SAPIENZA
UNIVERSITÀ DI ROMA

Design, Implementation and Evaluation of a Gateway-Device Coordination Protocol to enable Edge Computing over LoRaWAN

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Ivan Fardin

ID number 1747864

Thesis Advisor

Prof. Ioannis Chatzigiannakis

Co-Advisor

Prof. Francesca Cuomo

Academic Year 2020/2021

Thesis defended on 21 March 2022
in front of a Board of Examiners composed by:

Prof. Maurizio Lenzerini (chairman)
Prof. Roberto Beraldi
Prof. Roberto Capobianco
Prof. Ioannis Chatzigiannakis
Prof. Marco Console
Prof. Giorgio Grisetti
Prof. Daniele Nardi

Design, Implementation and Evaluation of a Gateway-Device Coordination Protocol to enable Edge Computing over LoRaWAN
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: 14 March 2022

Author's email: fardin.1747864@studenti.uniroma1.it

Abstract

The number of Internet of Things (IoT) devices is increasing year over year (about 30 billion devices were estimated by 2020 and about 70 billion devices by 2025) and consequently, the data traffic is generated and consumed at the edge of the network infrastructure. As this amount of data grows, current cloud-based approaches are becoming increasingly inefficient as massive datasets collected by sensors are moved from the edge to distant machines of the cloud platform. These large-scale deployments result in a latency that does not fit with time-critical IoT applications requirements, therefore in recent years innovative edge computing and fog computing solutions arose that are able to leverage resources along the path between the end devices and the cloud. At the same time, stream processing frameworks are become an essential component of an IoT system due to their capability to process huge amounts of data in near real-time both in well-established cloud computing architectures and in more recent edge and fog computing ones. In this master thesis, I propose an edge-based framework over LoRaWAN for processing data close to the source where it is generated. The location-aware functionality is realized through a Gateway-Device Coordination Protocol able to associate an end device to the best fit available LoRa gateway. This reduces the data traffic both at the edge and between the edge itself and the core of the network for storing ready-to-use historical data. Furthermore, the framework enhances Quality of Service (QoS) by allowing users to submit stream processing tasks to define how specific data streams have to be elaborated and accessing results directly at the edge without the need for a cloud data transfer minimizing latencies.

The performance of the coordination protocol is evaluated in terms of network utilization, latency and end devices scalability through gateway resources utilization and associations load balancing. The evaluation has been conducted by implementing the Gateway-Device Coordination Protocol in OMNeT++ simulator and revealed that the protocol can be applied with success to a real deployed LoRaWAN network since an effective load balancing of associations is achieved, minimal network traffic at the edge is generated (a mean of 13 LoRaWAN messages per end device are needed to finish the protocol and to start data sending, including in the count the Join Request message and 3 retransmissions for early-stage messages to deliver them with a high probability), a corresponding minimal number of frame losses and collisions is recorded (also due to the small-sized network deployments analyzed) and minimal additional storage is occupied on gateways (for storage space of 8GB the increment results in about $2.5 \cdot 10^{-6}\%$ per associated end device).

Acknowledgments

I would first like to thank my thesis advisor Prof. Ioannis Chatzigiannakis and co-advisor Prof. Francesca Cuomo who provided me with useful papers and suggestions to guide my research towards the right direction but also for their availability to answer questions and doubts that arose during the thesis development. Eventually, for giving me the possibility to experience for the first time academic research to caress what it means to be involved in such a process.

I would also like to thank Stefano Milani for his valuable comments and suggestions about my writing.

A special thank you to my parents who supported me throughout all my study years and in particular during this research, their contribution was extremely important for me, and to my family that always encouraged me with incredible passion every day.

I cannot miss thanking my friends that help me to face every moment with cheerfulness, distracting me from the daily routine and lightening my studies to prepare for exams.

Finally, a thank to all my colleagues with whom I participated in the course projects which represented very instructive and pleasant experiences, but also to those with whom I followed lessons and shared knowledge, despite the well-known global difficulties of the last two years.

Author

Ivan Fardin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) has become a pervasive technology with an increasing number of devices connected to the Internet year over year (about 30 billion devices at the time of writing [48, 78]). These include connected cars, machines, meters, sensors, smartphones and wearable devices.

IoT applications involve innovation in multiple areas such as home [89] and building automation [47], healthcare or Internet of Medical Things (IoMT) [43], Industrial Internet of Things (IIoT) [57], smart grids [84], smart metering [38], transportation [36] and vehicle communications or Internet of Vehicles (IoV) [97]; most belonging to the general concept of a smart city.

This results in the generation and consumption of unprecedented amounts of data at the edge of the network infrastructure [4] in contrast with current developing approaches typically based on cloud services located at the core of the network.

As the number of IoT devices (and the corresponding amount of data they generate and consume) grows, cloud-based approaches are becoming increasingly inefficient because data needs to travel the network infrastructure from the edge to the core (where is processed) and backward. Furthermore, the latencies associated with such data transfer may not be able to support time-critical (interactive or nearly real-time) applications [79].

The advent of 5G [1] and its massive deployment in the near future (in Italy the 700 MHz frequency band currently used by the transmission of digital terrestrial television channels will be free by the end of June 2022 [81, 80]) will represent a first turning point in the mobile and IoT ecosystems, indeed, the combination of high-speed connectivity, very low latency, and ubiquitous coverage will constitute a foundation for realizing the full potential of IoT [73, 87].

In general, IoT devices are battery-powered constrained devices [40] (which requires a careful power usage profile to extend their battery lifetimes) deployed on a large area (from hundreds of meters to several kilometers) whose extension depends on the application and requires a long-range communication at low cost and low complexity.
Common wireless networking technologies (IEEE 802.11 Wi-Fi and IEEE 802.15.1 Bluetooth) focus on enhancing the data rate rather than power consumption and more recent protocols (IEEE 802.15.11 Bluetooth Low Energy and IEEE 802.15.4 ZigBee) were proposed focusing on low-power wireless transmissions. However, all of them are designed to provide short-range communication in Wireless Local Area Networks (WLANs) and Wireless Personal Area Networks (WPANs).
On the other side, Wireless Cellular Networks (2G, 3G and 4G) proposed so far, although cover very large areas, do not meet device power consumption and performance requirements because were designed for voice and data communication and not for wireless sensor applications.
In last recent years, Low-Power Wide Area Networks (LPWANs) technologies (SigFox, LoRaWAN and NB-IoT) [58] were developed offering low power, long-range and low-cost transmission and therefore gaining increasing popularity in IoT industrial and research communities. Nowadays, a large IoT deployment may be only realized using a LPWAN technology [14].

Stream processing frameworks are revealed to be very effective at processing large amounts of data in near real-time [18], especially when deployed on cloud platforms providing seamless elasticity and scalability capabilities.
Conversely, this is challenging for IoT time-critical applications, so, in last recent years, several frameworks have been proposed to exploit existing and available resources at the edge. The emergence of edge computing and fog computing started an innovation process to efficiently develop large-scale IoT applications combining the aforementioned resources with the ones available in the fog and the cloud [77, 93].

Following this principle, starting from a real-world smart water metering use case, I designed an edge-based architecture based on LoRaWAN to generate, process and consume data near to the source without transferring it to the cloud but processed historical data ready-to-use in further analysis. This can include the development and deployment of complex machine learning models not suitable for the edge and that require massive datasets to be trained and/or Big Data or data mining techniques. In developing the system, I focused on the location of data producers to process their streams as close as possible to minimize data traffic size and corresponding

networking delays. At the same time, data consumers by accessing processed data directly on the edge, maximize the Quality of Service (QoS) because there is no latency due to a data transfer between the edge and the distant cloud machines. The key idea is to leverage the geographically distributed LoRaWAN gateway resources to enable edge computing over LoRaWAN. Instead of simply relaying packets as expected by the specification, gateways move from a passive role to an active one where they process, employing a stream processing engine, end device measurements before forwarding them. This involves many adaptations to the LoRaWAN protocol from security and keys, to lost messages and frame counters and is realized through the definition of a fully automatic Gateway-Device Coordination Protocol able to associate each end device of a network to the best fit gateway. End users' data access poses issues about authentication and privacy that are carefully considered in the client connection to a gateway. As multiple sensors may be accessed by multiple users based on users' privileges, a scale-out and corresponding scale-in algorithms are necessary to offload the computation on another gateway to avoid overloading a peer.

The Gateway-Device Coordination Protocol has been implemented using the well-known network simulator OMNeT++ and evaluated in terms of correctness, network utilization, latency and end devices scalability through gateway resources utilization and associations load balancing. The results collected running a set of experiments revealed that the protocol can efficiently coordinate end devices and gateways via a small number of messages (a mean of 13 messages per end device including OTAA and 3 retransmissions for specific message types to maximize frame delivery ratio) and performing associations as expected based on endpoints locations and the amount of work the gateways are carry on to avoid overloading them. Besides the small size of the network deployments (6 and 13 end devices and 2 gateways), two random wake-ups reduce the number of message collisions resulting in a 99.28% delivery ratio. Furthermore, the impact of the protocol on gateway storage of 8GB results in an imperceptible increment of occupied storage of about $2.5 \cdot 10^{-6}$% per associated end device suggesting large-scale networks bottleneck will not be represented by the storage size but the CPU, GPU and RAM availability according to the processing tasks the gateway has to perform.

Chapter 2

Related Works

In the literature, several frameworks have been proposed in recent years that leverage resources at the edge and at the fog of the network infrastructure to cope with emerging aforementioned issues.

Amaxilatis et al. [6, 5] presented a system based on the fog computing paradigm to facilitate greatly the analysis of fine-grained water consumption data collected by the smart meters. The authors introduced an intermediate layer of gateways connected to end devices via a wM-Bus interface and LoRa gateways via a LoRa interface. Data collected by sensors are processed by the two layers of gateways where the introduced layer can perform only basic manipulations due to its very constrained resources. While edge nodes perform a generic data processing based on a predefined set of functions for data engineering goals (extract useful information from end device data, clean them, impute missing values, ...), the backend realizes application-specific processing by offering to users the possibility of specifying tasks via lambda expressions.

Renart et al. [68] designed an edge-based programming framework that allows users to define how data streams are processed based on the content and the location of the data. Authors created an abstraction using a publish/subscribe overlay network that can discover available computational resources and allocate the computation in the most appropriated one at runtime (sensors produce data, users produce and consume data processed by the overlay network of rendezvous points). The system is implemented through the JXSA P2P protocol, so, all the interactions between sensors, users and rendezvous points are carried on through this application layer technology.

Das et al. [20] introduced Seagull, a framework for building large-scale IoT applications where sensors' data are distributed on edge nodes based on their proximity to the source and the amount of processing they can handle. This is realized as an extension of the Cowbird cloud-based framework [21] using the domain-

specific language SWAN-Song [63]. When a user registers a SWAN-Song expression, the Seagull manager deployed at the cloud assigns it to the least-busy Seagull Node (active threads) in the network and the closest available Seagull Node to the IoT device (euclidean distance between the estimated locations of the IP addresses).

Fu et al. [30] presented EDGEWISE, a new Edge-friendly Stream Processing Engine (SPE) that redesign the SPE runtime by incorporating a congestion-aware scheduler and a fixed-size worker pool. The scheduler separates the threads (execution) from the operations (data) so that a ready thread can be assigned to the operation with the most pending data. The fixed number of workers moves from operation to operation as assigned by the scheduler (rather than dedicating a worker to each operation) to reduce unnecessary overhead on constrained devices.

Corral-Plaza et al. [18] proposed a three-layer architecture for processing and analyzing data from heterogeneous sources with different structures in IoT scopes, allowing researchers to focus on data analysis, without having to worry about the structure of the data sources. The middle layer (between data producers and data consumers) combines a stream processing engine, that is in charge of transforming and processing data from the sources to make data ready for analysis, and a complex event processing which receives the processed data as input and has the role of detecting situations of interest by evaluating data against predefined patterns and notifying users of such events.

Zeuch et al. [96] introduced the NebulaStream (NES) platform, an end-to-end data management platform that enables future IoT applications by unifying sensors, fog and cloud and by addressing heterogeneity, unreliability and scalability challenges for state-of-the-art data management systems. NES copes with heterogeneity by maximizing sharing of results and efficiency of computing to significantly reduce the amount of data transferred and to exploit hardware capabilities efficiently. NES addresses unreliability by applying dynamic decisions and incremental optimizations during runtime to be as flexible as possible. NES enables elasticity by designing each node to react autonomously to a wide range of situations during runtime. NES has a centralized three-layers architecture where users send queries to the NES Coordinator deployed on the cloud and each sensor belonging to the sensor layer is connected to at least one low-end node in the Fog Layer, which is responsible for this sensor (entry node). In the fog layer, NES processes data as they flow from Entry Nodes to Exit Nodes where the data transfer is orchestrated by routing nodes. After leaving the Fog Layer through an Exit Node, data enter the Cloud Layer that can apply remaining processing and output the data to the user.

