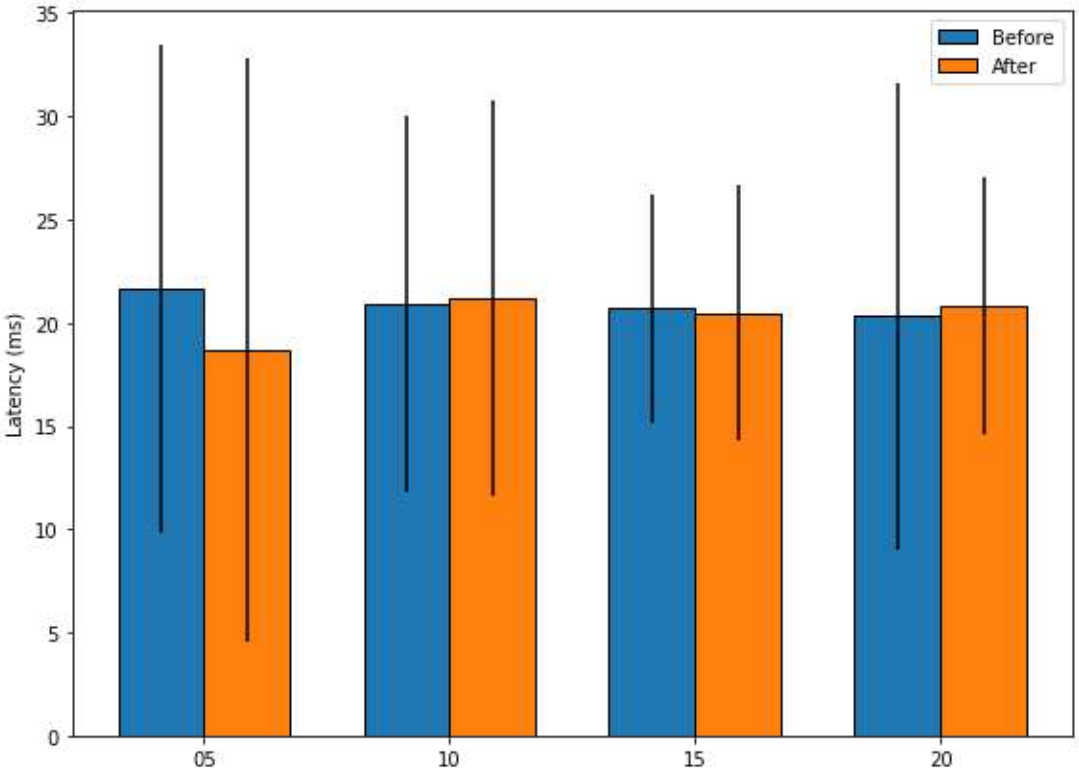



Figure 5.4: Average message latency before and after the update

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.

## 5.3 WAN results and analysis

As explained in Section 5.1.2, there were 75 devices in this experiment. Since in this context, 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 hours that were given to the test. A few responded to the manifest, but failed somewhere during the process. The 10 devices for the second test were chosen in a manner that they all at least answered to the manifest.

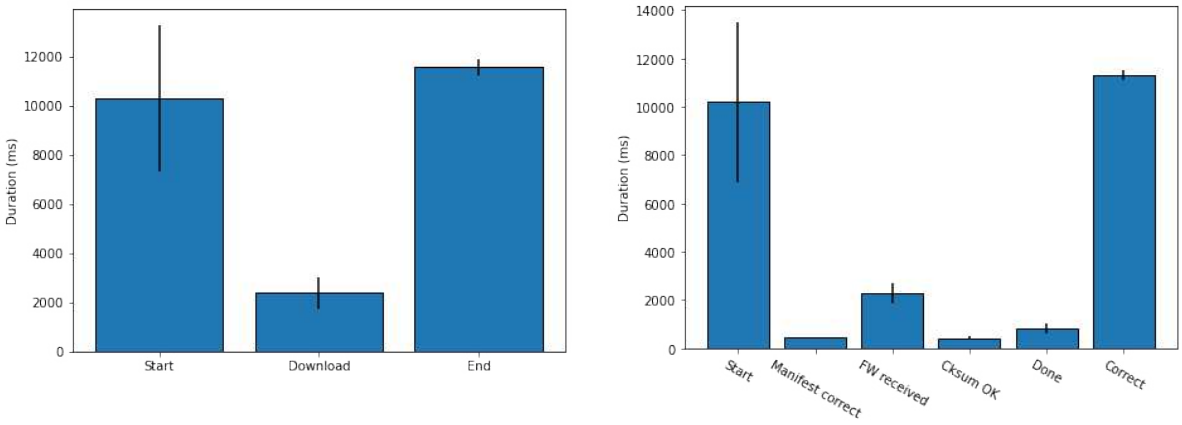
Table 5.5 shows the number of devices in each situation. In total, it was an 80% success rate for the first test scenario and 70% for the second.

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

Table 5.5: Distribution of successes and failures

It was not possible to retrieve all the data described in Section 5.1.2 for all devices, of the 75, only 68 have complete data, and it is from them that the following results were extracted.

Similar to Figure 5.2, Figures 5.5a and 5.5b shows the average duration of each step. In the case of Figure 5.5a, since not all steps have a message, some steps were incorporated together, according to Table 5.2. *Start* includes up until the manifest is correctly received, *Download* includes the firmware download and its verification, while *End* includes any post-processing and the time it took for devices to restart.




(a) Average step duration for 68 devices

(b) Average step duration for 10 devices

Figure 5.5: Average duration of each update step

The step's duration distribution is also similar to Figure 5.2, the biggest difference is that the Raspberry Pi devices query the platform for the manifest every 10 seconds instead of 30. The average total duration of the update for the first test was $24.24s \pm 3.10$ and for the second test was $25.45s \pm 3.45$, which are very similar values.

From the test with 10 devices, the data about device behavior was captured. Regarding memory usage, its variation through the update was almost negligible, which is expected since the update size is orders of magnitude smaller than the device memory. Meanwhile, the CPU utilization did show a small increase for manifest parsing, and a significant increase after downloading the firmware, as seen in Figure 5.6, 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 cases,

there were two attempts at performing the update. While in one of the attempts, device 02 had a similar behavior up until it failed to retrieve it, device 03 had a higher CPU utilization in both tries, in the second attempt the increase in utilization during manifest parsing was a lot more steep than the average. 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% of the time, added to that, they continued communicating with the platform after the failure, showing that the connection did not completely drop, but could be unstable. An intermittent connection could have caused the failure at the download step, and may even explain the higher CPU utilization, that could be caused by other processes in the device trying to deal with it.