Zeuch et al. [95] analyzed the requirements of upcoming IoT applications and the supported features of an IoT data management system to outline state-of-the-

art challenges and limitations and highlight how to efficiently use them in IoT infrastructures and how NebulaStream addresses them.

Gavriilidis et al. [34] presented a potential large-scale application implemented on top of NebulaStream to showcase how NES addresses the shortcomings of current cloud-based stream processing engines and thus enables future large-scale IoT applications.

Tönjes et al. [85] proposed CityPulse, a smart city framework for processing large-scale IoT data streams by enriching data streams with semantic annotations, enabling adaptive processing, aggregation and federation of data.

Xhafa et al. [92] presented an edge-computing system focused on semantic data enrichment and semantic data representation. This is divided into four layers: IoT Data Sensing Layer where the data is generated; Edge Processing Layer where edge nodes preprocess sensors' data and then semantically enrich them; Data Analysis and Reasoning Layer runs a cloud-based semantic engine that infers knowledge from semantic data and ontology-based on predefined rules; User Application Layer deployed on the cloud.

Buddhika et al. [12] designed NEPTUNE, a holistic framework built on top of Granules cloud runtime that addresses the CPU, memory, network, and kernel issues involved in stream processing for IoT environments. NEPTUNE makes effective use of the network bandwidth to achieve high throughput while maintaining the communication latencies at acceptable levels. To accomplish this, it buffers stream packets at the application layer and transfers a batch of buffered messages over the network rather than sending individual messages one at a time. NEPTUNE's buffering schemes are streamlined with batch processing to reduce the number of context switches among worker threads and to improve the use of the instruction cache. Moreover, the framework relieves memory pressure, through a frugal object creation scheme that reduces strain on the garbage collector via reuse of objects and data structures, and supports backpressure, via a flow control mechanism, to cope with discrepancies between processing rates and data arrival rates at certain stages of a stream processing job. In the end, NEPTUNE also incorporates a dynamic compression scheme that selectively compresses portions of a data stream based on their entropy levels.

Sittón-Candanedo et al. [77] proposed the use of GECA (Global Edge Computing Architecture), an Edge-IoT platform and a Social Computing framework to build a system aimed at smart energy efficiency in a public building scenario. Authors improve CAFCLA [32, 31] by combining it with the GECA architecture which integrates blockchain technologies to strengthen security on the three levels of its architecture. The IoT layer is composed of sensors and actuators connected over

Wi-Fi or ZigBee, the edge layer filters and preprocesses the data generated in the IoT layer in real-time, sending to the cloud the data used by Business Intelligence applications and the business solution layer is composed of a set of cloud-based services and business applications.

Amarasinghe et al. [4] devised an optimization framework that aims to minimize end-to-end latency through the appropriate placement of Data Stream Processing operators either on cloud nodes or edge devices. The authors represented the distributed stream processing system which consists of both cloud and edge resources as a connected graph. Then, they formulate a constraint satisfaction problem to realize an optimal task allocation aiming to minimize the end-to-end latency.

Cheng et al. [15] designed and implemented Geelytics, a system which can enable on-demand edge analytics over scoped data sources. It is a stream processing system that is tailored to IoT systems to enable on-demand edge analytics concerning edge computing. The entire system consists of three major components: a controller running on a controller node which is in charge of monitoring the status of the entire system, presenting them to application developers, and is the entry point of the system so that has to authenticate all data producers and consumers and also assigns a nearby worker to them when they join the system, based on the location proximity calculated from their GPS attributes; a large number of workers running on geo-distributed compute nodes either in the Cloud or at the network edge; a set of topology masters, that can run on the controller node together with the controller or separately on their dedicated master nodes, that has the role of managing task instances, including generating, configuring and assigning them to currently running workers aiming to reduce the workload of the controller. Geelytics provides friendly interfaces, for both data producers and consumers, to interact with the system based on the MQTT publish/subscribe paradigm.

Teranishi et al. [83] proposed a peer-to-peer-based dynamic data flow platform that replicates processes and changes the structure of the data flow dynamically on the distributed computational resources located at network edges and data centers. Authors extended topic-based pub/sub (TBPS) messaging by adding the notion of "index" to build a key-ordered overlay network for flexible allocations of the data stream processes on edge nodes and cloud nodes. To cope with peaks and troughs in the data stream, they implemented a mechanism for adding one or more computational nodes (scale-out) or removing the nodes dynamically (scale-in). Furthermore, the authors designed a peer-to-peer-based data stream routing algorithm called Locality-Aware Stream Routing (LASR) for dynamic data flow management such that by sorting entities in order of the network id, entities in the same edge network are located adjacent on the key-order preserving structured

overlay.

Wang et al. [90] introduced algorithms for solving the online application placement problem in the context of Mobile Edge-Clouds (MECs) by modeling such environments as hierarchical graphs to optimize the load balancing of the applications.

Lee et al. [54] presented iEdge, an IoT-assisted edge computing framework that enables the seamless execution of applications across an edge server and nearby IoT devices. It decomposes the source code of an application developed for cloud/edge servers and reconstructs a logically composite application by porting some of the function-level modules so that they can be independently executed on IoT devices. The resultant composite application is then executed across an edge server and IoT devices considering device context. A key element is generating machine-independent code for each offloadable function by identifying relationships between functions in an application program in terms of function calls.

Chapter 3

Background

3.1 LoRaWAN

LoRa physical layer is a wireless modulation for long-range (up to five kilometers in urban areas and up to 15 kilometers or more in rural areas [72]), low-power (up to 10 years battery-operated devices lifetime) and low-data-rate applications developed by Semtech and patented in 2014 [50, 9]. Its spread spectrum modulation technique is derived from chirp spread spectrum (CSS) technology and produces a chirp signal where all chirps will have practically the same time duration [40]. The chirp length of a transmitted digital symbol determines the spreading factor (SF) of a LoRa communication. LoRa modulation has a total of six orthogonal spreading factors (SF7 to SF12) and the larger the spreading factor used, the farther the signal will be able to travel and still be received without errors by the radio receiver [72]. The orthogonality property of SFs makes it possible to simultaneously transmit on the same frequency channel without that two signals modulated with different SFs interfering because they appear to be noise to each other. Furthermore, LoRa signals are robust and very resistant to both in-band and out-of-band interference mechanisms and LoRa modulation also offers immunity to multipath and fading, making it ideal for use in urban and suburban environments. It operates in the license-free sub-GHz bands such as the 433-, 868- or 915-MHz frequency bands but in Europe, only the first two can be used and due to transmission regulations, each transmission in any of the 868 MHz and 867 MHz sub-bands should comply with a 1% radio duty cycle or implement a listen-before-talk or adaptive frequency agility mechanism [49].

The LoRa frame structure includes a preamble, an optional header and the data payload that contains either LoRaWAN MAC layer control packets or data packets.

LoRaWAN is a Low Power Wide Area Networking (LPWAN) protocol that is

optimized for battery-powered end-devices and is designed for Internet of Things (IoT) application requirements such as bi-directional communication, end-to-end security, mobility and localization services. It's an open-source medium access control (MAC) protocol standardized by the LoRa Alliance that runs on top of the LoRa physical layer and enables the communication between LoRaWAN end-devices and gateways: only a few gateways, configured in a star network topology, are required to serve a multitude of end nodes so that the deployment cost is relatively low. Thanks to these characteristics and the Industrial, Scientific and Medical (ISM) bands [52] in which it operates that allows to deploy LoRaWAN networks without the involvement of mobile operators, LoRaWAN technology is gaining increasing popularity and is becoming one of the most valuable and used LPWAN technology [22].

LoRaWAN packets structure includes a MAC header, the data payload which contains the upper layer application frame and the Message Integrity Code (MIC).

3.1.1 LoRaWAN Architecture

LoRaWAN architecture is typically laid out in a star-of-stars topology in which gateways relay transmissions between end-devices and a central network server located at the backend (but there exist deployments in which the network server is located at the edge too) [23]. Gateways are connected to the network server via standard IP connections, whereas end-devices use single-hop radio-frequency communication to one or many gateways according to how many gateways are available in their radio ranges. All communication is generally bi-directional, although uplink transmission from an end-device to the network server is expected to be the predominant traffic. Communication between end-devices and gateways is distributed over different frequency channels and data rates and selecting the data rate is a trade-off between communication range and transmission duration. To maximize the capacity of the network given a fixed number of gateways, using an adaptive data rate (ADR) mechanism is essential because, although its main goal is to save battery power of end devices by having the end-nodes closest to a gateway transmit using the lowest spreading factor, their time on air is minimized, thereby prolonging their battery life and minimizing possible collisions. Beyond these three main components, other two entities participate in the LoRaWAN architecture, so summarizing all of them:

- **End devices:** are sensors and/or actuators that communicate with the gateways in their coverage areas by broadcasting messages over LoRa. Sensors send measurements to gateways via uplink frames while actuators are expected to only receive commands for reactive behaviors from the network server. The

manufacturer assigns to each produced end-device a 64-bit Extended Unique Identifier (DevEUI) that globally identifies it.

- **Gateways:** are indoor or outdoor devices that act as a transparent bridge between the end devices and the network server since relay messages from one interface to the other after the opportune conversion from LoRa frames to IP packets and vice versa. The IP traffic from a gateway to the network server can be backhauled via Wi-Fi, hardwired Ethernet or via a Cellular connection and according to the vendor a gateway may be provided with a GPS sensor, a variable number of radio channels and different hardware resources (CPU, GPU, RAM and ROM)

- **Network Server:** may be deployed at the cloud or the edge and has the role of managing the entire network (from end devices up to users). As multiple gateways can receive and forward to the network server the same LoRa frame from an end device, an essential feature of the network server consists in detecting duplicates and deleting them to forward uplink frames exactly once to the Application Server. Furthermore, it ensures the authenticity of every sensor on the network and the integrity of every message and is responsible for downlink message routing

- **Application Servers:** are deployed on the cloud and are responsible for securely handling, managing and interpreting sensor application-specific data. Moreover, they generate all the application-layer downlink payloads and send them to the end devices through the network server

- **Join Server:** is deployed on the cloud and manages the over-the-air activation process for end devices to be added to the network.

3.1.2 LoRaWAN Message Definition

LoRaWAN defines uplink and downlink messages, denoting the message transmission direction [50]. Uplink messages are broadcasted by end devices while downlink messages are sent by the network server. Nonetheless, packets structures are almost identical and slightly differ in the bit flags of the frame control field at the application layer (figure 3.2).

In addition to this first distinction, LoRaWAN messages include other frame types for initial end device set up and to require confirmation on uplink and downlink packets. This is achieved through the FType bit field of the data link header where the last 3 bits are used to distinguish among 6 frame types:

Figure 3.1. LoRaWAN architecture

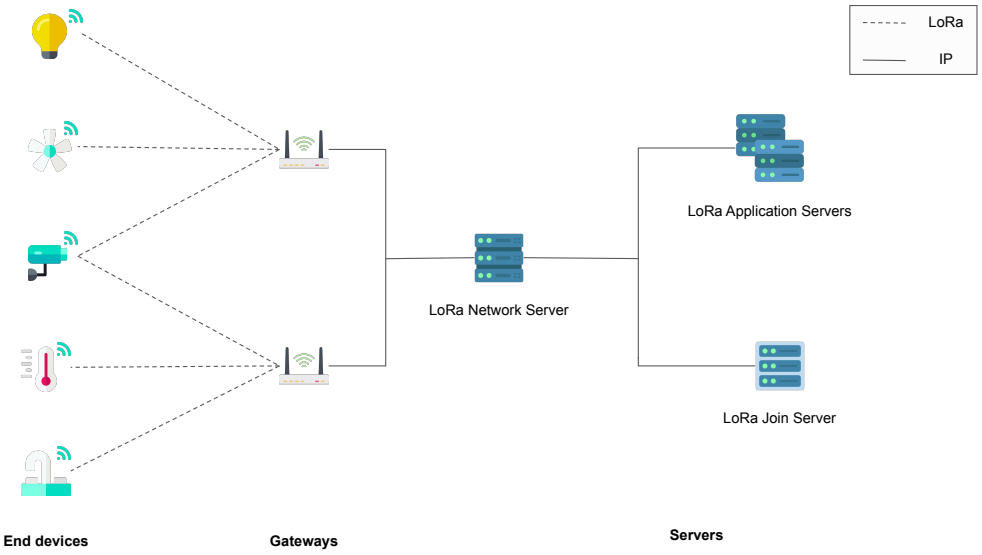


Figure 3.2. LoRaWAN frame (LoRa modulation)

Application layer

Bits	32	8	16	0-120	8	n
Field	Device address	Control	Counter	Options	Port	Payload

Data link layer

Bits	8	m	32
Field	Header	Payload	MIC

Physical layer (LoRa modulation)

Size	8 symbols	4.25 symbols	8 symbols		p bits	16 bits
Field	Preamble	Sync Word	Header	Header CRC	Payload	Payload CRC

- 000 → Join-Request
- 001 → Join-Accept
- 010 → unconfirmed data uplink
- 011 → unconfirmed data downlink
- 100 → confirmed data uplink

Figure 3.3. LoRaWAN frame (FSK modulation)

Application layer

Bits	32	8	16	0-120	8	n
Field	Device address	Control	Counter	Options	Port	Payload

Data link layer

Bits	8	m	32
Field	Header	Payload	MIC

Physical layer (FSK modulation)

Bits	40	24	8	p	16
Field	Preamble	Sync Word	Payload length	Payload	Payload CRC

- 101 → confirmed data downlink
- 110 → RFU
- 111 → Proprietary

The confirmation is expressed through the ACK flag of the Frame Control in the Application layer header (a bit that if set denotes the acknowledgment of the received frame).

The device address field of the application layer of the LoRaWAN frame always refers to the end device both in uplink and downlink transmissions, so, it is not used as the destination address.

The port field of the LoRaWAN frame denotes the type of the message and since port 0 is reserved for MAC messages, port 224 is reserved for MAC compliance testing and ports in the range 225-255 are reserved for future standardized application extensions, valid port numbers are between 1 and 223.

In figure 3.2, the LoRaWAN frame payload sizes are not reported with the exact number of bits because they are region- and data-rate-specific (min 11, max 242 bytes but may be smaller if the options field is not empty) while the physical layer frame format depends on the modulation used: LoRa™, LR-FHSS or FSK [50, 49]. All formats are similar and the changes mainly concern the size of the preamble and the header (figure 3.2 reports the LoRa modulation while the FSK modulation is presented in figure 3.3).

LoRaWAN defines a mechanism to deal with retransmissions through the Frame Counter field of the application layer header which contains the least significant 16 bits of the corresponding 32 bits counter. Every end device manages two counters called FCntUp and FCntDown, one per communication direction, that are initialized to zero when an end device joins a LoRaWAN network [50]:

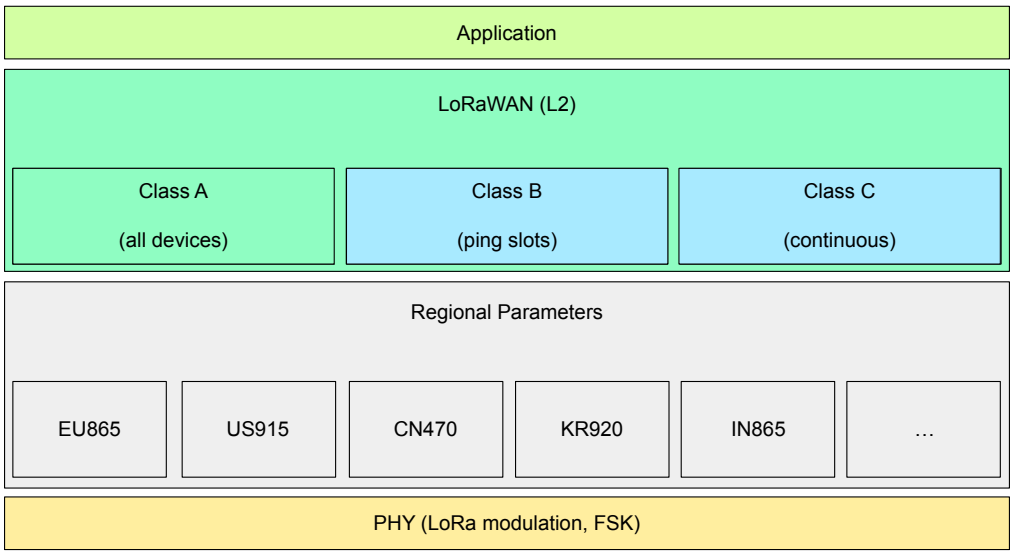
- FCntUp is incremented by an end device when a data frame is transmitted to the LoRaWAN network server (uplink),
- FCntDown is incremented by the LoRa network server when a data frame is transmitted to an end device (downlink)

The network server has to keep track of the uplink counter sent by a specific end device, in order to insert in the downlink frame the proper counter. In this way, duplicates can be easily detected and ignored by inspecting the packet header. The NbTrans parameter relaxes this constraint for uplink retransmissions having the same uplink counter.

3.1.3 LoRaWAN Device Classes

The LoRaWAN specification allows end devices to always send uplink frames at will but, on the other hand, defines three classes of end devices that determine when they can receive downlink frames, affecting the corresponding energy efficiency of the device: Class A, Class B and Class C. Since the last two classes are extensions to class A specification, all end devices must implement class A.

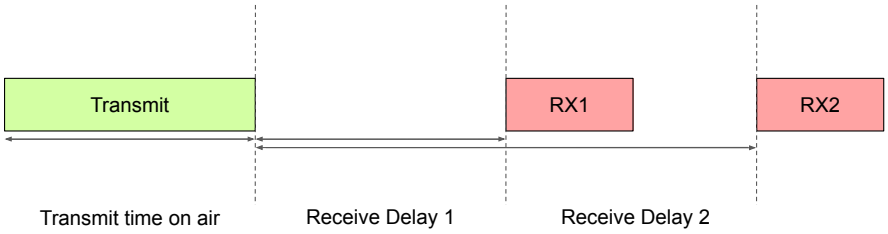
Figure 3.4. LoRaWAN classes



Class A

Class A (Aloha) devices spend most of their time in sleep mode and wake up when have to send an uplink message (e.g. due to a timeout or sensor reading). After that, they open two consecutive short receive windows to listen for a downlink message.
This means that such devices can receive frames only if they previously send a message and are not suitable for receiving multiple messages in response to a single uplink frame. Indeed, a message can only be received in one of the two receive windows because if a frame is detected and demodulated during the first receive window and the device address matches and Message Integrity Code (MIC) is valid, then the second receive window is not even opened. The first and second downlink receive windows respectively start by default 1 and 2 seconds after the end of the uplink transmission but can be changed via a specific MAC command or in the set up of the join procedure. Class A devices result in the lowest battery-power consumption at cost of a high downlink latency.

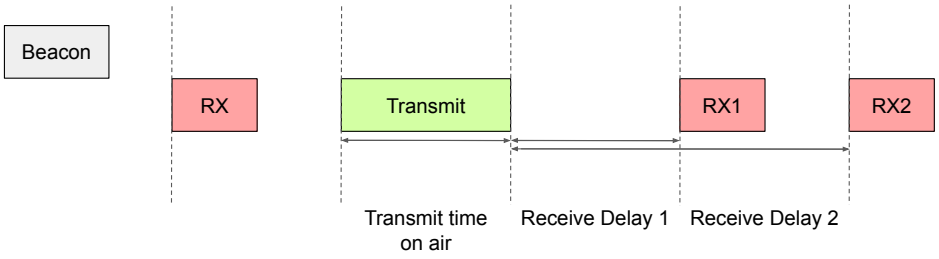
Figure 3.5. Class A receive windows



Class B

Class B (Beacon) devices extend Class A by opening additional receive windows at a scheduled time. Gateways send periodic beacons to the end devices to synchronize their clocks and make devices open receive windows periodically. Class B devices reduce the downlink latency of Class A devices at the cost of higher battery-power consumption.

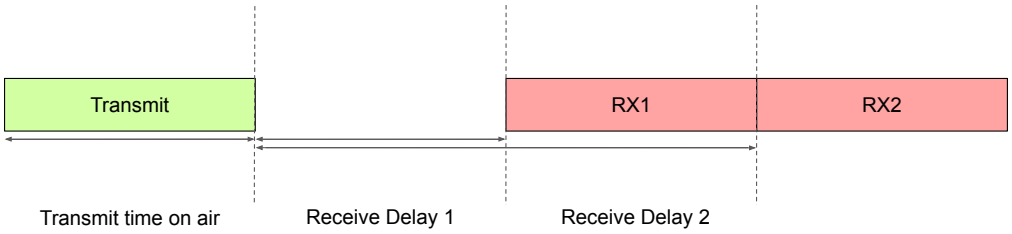
Figure 3.6. Class B receive windows



Class C

Class C (Continuous) devices extend Class A by keeping the receive windows open unless they are transmitting, so never go in sleep mode. So, such devices result in the highest battery-power consumption but guarantee no downlink latency.

Figure 3.7. Class C receive windows



3.1.4 LoRaWAN Device Activation Methods

Before sending and receiving messages on LoRaWAN, every deployed end device must be registered within the network. This takes place through a procedure called activation method and LoRaWAN specification allows for two types of activations:

- **Over-The-Air Activation (OTAA):** is the most secure and recommended activation method for end devices because these perform a join procedure with the network to be assigned a dynamic device address and to negotiate security keys
- **Activation By Personalization (ABP):** is the least secure as it requires hardcoding the device address and the security keys in the end devices. These static assignments have the additional downside that for switching network providers a manual intervention to change end device keys is needed.

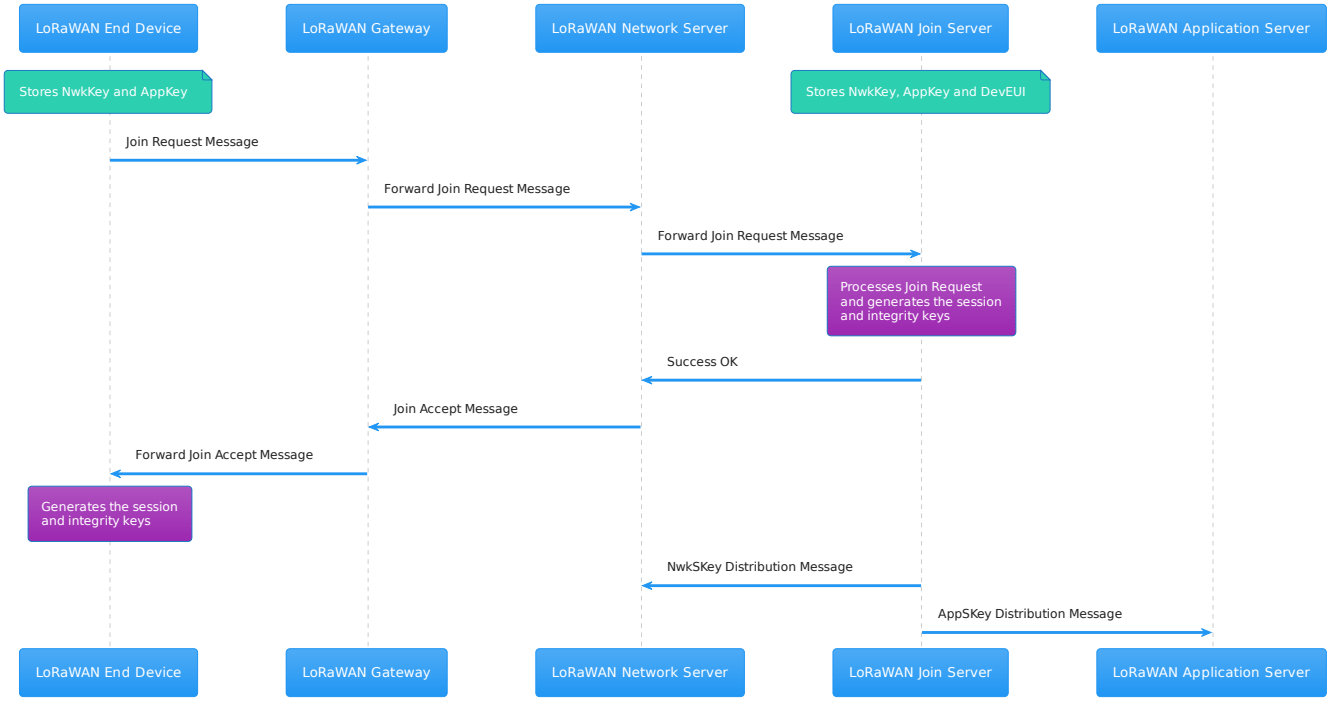
Going deeper in OTAA as described in LoRaWAN 1.1.x specification, the join procedure establishes mutual authentication between an end device and the LoRaWAN network to which it is connected and only authorized devices are allowed to join the network. Indeed, in LoRaWAN the payload of every message is encrypted with a unique 128-bit session key shared between the two parties involved in the communication (either the end device and network server or the end device and the application server) and AES algorithms are used to provide confidentiality, data integrity and authentication of packets. Hence, LoRaWAN through the CIA triad ensures that network traffic cannot be altered, eavesdropped on, captured and replayed and that only legitimate devices are connected to the LoRaWAN network. To derive the two session keys (NwkSEncKey and AppSEncKey), two root keys per end device are securely stored both on the end device and on the join server (together with the corresponding device EUI) and employed in the join procedure that consists in:

1. The end device sends a join request message to the join server specifying the Join EUI, the device EUI and a 16 bits nonce generated on-the-fly,
2. The join server authenticates it and replies with a join accept message including a 24 bits nonce generated on-the-fly, the 24 bits network ID, a 32-bit device address (which uniquely identifies such a device within the network it belongs) and other parameters about the network
3. The end device and the join server derive application and network session keys locally, based on root keys and fields in the join request and join accept messages,

4. The join server shares the two session keys with network and application servers

The idea of having two session keys (one for the LoRa network server and one for the application server) is to secure connections between the end device and the network server and between the end device and the application server such that neither the network server can read messages encrypted with the AppSEncKey. This means that in any case, the LoRa gateways cannot read the data traffic they relay because are not expected to be provided with the session keys since all of the infrastructure-side cryptography happens in the LoRa network server or the application server.