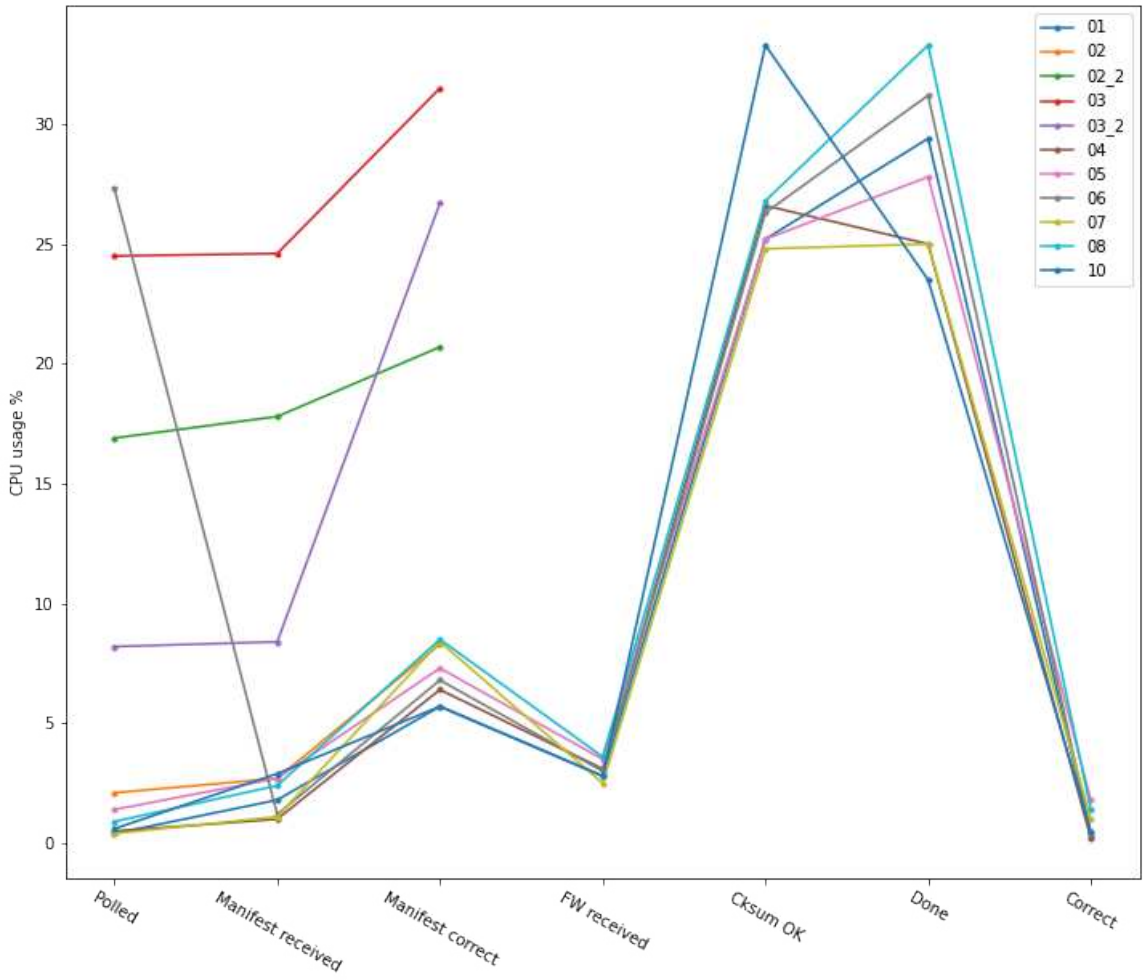



Figure 5.6: CPU utilization during the update for 10 devices

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.

Concerning these three devices and comparing to the first scenario, both 02 and 03 completed the update, but 09 failed in the checksum check, after firmware download.

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 information provided by the messages (or lack thereof) from Figure 4.2. This data can provide mitigation or solutions for these problems, helping improve the solution that is built on top of the RFC architecture.

# Chapter 6

# Conclusion

Updating the software of IoT devices is a critical part of the IoT life cycle in order to keep devices relevant with new features and secure against attackers. This work presented the advantages of maintaining devices up-to-date and the possible consequences of failing to do so. The challenges faced by IoT implementations as a whole were also presented, and a layered architecture was introduced as a way to visualize the elements of an IoT solution and how to tackle said challenges. Each layer purpose was described, alongside the technologies that are used in each one.

A summary of the state-of-the-art of OTA software update solutions and research for IoT was given, where challenges specific to OTA were explored, with special attention to fault tolerance and security issues and solutions. A table comparing the discussed solutions was used to consolidate the main points of research, as well as comparing these solutions with this work. The OTA architecture proposed by the IETF SUIT working group, in the form of RFC 9019, was explained in more details, alongside its accompanying manifest proposal.