Figure 3.8. LoRaWAN OTAA method message exchange



In details, the session keys are derived from root keys and parameters in Join request frame and Join accept frame as follows

$$NwkSKey = aes128\_encrypt(AppKey, 0{\times}01 \,|\, JoinNonce \,|\, NetID \,|\, DevNonce \,|\, pad_{16})$$

$$AppSKey = aes128\_encrypt(AppKey, 0{\times}02 \,|\, JoinNonce \,|\, NetID \,|\, DevNonce \,|\, pad_{16})$$

3.2 Edge Computing

The increasing number of connected IoT devices results in a massive amount of data produced at the edge of the network infrastructure. Moving this on the cloud for performing data processing, pushes network bandwidth requirements to the limit and introduces bottlenecks that could congest the network. Despite improvements of the network infrastructure, distant cloud machines cannot guarantee acceptable latency for IoT time-critical applications [79]. For this reason, in recent years research focuses on moving the computation from the cloud (Cloud Computing) to the edge (Fog Computing and Edge Computing) leveraging on resources deployed close to data producers, although less performing than the cloud machines [77]. This is expected to mainly reduce latency and save network bandwidth without neglecting privacy and security of data [3].

3.3 Stream Processing

IoT sensors produce huge volumes of data and time-critical applications need to process them in near real-time to perform aggregations (e.g., calculations such as sum, mean, standard deviation), analytics (e.g., predicting a future event based on patterns in the data), transformations (e.g., converting a data format into another), enrichment (e.g., combining the data point with other data sources to create more context and meaning) and ingestion (e.g., inserting the data into a database) [51]. As the sensors continuously send data at regular time intervals or when triggered by specific events, the batch processing paradigm which collects and stores data before processing it all at once in batches based on a schedule or some predefined threshold, is not suitable for real-time processing and solutions deployed at the edge due to their constrained resources [74, 65]. On the other hand, stream processing represents a suitable solution because efficiently performs real-time processing as soon as new data arrives from the source to the processing node.
In recent years, many frameworks have been developed implementing this computer programming paradigm such as the famous Apache Flink [25], Apache Kafka [27], Apache Spark [28] and Apache Storm [29]. Unlike the others mentioned, Spark is not a native streaming engine but a micro-batching engine that means collecting data and processing it together every few seconds, thus introducing small delays [16]. A stream processing engine listens for new task requests and receives in input a query denoting the operations to apply to a specific data stream. Then, it builds a corresponding dataflow graph of operations and executes them on the related data stream.

Typically, stream processing operations are grouped into four categories [64]:

- Single record operations: process a single event in the input
- Multiple records operations: process multiple events in input through windows
- Join operation: merge multiple data streams into one
- Split operation: separate a data stream into multiple ones

Common single record operations are Filter (removing undesirable data) and Map (transforming data) while common multiple records operations are analytics such as Count and Average and apply to data collected in a specified window. A window is a memory to look back at recent data efficiently.
The join operation is a challenging operation for the streaming framework because multiple data streams can probably have different timestamps and therefore the engine needs to align them. This is not surprising since time is a well-known problem that has been studied by the distributed systems community for decades [33].
From the time of events, it is a natural consequence to derive an ordering, thus stream processing operations also include the possibility to detect patterns by correlating events based on timestamps and the "happened-before" relation. This means that anomaly and fraud detections can be easily implemented without developing complex machine learning models [70].
However, if the application needs sophisticated predictions, then machine learning has to be introduced and there exist two main approaches:

- Develop and learn a model on the cloud, based on time series in a classical way and then deploy it to the edge
- Develop and learn a model directly at the edge, based on streaming machine learning [37]

3.4 MQTT

The Message Queuing Telemetry Transport (MQTT) is a standard lightweight messaging protocol designed for the Internet of Things. It is based on the client-server publish/subscribe message pattern and usually works over TCP/IP but can also be supported by any network protocol that provides ordered, lossless, bi-directional connections [8].
The server role is performed by the MQTT message broker that routes all incoming messages from the clients (publishers) to the appropriate destinations (subscribers) through topics. The MQTT client consists of any device (from a microcontroller up

to a smartphone or a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Chapter 4

Framework

IoT large-scale deployments produce huge amount of data and there exist many use cases such as face detection [54], smart public transportation [35], vehicle communications [97], smart industry [13] and smart grids [24] that require IoT processing at the edge to efficiently support these volumes in near real-time. A smart grid is an electricity network that employs smart meters to realize an advanced metering infrastructure involving end-users and aiming to ensure energy-efficient resources with low losses and high levels of quality and safety of supply [17]. In particular, smart water metering represents a milestone towards future smart cities as provides accurate measurements about water flows, can detect and react to anomalies (broken pipes, frauds in bills, ...) and enables the analysis of water demand that helps to design urban water supply networks [11, 5, 6, 38]. While this analysis is carried on using historical data and permanent storage, clearly, it is instead essential to detect anomalies in real-time so that meters and/or actuators can react as quickly as possible to an alert or show real-time consumption to end-users with low latency.

Starting from an existing smart water metering infrastructure over LoRaWAN consisting of

- End Devices: smart water meters deployed along the water supply network,
- Gateways: Unidata routers geographically distributed in the city,
- Network, Join and Application servers: deployed on the cloud

I designed an edge-based framework to address well-known issues of latency and data volumes of a large-scale cloud-based IoT deployment. So, instead of processing sensors' measurements through cloud computing, I enabled edge computing over LoRaWAN through a Gateway-Device protocol that associates every connected end device to at least a gateway.

The idea is to realize a **layer of Rendezvous Points** (RPs) [68, 20] represented by LoRa gateways (Unidata routers) in which data produced by end devices (water metering sensors) are processed according to client requests (through stream processing engines) at the best fit LoRa gateway for every end device. This is selected by the sensor itself based on multiple metrics (such as RSSI, CPU-load, GPU-load, RAM load, storage I/O statistics, network statistics, ...) by running a distributed algorithm over LoRaWAN. So, data streams are dealt out on RPs based on their proximity to the source (i.e. sensor) and on the workload they can handle (i.e. available resources).

To allow users to access them directly at the edge, once an association between a sensor and a gateway is performed, it is stored, together with the data profile of the end device, both on the gateway and on the network server, since the latter represents the entry point for external connections. A data profile consists of a set of attributes that defines the properties of the related sensor. Hence, in a generic heterogeneous deployment, it can report the type of the device, its mobility degree, the quality of its measurements and so on. This means that queries can be submitted at the edge simply specifying attributes that match the data profile of the end device of interest and the network server provides the user with the proper IP address of the gateway to which the end device is connected. Nevertheless, when the user's client retrieves the IP address of the LoRa gateway paired with the desired end device, it may be unable to instantiate a stream processing task on such RP because its available resources could not be sufficient for executing the query. Therefore, a mechanism to scale out data streams securely on other gateways than the selected ones and eventually scale-in is needed to elastically support the demand. Furthermore, to respect the privacy of people, before accessing data collected by an end device, a robust authentication method is required.

The contribution consists of the design, implementation and evaluation of the Gateway-Device Coordination protocol and of the design of the user's connection to a LoRaWAN gateway to submit queries and process data according to its needs.

Since sensors are deployed in LoRaWAN, multiple sensors are in the area of multiple LoRa gateways which relay messages between end-devices and a central LoRaWAN network server that represents the point of connection between the LoRaWAN and the Internet. The gateways have two networking interfaces and communicate with sensors over LoRa and with the LoRa network server and peers over IP (using either TCP or UDP transport protocols).

In this architecture, I can define 3 main roles:

- data producers,

- data processors,
- data consumers

Producers are represented by the sensors which continuously generate data. Processors are represented by the LoRa gateways which don't limit their task to forwarding data but first elaborate and then relay it.
Consumers are the users that through their clients connect to the gateways (over IP) and assign processing tasks on generated data.

Chapter 5

Gateway Activation Method

As described in section 3.1.4, no LoRaWAN activation method provides gateways with session and integrity keys to decrypt/encrypt uplink and downlink traffic they relay because all of the infrastructure-side cryptography happens in the LoRaWAN network server or the application server.

As in the proposed framework gateways need to access the content of end device frames in order to process them, the LoRaWAN standard poses an issue. To deal with it, every LoRa gateway can be provisioned with the network session keys (NwkSKeys) and application session keys (AppSKeys) of the sensors connected to it and generally used by them to respectively communicate with the network server and the application server. In this way, the gateway can decrypt incoming traffic, process it and then re-encrypt and forward on the other connection (like a MITM attack) if necessary. This implies that every network session key must be securely stored on at least a LoRa gateway in addition to the expected end device and LoRa network server and the same applies to the application session key.
Besides a storage problem that can be partially addressed by halving the occupied bits by only storing the NwkSKeys or AppSKeys used to encrypt data, the overall security of the LoRaWAN protocol is weakened because the same session key is stored in multiple actors (at least three) compared to necessary two endpoints involved in the communication, transferring a key is always a risky operation and the approach is not compliant with defense in depth principle according to which a system should use multi-layer protection.
Additionally, this naive approach only works for securing the stream of data sent by the end device after the binding with the LoRa gateway is performed as it makes no sense to provide every gateway with all the keys of the end devices when just a subset of them will be used by a gateway. Moreover, it is also not sufficient to distribute keys to gateways in the radio range of an end device that forward the

initial frame to the network server when the end device joins the network because the sensor can finalize an association with a LoRa gateway of level-1 or higher that would not have got the session keys to decrypt/encrypt messages.

A better solution consists of a sort of strip MITM attack where the gateway uses different keys for the two-side connections about an end device of which it takes part: one with the end device and another with the LoRa network server. Hence, the gateway should negotiate and store a session key for each association with an end device and a single one (or multiple ones) for the connection with the LoRa network server.

Moreover, to enable communication before an association takes place since no message is sent in cleartext over LoRaWAN and to prevent other possible attacks to the framework examined in chapter 7, every gateway must be also provided with two additional key types: one shared among the end device and the LoRa gateways in its radio range and another shared among the gateway and the peers in its radio range belonging to the same cluster.
This means that every gateway needs at least two session keys to run the Gateway-Device Coordination Protocol, which become at least four if it participates in at least a cluster and covers at least an end device which results in at least $4K$ available bits where K is the key length in bits (assuming constant key lengths and encryption, authentication and integrity algorithms). Then, an additional session key is stored for each end device that associates with it, and therefore, the first binding occupies at least $5K$ bits that using 128 bits session keys and AES algorithms adopted by LoRaWAN results in 640 bits. On the other hand, each subsequent pairing requires only 128 bits. In the end, the number of bits required by session keys on a Gateway is

$$2K + c\,K + n\,K + m\,K = K\,(c + n + m + 2) \tag{5.1}$$

where K is the key length in bits, c the clusters of peers in its radio range in which it participates, n are the end devices in its radio range and therefore a subset of N end devices in the LoRaWAN network and m the associated ones.

Due to limited storage resources and the high number of end devices covered by a single gateway in LoRaWAN, it can be considered to avoid generating a new session key exclusively shared between the end device and the LoRa gateway to which it is associated, linearly reducing the bits occupied by session keys and trying to compromise with the security reasons. However, instead of avoiding generating the association session keys, a better implementation consists in removing the common session key about a sensor when it associates with a gateway since henceforth the common session key is never used anymore. Indeed, in case of a new execution of

the protocol from the end device, a new common session key is generated, as it would be if the reboot were far away in time as the key would no longer be valid. Nevertheless, must be also taken into account that the constrained resources of the LoRa gateways may be insufficient to also storing the minimum required $c + m + 2$ session keys.

The negotiation phase of a session key may be inspired to the join procedure of the OTAA method rather than to Diffie-Hellman or IKE methods that rely on public key infrastructure.
So, the gateways must be provisioned with root keys and these can either be the same used by the end devices and the join server (i.e. the AppKey and the NwkKey) or two different ones. Again the latter option should be preferred as it is reflected in a negligible additional amount of occupied storage space on the end devices of $2K$ bits where K is the key length in bits. Of course, these root keys are shared among all sensors and gateways as there is no a priori knowledge of where a device will be located.

Thus, once an end device is activated, it can generate two nonces (and not one as in OTAA because it has to share the session key with many and not with just another entity) from which, together with the root keys, can derive the common session key to be used to provide CIA to data exchanged with neighboring LoRa gateways. Hence, the end device spreads the newly generated nonces to the gateways in its radio range via a GENERATE_COMMON_KEY message encrypted with one of the two root keys. The gateways will decrypt the message and generate the common session key in turn.
As with the HELLO frame, to deliver the nonces with a high probability, the message can be submitted multiple times and ACKs can be considered.

In the discussion about key generation and management, the last element of the puzzle is the generation of a common session key among nearby LoRa gateways belonging to the same cluster.
There is a big difference with the other session keys because this time the end device is not involved in the communication and therefore the messages over LoRa are exclusively downlinks. This means that if full-duplex radio is not available, then to handle as many end devices as possible per gateway, the available transmission time over LoRa for possible messages is reduced to the downlink time window that may be about 10% of the time. Besides this LoRaWAN limitation, the situation is completely dissimilar because there are multiple peers and only one should generate a nonce to be used to derive the session key or they should collaborate in such generation based on the methods of secret sharing.

To face these issues, one can think that a straightforward and networking efficient solution is to take advantage of the nonces generated by an end device and reuse them with mixed or completely different root keys in a trade-off between security and resources. However, should be remembered that multiple end devices are covered by a gateway and a subset of them can run the algorithm in close intervals in time. So, two gateways covering the same two end devices running the algorithm close in time can receive different nonces when they receive the first message as a message sent by the first device could be lost. Furthermore, two gateways having each other in the radio range can have a different set of devices in their range. So, reusing end device nonces generated for the other common session key type is not feasible.

Conversely, the solution consists of a gateways key agreement stage in which nearby peers collaborate to create a session key. This can be achieved by cooperating to create nonces in a scenario similar to secret sharing or running a key agreement protocol over IP. Nevertheless, IP does not provide an accurate location, so, to respect the locality constraint according to which a session key is exclusively shared with peers in the radio range, gateways can share their IP addresses over LoRa (instead of actual values to create nonces) and then generate session keys over IP with one of the two methods aforementioned, thus overcoming LoRaWAN limitations. In the end, it is reduced to a problem of building a common set of IP addresses among peers.

The gateway activation procedure can be triggered by the first message broadcasted by an end device during its activation method. So, in this phase, the gateways in the end device range send downlinks to spread their IP addresses to nearby peers and provide the end device with the Join Accept response once received from the network server. In this way, gateways and at least a device activate simultaneously, so that the gateway session keys are generated and guaranteed to be used before expiration unlike if the gateway activation happens too before an end device activation. If gateways and end devices are represented through a graph, all gateways in the graph connected component to which the end device belongs are activated because gateways in the range of the end device send their IP addresses to nearby peers in their range and these do the same when receiving a triggering HELLO_GATEWAY message. Of course, all gateway outside the graph connected component are not activated and this is perfectly fine for the protocol objective because they are "invisible" to the end device algorithm run as they are out of range also for the chain of neighboring gateways and therefore is useless to activate them until a sensor in such a connected component will not be activated to avoid an early expiration of the session keys. When sufficient retransmissions of HELLO_GATEWAY message are met, gateways can share collected sets over IP to detect clusters they belong to

and run a key agreement protocol per cluster.

Clearly, the gateway activation process takes a longer time than the end device activation (n messages against two), so, once a sensor is activated, it can wait some minutes before initiating a Gateway-Device Coordination Protocol run. If in the elapsed time, a gateway activation is not already finished, the gateway needs to respond to the end device to wait more before sending again the GENERATE_COMMON_KEY message. In fact, if at least a gateway in the radio range of the end device has not completed the activation, it does not have a common session key for at least a cluster, so, it cannot securely forward the sensor's requests to neighbors. Additionally, the response may be not received by the sensor and this situation can represent an issue due to the time uncertainty when the gateway activation will finish. However, when the sensor needs to explore the nearby network structure and a gateway is not still activated and its message to wait has been lost, the gateway can respond again to wait to avoid introducing security risks. Nonetheless, if multiple gateways belonging to a cluster are in the radio range of the end device, then it is unlikely that a message claiming to wait will not be captured by the sensor. Anyway, as stated above, this situation is very limited if the end device wakes up a sufficient number of minutes (e.g. 5) after its activation to execute the protocol.

A cluster of gateways is defined as the set of peers that are in each other range. This property is essential in the common session key generation because it is not sufficient to include in the collected list of G_1 an IP address of G_3 provided by a nearby gateway G_2, whose IP address is in the list, to assess that the new peer G_3 is also a neighbor of G_1. In fact, even if the IP address of G_3 could be not received due to lost messages (retransmissions until a threshold makes this situation very unlikely), an equally valid reason is that G_1 and G_3 are not in each other radio range. Thus, if this is the situation, it is not correct for a gateway to create a single session key to communicate with nearby peers because it can have in its radio range peers that are not in each other range and if the gateways share lists as stated before, this leads to a single session key where also non in range gateways are included. Reasoning by induction, in a chain of gateways, a single session key is generated for all the gateways belonging to a connected component of the graph, making useless the keys whose objective is to cluster gateways.

So, the cluster property implies moving from a unilateral to a bilateral constraint when adding a peer to a cluster and, of course, a gateway participated in at least a cluster. In this way, every gateway needs to create at least a common session key per cluster to separate the cluster communications. Clearly, this implies that to contact neighbors, a message must be sent at least once and it is an issue for transmitting

over LoRa due to its limitations. However, as at this stage gateways have collected the list of neighbor IP addresses, they can directly communicate with peers over IP, improving performance at the same time. So, gateways can only rely on what they collect and should not include in their sets peers from which a message has not been received. Therefore, each gateway shares the collected set over IP to IP addresses included in it, and peers that receive the message evaluate the intersection among the received set and theirs to detect clusters they belong to.

Chapter 6

Gateway-Device Coordination Protocol: Design

To minimize data traffic size and data accessing delays (maximizing the QoS) by achieving the **location-aware functionality**, LoRa is particularly useful since as any other radio technology its characterized by a limited range function of the transmitter location. This is most pronounced in urban environments where the covered area is close to the transmitter rather than in rural settings.

The idea is to realize a **layer of Rendezvous Points** (RPs) [68, 20] represented by LoRa gateways (Unidata routers) in which data produced by end devices (water metering sensors) are processed according to client requests (through stream processing engines) at the best fit LoRa gateway for every end device. This is selected by the sensor itself based on multiple metrics (such as RSSI, CPU-load, GPU-load, RAM load, storage I/O statistics, network statistics, ...) by running a distributed algorithm. So, data streams are dealt out on RPs based on their proximity to the source (i.e. sensor) and on the workload they can handle (i.e. available resources).

Has been proved that the traffic generated by an end device and traveling the network infrastructure till the cloud is reduced a lot (1000x, from GB to MB per day, in a city deployment of 50000 houses) if processed close to the source [5, 6].

In a large-scale deployment, the end devices can be grouped in multiple LoRaWAN networks and if geographically extended, a sensor may have in its radio range only a subset of all LoRa gateways of the network to which it belongs.
Furthermore, as the LoRa gateways in the sensor's range may not be suitable for the end device (a good load balancing should be achieved by distributing homogeneously the sensors over RPs where possible), levels of indirection are needed to explore the nearby network structure and make better decisions.

Since the Received Signal Strength Indication (RSSI) parameter (like other radio parameters) about a LoRa gateway is only available at the end device if the gateway is in the sensor's area (min sensitivity -120 dBm, max sensitivity -30 dBm [61]), outside gateways RSSIs can be obtained through reachable gateways and then properly calculating an indirect value which estimates the RSSI at the end device. In the end, a chain of delegation can be built before selecting the best fit RP where the end device asks reached gateways so far (current level i) to forward its request to neighbors (next level i+1) only if no received response meets the protocol metrics.

A more complex alternative consists in deriving a device position via triangulation or multilateration of at least three LoRa gateways values which receive the device's messages [53]. In this case, more work is moved at the gateway side rather than before because beyond forwarding frames and processing their contents, gateways have to collaborate on performing calculations for each end device and LoRa gateway. However, the perspective can be reversed and the collaboration may be implemented among end devices. Anyway, LoRaWAN limitations can be a serious issue for such an implementation.

In the proposed protocol, the decision to start and execute the algorithm on the end devices rather than on the network server is due to the greater scalability offered by a distributed version compared to a centralized one. The resulting benefit is particularly noticeable in a deployment composed of hundreds or thousands of sensors where centralized management may result in worse performance, even more pronounced if the network server is not located at the edge.

For this reason and to better distribute end device associations on gateways, the selection strategy of the best fit gateway is the responsibility of the end device is running the protocol. Indeed, this allows to explore the nearby network infrastructure if gateways in the radio range of the end device do not satisfy metrics and that would not be possible if the selection strategy is carried on by the network server, choosing, for example, as the best gateway one among the gateways in the end device's radio range from which it receives uplinks. Clearly, in such a case, a denser area leads to overloading the affected gateways because there is no support from the neighbors, as opposed to the proposed distributed solution. Although the network server strategy can consider the number of associations performed by gateways by selecting the one with the fewest bindings, inside or outside the end device radio range, it misses any information about the location of the end device and the gateway because the IP address by itself does not provide an accurate location of the device to which it is assigned, unlike the more reliable RSSI used by the proposed solution to estimate the distance to the transmitter.

6.1 Protocol message definition

In the proposed Gateway-Device Coordination Protocol, the following set of LoRaWAN messages is defined to associate an end device to its best fit RP:

- **GENERATE_COMMON_KEY**: sent by a sensor to a RP to generate a common session key and include two nonces of respectively three and two bytes to be employed in a key generation process based on the OTAA method,
- **HELLO_GATEWAY**: sent by a RP to a peer to share its IP address,
- **HELLO**: sent by a sensor to a RP inside the radio range to request information about its state (may optionally include the device EUI),
- **FORWARD**: sent by a sensor/RP to a RP outside the radio range to indirectly request information about its state (may optionally include the device EUI). It is intended to be forwarded by gateways that already answered to the end device in the previous level (or round) of the current algorithm execution while to be responded by unreached gateways so far,
- **STATS**: sent by a RP to a sensor/RP in response to an end device request and include information about the RP itself (such as the CPU-load, GPU-load, storage I/O statistics, network statistics) besides the RP's address (or EUI),
- **PAIRING_REQUEST**: sent by a sensor to the selected RP to associate with and include the RP's address (or EUI) and optionally the sensor's EUI,
- **PAIRING_ACCEPT**: sent by a RP to a sensor/RP in response to an end device request to accept or refuse a pairing attempt,
- **CONNECTION**: used both for uplinks and downlinks. Sent by a sensor to the LoRaWAN network server to notify it of the new association is taking place and includes the RP's address (or EUI). Sent by the network server in response to a sensor to confirm or not the association,
- **GENERATE_ASSOCIATION_KEY**: used both for uplinks and downlinks. Sent by a sensor to the selected RP with whom it is associating and include the RP's address (or EUI) and a nonce (as in the OTAA method). Sent by a RP in response to a sensor to provide it with the second nonce from which derive the session key,
- **DATA_PROFILE**: used both for uplinks and downlinks. Sent by a sensor to the associated RP to upload the data profile of the sensor on it. The communication consists of a data transfer where all payload is fit with a

portion of the data profile that is fragmented in multiple messages. Sent by a RP in response to a sensor to confirm the receiving of the previous fragment,

- **DATA**: sent by a sensor to the associated RP to provide it with collected measurements and the timestamp when these are collected.