Given this context, this dissertation proposed an OTA firmware update solution for constrained IoT devices based on the IETF standard architecture. The design of the proposal is described, explained and illustrated by diagrams, a state machine diagram and a sequence diagram for the message flow among the involved entities. At the implementation level, the algorithms for the IoT devices were discussed by presenting the pseudocode.

One form of evaluating the solution was a qualitative comparison with requirements from earlier versions of the RFC, as well as discussing how the devices handle fault detection and correction, and which security requirements are fulfilled.

With the goal of evaluation the proposed solution performance, two different scenarios were tested: one testbed including up to 20 IoT devices in a LAN setting, with four different scenarios, and another with 75 devices deployed in different cities as part of a real application monitoring restaurants, with two different scenarios. All the developed code has an open source license and can be found on GitHub. 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. The success rate of the LAN experiment was 100% and the WAN, an average of 75%, due to connection conditions.

In summary, the contributions were the design and implementation of an OTA update

solution based on the RFC 9019 specification, and its evaluation in two scenarios, with constrained and non-constrained devices. The code for both implementations is also available as an open-source library.

## 6.1 Lessons Learned

Since IoT is a fairly recent consolidated field of study, there are not many standardized ways of developing solutions. Different authors choose different requirements to focus on, generating very distinct solutions in terms of complexity and scope. This was the first hurdle when planning how to classify and choose requirements, and the use of the RFC 9019 is a good start, both in terms of narrowing scope and in terms of defining a standard for OTA solutions.

One of the main advantages of the RFC architecture was the flexibility to apply it to very different settings and adapt to divergent requirements without becoming incompatible, where more specific solutions might fall short when used in different contexts, requiring re-engineering of the solution. In this case, despite quite different code implementations, both had the same underlying process and flow, without the need to reformulate the architecture.

The disparity of success rates in the LAN and WAN settings highlight the difference between theory and practice. In one setting, there was only one entity making decisions and with total control of all the parts. This is important for testing the viability of a solution, but it is incomplete when considering solutions that will be used by companies and people. The WAN setting was a more approximate demonstration of this scenario, where multiple entities had control over parts of the architecture: the restaurant chain had control over devices' usage, the authors of the firmware design and implementation, and even the platform had more stages to go through before adding a new feature. Though this setting does not emulate multiple decision-making parties, such as devices manufactures and vendors, clients or data analysts.

The decision of deploying an update happens before anything in the scope of this work, but it does not mean the results do not provide useful insights about how to facilitate the implementation of an OTA process. Having a library that is compatible with a platform and can be used by different devices facilitate implementation, and may even be a selling point for a platform provider.

## 6.2 Limitations and Future Work

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. So far, the platform receives the messages of the update process, but does not act on them, doing so would further automate the management of the devices. The most straightforward in terms of implementation is a retrial system for devices that responded to the manifest, but failed during the update.

One of the main aspects that could be improved in this work is the improvement

of security features, for example, signing the manifest and encrypting the firmware, in addition to 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 ideal to measure how it impacts the implementation, mainly compared to the results presented in this work.

# Bibliography

[1] Brendan Moran, D. Brown, M. Meriac, and H. Tschofenig, "A Firmware Update Architecture for Internet of Things," RFC Editor, Request for comment 9019, Apr. 2021, access date: 2020-03-11. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9019.html

[2] Ericsson, "IoT connections forecast – Mobility Report," Nov. 2020, access date: 2021-04-28. [Online]. Available: https://www.ericsson.com/en/mobility-report/dataforecasts/iot-connections-outlook

[3] E. Ahmed, I. Yaqoob, A. Gani, M. Imran, and M. Guizani, "Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 10–16, Oct. 2016.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[5] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, 2012, pp. 257–260.

[6] Qiang Wang, Yaoyao Zhu, and Liang Cheng, "Reprogramming wireless sensor networks: challenges and approaches," *IEEE Network*, vol. 20, no. 3, pp. 48–55, May 2006.

[7] S. Brown and C. Sreenan, "Software updating in wireless sensor networks: A survey and lacunae," *Journal of Sensor and Actuator Networks*, vol. 2, no. 4, pp. 717–760, 2013.

[8] J. L. Hernández-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta, "Updating IoT devices: challenges and potential approaches," in *2020 Global Internet of Things Summit (GIoTS)*, Jun. 2020, pp. 1–5.

[9] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification of firmware updates over the air in vehicle ECUs," in *GLOBECOM Workshops, 2008 IEEE*, 2008, pp. 1–5.

[10] P. Ruckebusch, E. De Poorter, C. Fortuna, and I. Moerman, "GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules," *Ad Hoc Networks*, vol. 36, pp. 127–151, Jan. 2016.

[11] S. Schmidt, M. Tausig, M. Hudler, and G. Simhandl, "Secure firmware update over the air in the internet of things focusing on flexibility and feasibility," in *Internet of Things Software Update Workshop (IoTSU). Proceeding*, 2016.

[12] L. D. Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, Nov. 2014.

[13] J. d. C. Silva, M. L. Proença Jr., and J. J. P. C. Rodrigues, "IoT Network Management: Content and Analysis," *XXXV SIMPÓSIO BRASILEIRO DE TELECOMUNICAÇÕES E PROCESSAMENTO DE SINAIS*, 2017.

[14] M. Frustaci, P. Pace, and G. Aloi, "Securing the IoT world: Issues and perspectives," in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*. Helsinki, Finland: IEEE, Sep. 2017, pp. 246–251.

[15] 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.

[16] B. K. Mohanta, D. Jena, U. Satapathy, and S. Patnaik, "Survey on IoT security: Challenges and solution using machine learning, artificial intelligence and blockchain technology," *Internet of Things*, vol. 11, p. 100227, Sep. 2020.

[17] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*. Vienna, Austria: IEEE, Oct. 2017, pp. 1–7.

[18] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and others, "SmartSantander: IoT experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014.

[19] Marco and Mirisola Davide and A. a. G. Acquaviva, "Firmware update for 6LoWPAN networks of OMA-LwM2M IoT devices," Ph.D. dissertation, POLITECNICO DI TORINO, 2018.

[20] A. Thantharate, C. Beard, and 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)*. Atlanta, GA, USA: IEEE, Jul. 2019, pp. 1065–1070.

[21] J. Tournier, F. Lesueur, F. L. Mouël, L. Guyon, and H. Ben-Hassine, "A survey of IoT protocols and their security issues through the lens of a generic IoT stack," *Internet of Things*, vol. 16, p. 100264, Dec. 2021.

[22] C. Bormann, M. Ersue, and A. Keränen, "Terminology for Constrained-Node Networks," RFC Editor, Request for Comments RFC 7228, May 2014, access date: 2022-03-22. [Online]. Available: https://datatracker.ietf.org/doc/rfc7228

[23] S. El Jaouhari and E. Bouvet, "Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions," *Internet of Things*, vol. 18, p. 100508, May 2022.

[24] H. Gupta and P. C. van Oorschot, "Onboarding and Software Update Architecture for IoT Devices," *2019 17th International Conference on Privacy, Security and Trust (PST)*, pp. 1–11, 2019.

[25] P. Ruckebusch, S. Giannoulis, I. Moerman, J. Hoebeke, and 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*, vol. 3-4, pp. 104–119, Oct. 2018.

[26] Connectivity Standards Alliance, "ZigBee Specification," 2017.

[27] G. Montenegro, J. Hui, D. Culler, and N. Kushalnagar, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," Internet Engineering Task Force, Request for Comments RFC 4944, Sep. 2007, access date: 2022-09-29.