Every message is therefore mapped to the port field of the LoRaWAN frame as represented in table 6.1.

Table 6.1. LoRaWAN message mapping

Port	Message
1	GENERATE_COMMON_KEY
2	HELLO_GATEWAY
3	HELLO
4	FORWARD
5	STATS
6	PAIRING_REQUEST
7	PAIRING_ACCEPT
8	CONNECTION
9	GENERATE_ASSOCIATION_KEY
10	DATA_PROFILE
11	DATA

Moving the focus on the IP infrastructure, the following set of messages (to be sent exclusively over IP) is defined to associate an end device to its best fit RP:

- **NEARBY_GATEWAYS**: sent by a RP to the set of peers whose IP addresses are collected through the receiving of HELLO_GATEWAY messages spread over LoRa. It includes the whole set of nearby gateways IP addresses in order to detect clusters by a common set of items,
- **SYNC_COUNTER**: sent by a RP to the network server to update the frame counters about an end device and includes the sensor's address (or EUI),
- **FORWARD_OVER_IP**: sent by a RP to the appropriate destination (peer or network server) to forward all messages dispatched by an end device over LoRa except the HELLO message and to send the corresponding replies back to the RP that contacted the recipient,
- **CONNECTION_GATEWAY**: sent both by RP and network server. Sent by a RP to the network server to notify it of a new association between the RP itself and a sensor and includes the sensor's address (or EUI). Sent by the network server to a RP to confirm or not the association,

- **PROCESSED_DATA**: sent by a RP to the network server to provide it with processed sensor data and corresponding timestamp when was collected to enable historical analysis on the cloud thanks to the permanent storage layer

As for LoRa messages, each IP message is mapped to an upper layer TCP or UDP port as represented in table 6.2.

Table 6.2. IP message mapping

Port	Transport Protocol	Message
5000	TCP	NEARBY_GATEWAYS
5001	TCP	SYNC_COUNTER
5002	UDP	FORWARD_OVER_IP
5005	TCP	CONNECTION_GATEWAY
5010	TCP	PROCESSED_DATA

The choice between UDP and TCP depends on how important is that the message is delivered as TCP implements retransmissions based on acknowledgments. Specifically, only forwardings are transmitted over UDP as the burden of managing retransmissions is entrusted to the sender. Processed data are sent over TCP/IP but if the data streams represent a continuous massive volume, if the IoT application has an acceptable tolerance to missing values, then data can be sent using the lighter UDP protocol. In any case, both the LoRaWAN gateways and the network server listen on the specific ports for incoming connections.

Due to LoRaWAN constrained networking resources discussed in section 3.1.2, in order to support any data rate across all region specifications, it is necessary to minimize the payload size of the LoRaWAN frame (i.e. employ at most 11 bytes). To accomplish this, if the algorithm run makes use of device EUIs, an idea could be hashing them to at least halve their storage sizes (from 64 bits to 32 or fewer bits) based on a collision probability influenced by the number of deployed end devices and the collision resistance of the selected hash function which has to be not too CPU intensive and therefore compatible with embedded systems. In the same way, the gateway IP addresses must be hashed if using IPv6 as the addresses are 128 bits in size (16 bytes) while should not be necessary for IPv4 addresses that are 32 bits long. However, while hashing device EUIs may not lead to great optimizations since the resulting codomain of the hash function should not be excessively less than the number of deployed devices to avoid collisions with high probability, hashing gateway IP addresses can provide an effective optimization as a few concentrators

can cover very large-scale LoRaWAN networks. In any case, it is always a good idea to minimize the frame sizes to achieve better networking performances.

As the reader may have noticed, the device EUI is optional in some messages of the proposed protocol (such as HELLO and FORWARD) and is opposed to the device address (i.e. use one of the two but not both) because the device EUI should not be necessary in standard LoRaWAN deployments since the device address (included in the LoRaWAN frame header as described in section 3.1.2) should be sufficient to identify every sensor is requiring an association with a RP among concurrent executions.
Indeed, although the device EUI is globally-unique while the device address is unique only within the network the device belongs, so, the same address is probably received by multiple devices on different networks [23], the latter represents a necessary and sufficient condition to achieve the task as any LoRaWAN network runs the algorithm independently (because an end device looks for a RP to which connect on the current network where this is executed to accomplish the location-aware functionality). This statement does not hold anymore in a particular very large-scale deployment where the limited number of gateways in a network is insufficient to handle the very large amount of data streams produced by IoT sensors and therefore a gateway capable to accommodate the stream processing can be searched on a nearby external LoRaWAN network.

The difference between the HELLO and FORWARD messages is the former is intended for gateways in the range of the end device while the latter for out-of-area gateways that are contacted by neighbors whose STATS responses have been already collected and elaborated in the previous algorithm round.

To avoid already analyzed LoRa gateways to repeat sending the STATS message during an algorithm execution and hence obtain an exactly once STATS receive behavior, an idea is to build a blacklist of RPs to include in the FORWARD message. Since the blacklist of RPs is incrementally created by an end device looking for a LoRa gateway to pair with, its size (and therefore the payload size of a data frame) grows over time. As in LoRaWAN frame fragmentation is not natively supported (extensions are available but not suitable for the association task), a workaround may be to limit the blacklist size to the available payload size but this may cause issues or performance degradation that must be carefully evaluated.

Discarding the idea of the blacklist (due to its growing nature), an alternative approach may be relying on the tuple storage (end device address, timeout) or (end device EUI, timeout) when a STATS message is sent by a RP. If properly tuned, the timeout can last for all the algorithm execution (should be short but pay attention

to duty cycle restrictions), and once expired, the entry in the table can be removed and the RP can be available again for future executions of the same sensor (for scale-out tasks for example).

Another solution based on the same approach consists in replacing the timeout with a request ID and therefore in storing the tuple (sensor address, request ID) or (sensor EUI, request ID) on the LoRa gateway, where the request ID is included in the FORWARD message and is unique during an algorithm run. The request ID can be represented by a simple counter such as to obtain a repetition very far in time using a few bits (e.g. 1 byte $\Rightarrow$ 256 algorithm runs before a repetition occurs) or implementing a pseudo-random ID generator.
Thus, it identifies a run of the protocol started by a specific end device and when changes denote a new execution of the algorithm is fired.

A new run is expected to happen in rare cases such as an end device or gateway battery substitution, an end device network rejoin, an end device or gateway failure and similar. However, when the end device reboots, as the device address is dynamically assigned by the join server, a new address can be received by the end device and therefore the old request ID has no relation with the new algorithm run, so, it could also be the same.

In other cases, a new protocol run can be expected to be executed at constant intervals of time so that the tree structure composed of links (end device, selected gateway) is regularly changed for obtaining variable node distributions over time and consequent load balancing. Unlike before, here the end device address does not change, so there is a relation with the old request ID and must be different to distinguish among runs otherwise the gateways will not respond to HELLO and FORWARD messages because they have already replied to such a request.
However, the RPs do not need to persistently keep an entry and perhaps update it in the future. What matters is that the entry lasts for an entire algorithm run to obtain the exactly once behavior, therefore, to save storage space a timeout can be set to remove an old entry when the corresponding algorithm run is surely ended (e.g. 10 minutes, half an hour, 1 hour, ...) like the Time-To-Live (TTL) of a temporary address.

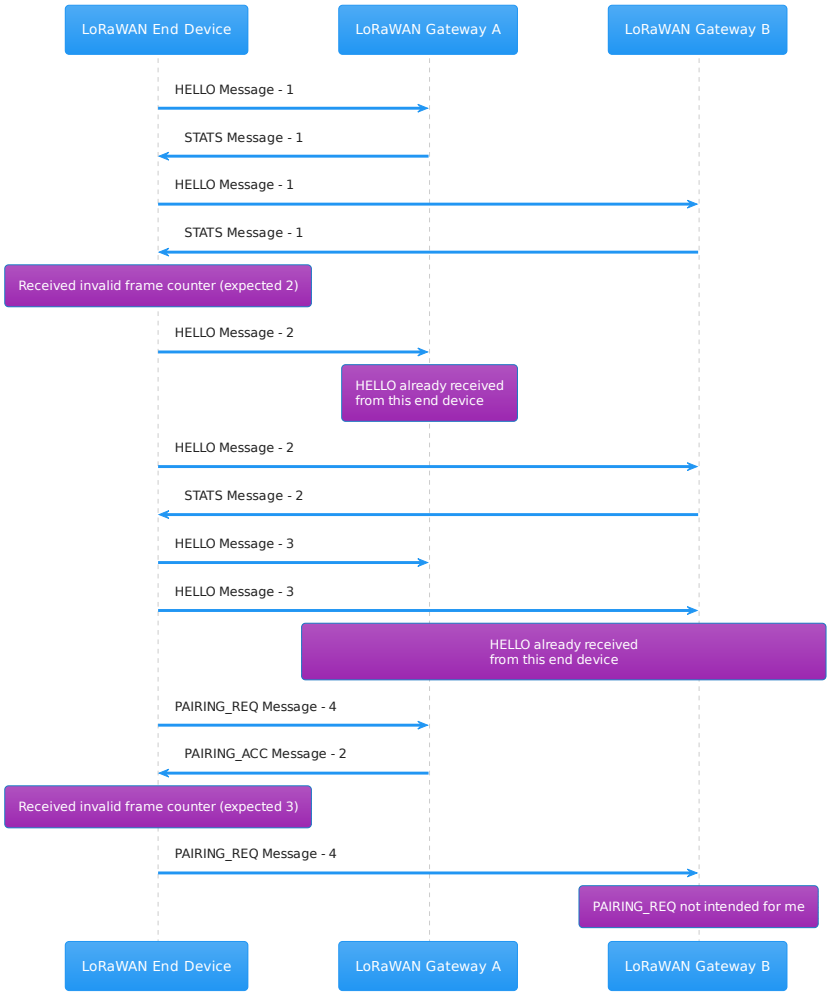
In the last two proposals (timeout and request ID), the message size is fixed and always fits the lowest available LoRa frame payload size (e.g. payload of 11 bytes available and just 1 byte is used for the request ID). Besides, as stated before, small data frames are always preferable for achieving better network performances.
As the first alternative is timeout-based while the second is ID-based, the latter may be preferable (although it requires a slightly larger payload size of 1 byte) because

its substantial temporal independence avoids possible issues related to network ones by design. Furthermore, two different algorithm runs of the same end device can always be detected regardless of their execution over time.

In addition to the request ID (which replaces the blacklist of gateways as a method of obtaining exactly one STATS message receive per LoRa gateway reached by the algorithm), the FORWARD message expects a sequence number such that the storage of the tuple (sensor address, request ID, sequence number) or (sensor EUI, request ID, sequence number) on the RP after sending the corresponding FORWARD message avoids the creation of cycles between gateways that represent infinite forwarding loops.
However, as described in section 3.1.2, LoRaWAN already defines a mechanism to detect retransmissions through the Frame Counter field of the application layer header of the LoRaWAN frame, so, it is redundant to add a custom sequence number in the payload of the FORWARD message. Thus, in the proposed framework, the frame counter management can be only adapted at the LoRa gateways that in the original protocol is instead performed at the network server. In addition to this, end devices detect and ignore duplicates as well, so, if gateways transmit messages using independent downlink frame counters (i.e. each gateway manage a counter per device) as done by the network server in LoRaWAN specification, the result is the end device can receive invalid frames, therefore, increasing network traffic and latency. Indeed, if peers manage counters independently of each other, they are not aware of other transmissions (two gateways in an end device range may not be in the range of each other) while actually, multiple entities are communicating with the end device. Hence, when the gateway receives a message it is expected to respond to (e.g. a FORWARD or PAIRING message), it replies with its own counter related to the end device but its value can already be used by a peer in a previous transmission. An example of this undesirable situation is presented in figure 6.1.

The problem is the communication is no longer 1-to-1, as in LoRaWAN specification, but 1-to-n and this must be properly handled.
The proposed solution consists in realizing a shared downlink counter among peers so that every downlink message will never result in a duplicate. An efficient implementation for class A devices consists in synchronizing the downlink counter of every gateway with the uplink counter, as the former will never be greater than the latter. This is also effective for out-of-range gateways which, although they have never sent a frame to the end device before, directly include the correct counter value in the header of the response.
Furthermore, the network server must be kept updated as well about the shared downlink counter otherwise it cannot send MAC messages while the algorithm is

Figure 6.1. Gateways independent frame counters example



running. So, the only additional traffic compared to the LoRaWAN standard is represented by SYNC_COUNTER messages sent over UDP to synchronize the network server downlink counter for a given end device.

Whereas the custom sequence number is not needed, it is necessary a method for gateways located in the farther level of the algorithm round to decide when incoming FORWARD frames must be replied to, ignored or forwarded at the next level. To add this functionality to gateways, the sequence number is replaced by the level number (or round number) so that the gateways can discover at which level the current round is collecting STATS messages. Therefore, by keeping the request ID constant during an algorithm run and incrementing the level number step by step, the end device can collect out-of-range STATS messages leveraging on nearby gateway radio ranges.

Of course, this cannot be achieved only using the request ID. Indeed, if only a

fixed request ID is used, then gateways do not know when forwarding a frame to the next level and when it changes from round to round, so, either the gateways will never forward the message to neighbors but only respond with STATS messages once or the gateways after a STATS message has been sent, forward the incoming message from the end device to neighbors without distinguishing between rounds. If instead, the request ID changes between rounds, then gateways always reply with a STATS message as if a new run of the protocol was started.

Conversely, if only the level number is implemented, then gateways can handle with success only a single execution of the algorithm because can distinguish among rounds but not among runs, so, a new message should be defined to restart the process.

Furthermore, introducing the level number may be no longer necessary to have two messages to differentiate between the first level and others; the only motivation to keep them separated is to halve the payload size from two bytes (FORWARD) to one byte (HELLO) to reduce traffic in the first round.

Once a pairing is accomplished, the LoRaWAN network server has to be warned about it in order to update its table of associations. This table is fundamental to make the network server able to redirect a user client, that wants to access specific sensor data in real-time, to the gateway that is in charge of the processing. Indeed, as explained in section 3.1.1, the network server represents the entry point for external connections. This feature, through which the network server has a complete view of the LoRaWAN network state and therefore of the current tree of connections whose links are represented by the associations between end-devices and gateways, that allows making a service addressable is commonly offered by Software-Defined Networking (SDN) technologies by disassociating the data plane from the control plane. This functionality is achieved by sending a CONNECTION_GATEWAY message from the gateway to the network server specifying the sensor address (or EUI) to which it is associating and storing the mapping in a routing table like storage structure. Moreover, the network server adds to the new entry the EUI of the device so that a user does not need to know the underlying LoRaWAN infrastructure to access its data.

However, before adding a new association to the table, the network server has to be certain that it is a real and non a fake attempt, so, awaits the CONNECTION message from the end device for verification. To describe how this is managed in the details, the reader is pointed to chapter 7 where a security analysis about the proposed protocol is carried on.

6.2 Tentative algorithm

In light of these considerations, a tentative algorithm to approximately summarize what has been discussed so far about pairing a sensor with its best fit RP could be the following:

1. The sensor wakes up $t \in [0, 600]$ seconds after the OTAA method is completed and generates two nonces from which it derives the corresponding common session key. Then, it sends a GENERATE_COMMON_KEY message to the LoRa gateways in its radio range including the nonces,

2. Each LoRa gateway in the sensor's area that receives the message verifies if it has valid common gateway session keys (one per cluster). If the check succeeds, then the gateway derives the common session key with the sensor from the provided nonces and sends back a GENERATE_COMMON_KEY message including an ACK, otherwise, it sends a GENERATE_COMMON_KEY response including a NACK. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

3. The LoRa network server receives the SYNC_COUNTER message and updates its uplink and downlink counters for the sensor of interest,

4. The sensor listens for gateway responses and if one includes a NACK, then it awaits $t \in [180, 360]$ seconds before repeating 1., otherwise it sends a HELLO message to the LoRa gateways in its radio range,

5. LoRa gateways in the sensor's area that receive the message reply with a STATS message (at least a gateway should be present otherwise messages from the sensor will be definitely lost) and store the tuple (sensor address, request ID, 0) or (sensor EUI, request ID, 0) to keep track that they have responded to such sensor in the current algorithm execution. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

6. The sensor progressively collects the STATS responses and if a LoRa gateway satisfies the metrics requirements (i.e. CPU-load, RSSI, ...), then the sensor selects the best one and goes to 11., otherwise asks to the LoRa gateways, by sending a FORWARD message specifying the next level of RPs it wants to access, to contact nearby gateways in their ranges, but out-of-area for the sensor, in order to evaluate other STATS messages,

7. LoRa gateways in the sensor's area that receive the sensor's request (level-1) forward it to their neighbor gateways (level-2). Then, store an entry (sensor address, request ID, level number), or (sensor EUI, request ID, level number) if the sensor EUI is included in the message, to remember that the FORWARD message for such sensor in the current algorithm run and level has been sent,

8. The (level-2) LoRa gateways receive the sensor's request and look for a matching entry (sensor address, request ID, level number) or (sensor EUI, request ID, level number) in their tables. If a match is found, then the incoming message is ignored to obtain exactly once receive behavior, otherwise, they reply to the (level-1) gateways with a STATS message and store the pair (sensor address, request ID, level number) or (sensor EUI, request ID, level number) to remember that the STATS message for such a sensor in the current algorithm execution has been sent. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

9. The (level-1) LoRa gateways receive the responses of nearby gateways but out of range for the sensor and if and only if they find a matching entry (sensor address, request ID) or (sensor EUI, request ID) in their tables (i.e. they have previously sent a FORWARD message about the sensor), forward the STATS messages to the sensor itself,

10. The sensor repeats 6. and in case of unsuccessful, a chain of delegated gateways is created until a suitable gateway is found,

11. When a LoRa gateway is finally selected, the sensor requests an association with it by sending a PAIRING_REQUEST message including the RP's address (or EUI) and optionally its own device EUI,

12. The (level-1) LoRa gateways receive the sensor's request and each one verifies if the message is intended for it. If this is not the case, the message is forwarded to the proper IP address via a FORWARD_OVER_IP message, otherwise, the selected gateway evaluates the request and responds to the sensor confirming or refusing it via a PAIRING_ACCEPT message, encapsulated into a FORWARD_OVER_IP packet if it not in direct contact with the sensor. If the pairing is accepted, the gateway notifies the network server about the new association is taking place through a CONNECTION_GATEWAY message including the sensor's address. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

13. The sensor receives the PAIRING_ACCEPT message from the selected LoRa gateway (directly or indirectly) and if the response is positive, then sends a CONNECTION message to the network server including the selected gateway address, otherwise repeats 6. to verify if some other gateway in the current round meets metrics and to eventually explore the next level of gateways,

14. The LoRa network server receives the CONNECTION_GATEWAY message from the selected LoRa gateway and stores the possible association (sensor's address, RP's address) in the corresponding table,

15. The (level-1) LoRa gateways receive the sensor's request that is forwarded to the LoRa network server via a FORWARD_OVER_IP message,

16. The LoRa network server receives the CONNECTION message from the sensor through a LoRa gateway and compares the newly received association with the temporary one received from the selected gateway and stored in the corresponding table. If they match, then the binding is confirmed, otherwise, entities are alerted that the pairing has been refused. In both cases, the gateway is notified with a CONNECTION_GATEWAY message while the sensor with a CONNECTION message,

17. The LoRa gateway receives the CONNECTION_GATEWAY message from the network server and if it is positive, then stores the entry (sensor's address, null) in the corresponding table of paired devices, otherwise, a negative response means that the two endpoints do not agree on the pairing and it can only occur if a session key is compromised and an attack is taking place. Section 7.2 discusses possible countermeasures,

18. The sensor receives the CONNECTION message from the network server through a gateway in its radio range and if the response is positive, then generates a nonce and include it, together with the selected gateway address, in a GENERATE_ASSOCIATION_KEY message, otherwise, the sensor is alerted an attack is taking place as an unexpected event occurs involving a session key is compromised. Possible countermeasures are discussed in section 7.2,

19. The (level-1) LoRa gateways receive the GENERATE_ASSOCIATION_KEY message from the sensor and repeat the check at 12. including the possible forwarding over IP to eventually redirect the request to the right destination. When the selected gateway receives the message, it generates a nonce, from which, together with the other retrieved from the message, derives the relative session key. After having securely stored it holding a reference with

the paired device, the selected gateway includes the generated nonce into a GENERATE_ASSOCIATION_KEY message, possibly encapsulated into a FORWARD_OVER_IP packet if the message was received over IP, and sends it back. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

20. The sensor receives the GENERATE_ASSOCIATION_KEY message from the selected RP (directly or indirectly) and retrieves the gateway's nonce from which, together with the previously generated nonce, derives the relative session key,

21. The sensor sends the first fragment of the data profile that fits into the DATA_PROFILE message to the selected RP,

22. The (level-1) LoRa gateways receive the DATA_PROFILE message and forward it to the correct destination using the FORWARD_OVER_IP message but the selected LoRa gateway which stores the fragment of the sensor's profile in the entry (sensor's address, sensor's profile) of the corresponding table of paired devices and prepares to receive the next. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

23. The sensor repeats 21. until the entire data profile is uploaded on the selected gateway,

24. The LoRa selected gateway repeats 22. and spreads the data profile to the network server via a DATA_PROFILE message once all fragments are received and stored in the relative entry,

25. The LoRa network server receives the DATA_PROFILE message from the LoRa gateway and stores it in the association entry (sensor's EUI, gateway address, sensor's profile),

26. The sensor collects measurements and broadcasts them via a DATA message,

27. The (level-1) LoRa gateways receive the DATA message and forward it to the correct destination using the FORWARD_OVER_IP message but the selected LoRa gateway which processes it and then sends the elaboration to the network server via a PROCESSED_DATA message. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

28. The LoRa network server receives the message from the gateway and forwards it to the cloud for persistent storage,

29. The sensor repeats 26. until a LoRaWAN Rejoin takes place,

30. The LoRa gateways repeats 27. until the corresponding message is received from the sensor,

31. The LoRa network server repeats 28. until the corresponding message is received from the gateway.

In writing the steps of the algorithm, the continuous updates of the frame counters about an end device by the network server when a SYNC_COUNTER message is received, are reported once at 3. to not lengthen the description further.

6.3 Forwarding of messages

When a gateway receives a FORWARD message or messages sent by an end device when finally selects a RP, any possible forwarding on the two interfaces (LoRa and IP) of the subsequent packets must be properly managed.
The FORWARD message is the only frame that is supposed to be transmitted by a gateway to a peer over Lora as nearby gateways but out-of-range for the sender end device are expected to be contacted. However, as the framework includes a gateway activation process during which gateways store the list of neighbors, these lists can be reused to send the FORWARD message to nearby gateways by encapsulating it in a FORWARD_OVER_IP message. Vice versa, for the other messages to be forwarded there is no reason to work under the LoRa limitations (in terms of range, bit rate, duty cycle and so on) when IP connectivity is available and allows to achieve better networking performances. Additionally, LoRa is a radio technology and therefore the frames are always sent in broadcast while the message can be encapsulated into an IP packet and forwarded directly to the next hop to the destination.

Even in the proposed framework, the broadcast communication is not suitable for messages sent by an end device and only intended to be elaborated by the selected gateway (i.e. PAIRING_REQUEST, GENERATE_ASSOCIATION_KEY, DATA_PROFILE, DATA).
As the end device cannot send the message to the chosen RP in a one-to-one channel, to express its decision it has to include the RP address (retrieved from the previous STATS message received) in the PAIRING_REQUEST and GENERATE_ASSOCIATION_KEY messages such that these are only processed by the gateway of interest. In fact, the gateway can compare its IP address and the one

included in the payload of the message and if they match it is the selected gateway and therefore it has to evaluate the request, otherwise, the frame must be forwarded to the proper gateway by encapsulating it into a FORWARD_OVER_IP packet.

However, this works until frames transmitted by an end device are encrypted with the common session key shared with gateways in its radio range as forwarding is accomplished by looking at the PAIRING_REQUEST and GENERATE_ASSOCIATION_KEY payloads.

Since multiple LoRa gateways are typically in the end device's radio range, it is highly probable that duplicates of the message are sent over IP to the right destination but the selected gateway can easily detect them through the uplink counter included in the header of the LoRaWAN frame.

When a session key is agreed exclusively between an end device and the chosen gateway, other gateways cannot inspect packet content anymore, so they need another method to forward data to the final destination. A solution is to store associations also in the gateways that are in the end device's area, so, when a GENERATE_ASSOCIATION_KEY is broadcasted, they know which is the pairing is taking place and therefore can add an entry (end device address, gateway IP address) in an appropriate routing table. In this way, when gateways receive packets from such end device address, then they know how they must be forwarded.