[28] Thread Group, "Thread Speification," access date: 2022-10-04. [Online]. Available: https://www.threadgroup.org/

[29] FieldComm Group, "WirelessHART | FieldComm Group," access date: 2022-09-29. [Online]. Available: https://www.fieldcommgroup.org/technologies/wirelesshart

[30] Bluetooth Special Interest Group, "Bluetooth Core Specification 4.0," access date: 2022-10-25. [Online]. Available: https://www.bluetooth.com/specifications/specs/core-specification-4-0/

[31] M. Beale, H. Uchiyama, and J. C. Clifton, "IoT Evolution: What's Next?" *IEEE Wireless Communications*, vol. 28, no. 5, pp. 5–7, Oct. 2021, conference Name: IEEE Wireless Communications.

[32] LoRa Alliance, "What is LoRaWAN," 2015. [Online]. Available: https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf

[33] B. H. Corak, F. Y. Okay, M. Guzel, S. Murt, and S. Ozdemir, "Comparative Analysis of IoT Communication Protocols," in *2018 International Symposium on Networks, Computers and Communications (ISNCC)*. Rome: IEEE, Jun. 2018, pp. 1–6.

[34] T. Pinho and M. L. Pardal, "UpdaThing: um sistema de atualizações seguro para a Internet das Coisas," *INForum*, 2016.

[35] OASIS MQTT Technical Committee, "MQTT Specification," access date: 2022-07-13. [Online]. Available: https://mqtt.org/mqtt-specification/

[36] P. G. V. Naranjo, Z. Pooranian, M. Shojafar, M. Conti, and R. Buyya, "FOCAN: A Fog-supported Smart City Network Architecture for Management of Applications in the Internet of Everything Environments," *CoRR*, vol. abs/1710.01801, 2017.

[37] I. Z. Ofer Gayer, Or Wilder, "CCTV DDoS Botnet In Our Own Back Yard," 2015. [Online]. Available: https://www.incapsula.com/blog/cctv-ddos-botnet-back-yard.html

[38] M. A. Dave McMillen, "Mirai IoT Botnet: Mining for Bitcoins?" 2017. [Online]. Available: https://securityintelligence.com/mirai-iot-botnet-mining-for-bitcoins/

[39] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "IoT goes nuclear: Creating a ZigBee chain reaction," in *Security and Privacy (SP), 2017 IEEE Symposium on*, 2017, pp. 195–212.

[40] S. K. Datta and C. Bonnet, "A lightweight framework for efficient M2M device management in oneM2M architecture," in *Recent Advances in Internet of Things (RIoT), 2015 International Conference on*, 2015, pp. 1–6.

[41] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter, "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles," *IEEE Communications Magazine*, vol. 58, no. 2, pp. 35–41, Feb. 2020.

[42] S. Unterschutz and 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*, pp. 27–32, 2012.

[43] Z. Yang, M. Li, and W. Lou, "R-Code: Network Coding Based Reliable Broadcast in Wireless Mesh Networks with Unreliable Links," in *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. Honolulu, Hawaii: IEEE, Nov. 2009, pp. 1–6.

[44] S. Brown and C. J. Sreenan, "Updating software in wireless sensor networks: A survey," *Dept. of Computer Science, National Univ. of Ireland, Maynooth, Tech. Rep*, pp. 1–14, 2006.

[45] TinyOS, "TinyOS," access date: 2019-12-02. [Online]. Available: http://tinyos.net/

[46] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys '04*. Baltimore, MD, USA: ACM Press, 2004, p. 81.

[47] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *Listening*, vol. 10, no. 8, p. 14, 2004.

[48] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors." *29th annual IEEE international conference on local computer networks*, pp. 455–462, 2004.

[49] Contiki-NG, "Contiki-NG," access date: 2021-03-25. [Online]. Available: https://www.contiki-ng.org/

[50] T. Stathopoulos, J. Heidemann, and D. Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks:," Defense Technical Information Center, Fort Belvoir, VA, Tech. Rep., Nov. 2003, access date: 2019-09-04.

[51] "RIOT OS," 2022, access date: 2022-08-23. [Online]. Available: https://www.riot-os.org/

[52] ARM, "ARM Mbed homepage," 2022, access date: 2022-08-23. [Online]. Available: https://os.mbed.com/

[53] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware Over-the-air Programming Techniques for IoT Networks - A Survey," *ACM Computing Surveys*, vol. 54, no. 9, pp. 1–36, Dec. 2022.

[54] M. M. Villegas, C. Orellana, and 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 - volume 2.* Paris, France: ACM Press, 2019, pp. 269–272.

[55] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check," *IEEE Access*, vol. 7, pp. 71 907–71 920, 2019, conference Name: IEEE Access.

[56] Open Mobile Alliance, "Open Mobile Alliance - LightweightM2M Overview," access date: 2021-03-25. [Online]. Available: http://www.openmobilealliance.org/wp/Overviews/lightweightm2m_overview.html

[57] 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).* Halifax, NS, Canada: IEEE, Jul. 2018, pp. 112–117.

[58] M. Khurram, H. Kumar, A. Chandak, V. Sarwade, N. Arora, and T. Quach, "Enhancing connected car adoption: Security and over the air update framework," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 194–198.

[59] D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in *Communications Workshops, 2008. ICC Workshops' 08. IEEE International Conference on*, 2008, pp. 380–384.

[60] M. Hessar, A. Najafi, V. Iyer, and S. Gollakota, "TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds," *17th USENIX Symposium on Networked Systems Design and Implementation*, p. 17, 2020.

[61] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the internet of things: A standardization perspective," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265–275, 2014.

[62] B.-C. Choi, S.-H. Lee, J.-C. Na, and J.-H. Lee, "Secure firmware validation and update for consumer devices in home networking," *IEEE Transactions on Consumer Electronics*, vol. 62, no. 1, pp. 39–44, 2016.

[63] A. K. Srivastava, K. CS, D. Lilaramani, R. R, and K. Sree, "An open-source SWUpdate and Hawkbit framework for OTA Updates of RISC-V based resource constrained devices," in *2021 2nd International Conference on Communication, Computing and Industry 4.0 (C2I4)*, Dec. 2021, pp. 1–6.

[64] S. Schmidt, M. Tausig, M. Koschuch, M. Hudler, G. Simhandl, P. Puddu, and Z. Stojkovic, "How Little is Enough? Implementation and Evaluation of a Lightweight Secure Firmware Update Process for the Internet of Things:," in *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*. Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 63–72.

[65] D. K. Nilsson, U. E. Larson, and E. Jonsson, "Creating a Secure Infrastructure for Wireless Diagnostics and Software Updates in Vehicles," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, M. D. Harrison and M.-A. Sujan, Eds. Berlin, Heidelberg: Springer, 2008, pp. 207–220.

[66] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2290–2300, Nov. 2018.

[67] H. Chandra, E. Anggadjaja, P. S. Wijaya, and E. Gunawan, "Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development," in *Communications (APCC), 2016 22nd Asia-Pacific Conference on*, 2016, pp. 115–118.

[68] The Linux Foundation, "The Update Framework," access date: 2022-05-12. [Online]. Available: https://theupdateframework.io/

[69] Eclipse Foundation, "Eclipse hawkBit," access date: 2022-05-12. [Online]. Available: https://www.eclipse.org/hawkbit/

[70] M. M. Villegas and H. Astudillo, "OTA Updates Mechanisms: A Taxonomy and Techniques Catalog," *XXI Simposio Argentino de Ingeniería de Software (ASSE 2020) - JAIIO 49 (Modalidad virtual)*, pp. 139–158, 2020.

[71] B. Moran, H. Birkholz, and H. Tschofenig, “An Information Model for Firmware Updates in IoT Devices,” RFC Editor, Request for comment 9124, Jan. 2022, access date: 2020-04-03. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9124.html