Alternatively, in a trade-off between storage and security, an option is to suppress the agreement of an exclusive session key saving $n(A_{ED} + A_{GW}) \approx 8n$ bits where n are the end devices in the radio range of a gateway and a subset of N end devices in the LoRaWAN network, A_{ED} the length of the sensor's address in bits and A_{GW} the length of the gateway's address in bits.

Another possibility consists in using the exclusive session key without storing the associations on level-1 gateways by introducing an additional ARP-like message exchange over IP to know to which gateway the frame must be forwarded given the address. Clearly, this results in higher traffic and latency.

6.4 Number of replies & lost messages

An important point of the algorithm regards how many replies a sensor has to collect before selecting a RP because it doesn't know how many RPs have been contacted in the current round. The end device can start a timeout when a STATS message is received and if no more responses arrive in such time window, it can select the best RP found so far.
If none, the sensor can send the FORWARD message for starting the next round.

Another approach implies an initial phase in which at each round the RPs send

a HELLO message in response to an end device HELLO or FORWARD. In this case, the sensor knows how many messages has to expect and awaits them before making a decision.

Clearly, in both solutions, a related problem is represented by lost messages. In the first approach, a lost (or a delayed) HELLO or STATS message may lead to a sub-optimal choice of the RP to pair with the sensor.
In the second approach, a lost message may lead to a sub-optimal pairing (sensor HELLO or RP HELLO lost) or to a deadlock (RP HELLO received but STATS lost), so it must be properly handled.

So, since the protocol has to deal with frame loss, another idea is introducing ACK messages where the end device after receiving a STATS message acknowledges the corresponding LoRa gateway. If a RP message is lost, the gateway doesn't receive the corresponding ACK in a time window and resends the STATS message. At the same time, the end device starts a timeout when a STATS message is received. This increases network traffic and is still susceptible to network delays but if timeouts are fine-tuned should always lead to making better decisions than not employing message confirmation (best-effort strategy).

However, it should not be forgotten that the end device communicates in broadcast, so, the ACK for a received frame must include the RP addresses (or EUIs) to which the acknowledgment is referred.
The introduction of a list is subject to the payload size issue discussed in section 6.1 but there is one important difference with the blacklist of RPs: unlike the last mentioned which persists over rounds, the ACK list is round specific. This means that such a list covers fewer RPs but may still not fit the payload size of the LoRa frame. To solve this, a sequence of bits denoting continuation (like an ellipsis) may be employed such that RPs that do not find their ID in the list, restart the timeout for the message confirmation without resending the STATS message and waiting for the next uplink frame. The implementation depends on the LoRa class in which the end device operates because in class A it is impossible to listen to multiple downlinks during the receive windows, so, a list is never sent in uplink but just an address.

As described in section 3.1.2, LoRaWAN already gives the possibility to require confirmation on uplink and downlink data frames via the FType field of the MAC header. Therefore, as for LoRa frame counters and custom sequence numbers, it may not be necessary to define custom ACK messages but it should be remembered that in the original protocol confirmations are managed by the end device and the LoRa network server while in the proposed framework the LoRa gateway must implement such verification.

It follows that standard ACKs work with lost STATS messages because represent

a one-to-one communication where a LoRa gateway sends a frame to a single destination but don't work with lost HELLO messages as they are broadcasted by an end device. Since the message is sent on a one-to-many channel, the ACK can only ensure that one RP has received the frame but does not say anything about who sent the confirmation and any possible missed receiving of other RPs.

Abandoning the idea of ACKs, a straightforward fix consists in repeating the submission of the end device's message n times to reduce the frame loss rate to a very small value. The number of repetitions must be derived empirically because it is influenced by the deployment in terms of the geographical distribution of devices and morphology of the territory but also by unpredictable events (e.g. cars movement, flocks of birds, ...).

The ideal situation is such that the probability of missing frames is close to 0 or, in the proposed framework, the end device knows the number of RPs in its radio range and therefore the number of expected responses to detect missing ones. As the first variable is location and events dependent, the system can only deal with the second to get closer to a perfect delivery of messages broadcasted by a LoRa end device.

To obtain an always valid solution without adjusting a parameter for each deployment, instead of elaborating an algorithm to find nearby gateways, the framework can exploit end device radio capabilities again. Indeed, in the Gateway-Device Coordination Protocol, the sensor evaluates among metrics the RSSI of the received STATS message. If the end device already had a list of radio devices (exclusively gateways is preferable) in its radio range with associated RSSIs before sending the HELLO message, it could easily check if the number of expected STATS responses is reached. If at least one is missing, the sensor would repeat the sending of the HELLO message based on a maximum retry value to avoid getting stuck in an infinite dispatch attempt.
Such a list can be built passively (like a smartphone or a personal computer displays the available Wi-Fi networks with related signal strengths) if LoRa gateways periodically send heartbeat frames that notify the end devices of their presence.
The same applies to the FORWARD message in any LoRaWAN broadcast communication.

Hence, must be assessed whether a repetition of a frame is preferable in case of a broadcast transmission or a regular downlink sending to notify that a LoRa gateway is still operational. As the latter, although very reliable, requires continuous sending and an initial configuration phase in which the end device must listen for incoming messages to collect RSSIs of nearby gateways, from a point of view of battery life, network traffic and LoRaWAN limitations (in case of full-duplex radio is

not available, 90% of transmission time is dedicated to the end devices and remaining 10% to the gateways), a probabilistic approximation resulting from simple repetition of uplink frames may be suitable for the proposed protocol to ensure that all intended devices are reached with a very high probability.

Furthermore, to improve the probabilistic solution, instead of just repeating a fixed number of times the same HELLO or FORWARD frame, the end device can additionally resend it until no reply is received, which means all RPs in the current round have received the message or some RP has never received the request, although, it is unlikely after n retransmissions. For class A devices this approach is essential to collect as many STATS messages as possible but this does not ensure that a RP response is received because, while the uplink is very likely to be listened to by a gateway at least once, the corresponding downlink can be lost if sent once.

To avoid this situation, also STATS messages can be repeated multiple times on the same philosophy of the corresponding uplink. However, this introduces additional traffic and if the end device operates in LoRa class A, there is still no guarantee that a message is received with a high probability although retransmitted because another gateway message can be accepted twice or more and the sending of the message requires be tuned so that it is received after a possible uplink. To stop a gateway to retransmit a downlink and give the opportunity to other peers contacted by the end device to respond to it, custom ACKs composed of the RP's addresses are necessary. At the same time, this does not ensure that the gateway will stop because if the end device does not find a suitable gateway in a protocol execution, then it sleeps a few hours before beginning a new run and therefore the downlink can be repeated uselessly. This means that if downlink retransmissions are expected, these must be upper bounded and since an end awaits just a receive window after the retransmission parameter is met to make a decision, then 1 or 2 attempts are sufficient. Indeed, if the messages are not received by the end device, they result in a small increase in traffic and if the sensor is still active it continues to broadcast messages and the gateway will probably receive a request to be answered.

In this case, the repetition of the HELLO and FORWARD messages with different ACKs in the payload, clearly implies that the default frame counter of LoRa frames is incremented each time because the retransmission includes a different payload. Even if the payload was not changed, by keeping the NbTrans parameter of the end device with the default value of 1, retransmissions of the same frame include different uplink counters, so, the custom sequence number, already excluded in section 6.1, is never necessary.

Nonetheless, the last RP response can be missed as the probability of losing a message from a single gateway is greater than losing two messages from two gateways

and so on. Therefore, to ensure also the last STATS message is received, the time window in which the end device waits for a reply before selecting a gateway when STATS messages are no longer received can be doubled to halve the probability of losing the response. Of course, this does not apply to LoraWAN class A devices.

6.5 Concurrency

Another important aspect to discuss is the concurrency of the algorithm due to its distributed nature. Imagine the situation in which a RP is currently free or sufficiently free to meet the end device's metrics and multiple concurrent executions of the Gateway-Device Coordination Protocol select the same RP. It may happen that the RP does not have enough resources to handle all sensors' requests and/or if the sensors had been aware of the other pairing attempts, perhaps the constraints would no longer have been valid for some of them after the i-th association with such RP and instead of overloading it, another would have been selected.

The issue is that the sensor may have an old view about the selected RP's state. The RP instead is obviously always aware of what happens itself (incoming pairing attempts) and has the actual and updated view of its state. So, a possible solution is that an RP accepts pairing requests till a threshold is reached by responding with an ACK message while further requests are refused and responded with a NACK message denoting a busy state of the RP itself. Clearly, this requires an extra effort by the RP itself because it needs to monitor its resources before accepting a pairing request but allows the protocol to better distribute end devices among nearby RPs. In fact, the end device which receives a negative response continues the research of an RP to associate with as if the selected RP had not passed the metrics check.

Unlike the ACK message, NACK is not natively defined in LoRaWAN but the ACK flag of the LoRaWAN header can be reused together with an additional payload of one bit denoting the NACK flag. This means sending a PAIRING_ACCEPT message with the ACK flag set and including in the payload the NACK flag. Anyway, to make the end device easier to understand, the gateway can avoid setting the ACK flag of the response and instead only use a bit of the payload to denote NACK (0) or ACK (1). This makes the message handling easier and more homogeneous against a payload larger than a single bit.

Nevertheless, this approach is effective only when a gateway is closer to its limit, so may happen that the load results unbalanced if the thresholds are too high or during the evolution of the system. A better approach considers detecting variations of the gateway state between the time t in which the STATS message was received

by an end device and the time $t+\delta$ when the PAIRING_REQUEST message arrives at the same gateway. Indeed, the current state of the gateway can differ from the one in the past on which the end device has based its decision (e.g. new associations and/or new client connections). To deal with this, the idea is to notify the end device about the new gateway state when a PAIRING_REQUEST arrives at a gateway with a timestamp denoting when the STATS message was received such that at least a variation in the δ time interval between the present and the moment in the past is detected. Of course, this requires logging the timestamp on the gateway when an association occurs. So, if nothing happened in the time interval, then the end device decision is still based on a valid state of the gateway and can proceed with the algorithm as usual. Otherwise, the end device is notified of the new gateway state via a PAIRING_ACCEPT message including a STATS_UPDATE flag and the content of a STATS message except the IP address because the end device already collected it. Therefore, the end device can elaborate it, select a new gateway from the updated pool of STATS messages, send a new PAIRING_REQUEST and continue as the algorithm states.

Clearly, this last approach replaces the task of the gateway to monitor its resources with the timestamp comparison because an end device if it is updated about a state will never choose an already busy gateway.

The additional STATS_UPDATE flag further validates the decision of using custom ACK/NACK flags because in this way the new functionality can be easily entered into the protocol without any upheaval. Thus, the single-bit flag is replaced by a two-bits flag to represent the 3 required states: NACK (0), ACK (1) and STATS_UPDATE (2).

In the end, in a typical LoRaWAN deployment, it is common for end devices to wake up at different moments in time to initiate the procedure to join the LoRaWAN network. Of course, this reduces concurrent executions of the Gateway-Device Coordination Protocol and if the time differences are large the algorithm becomes almost sequential. The sequentiality implies minimizing the number of messages that an end device needs to exchange to pair with a gateway because decreases the potential issues of concurrency.

To have a full sequential distributed algorithm, reducing the number of messages and increasing the execution time of the entire setup, devices should transmit one by one. This can be accomplished involving the network server that decides during the entire setup execution time who can perform the exchange of messages while the others sleep.

Another alternative to the gateway notification in case of variations about the gateway state, without converting the algorithm to a sequential one, is to use the

LoRaWAN Rejoin mechanism to re-run the protocol and thus alter the load balancing. The number of messages is greater than the introduction of the STATS_UPDATE flag but, as explained in section 6.1, such a reboot of the system can be launched periodically to change the graph of end device distributions over time, obtaining a dynamic framework rather than a static one.

6.6 LoRaWAN Device Classes analysis

Although class A is preferable due to its limited power consumption as explained in section 3.1.3, its reception behavior poses a problem for the Gateway-Device Coordination Protocol because the end device which sends an HELLO message expects multiple responses but only the first detected and intended for the end device is received. At this point, two opportunities are remaining in class A:

- to modify the listening behavior of the end device, deviating from the original protocol intended for a one-to-one communication with the network server, to support many-to-one communication in receive windows
- to adapt the algorithm to the constraints of the LoRaWAN protocol.

As in the first case, the algorithm can proceed as specified in previous sections but at the cost that the effort needed to implement it may be high and energy consumption increased, it is better to follow the second approach. So, in LoRaWAN class A, as mentioned in section 6.4, the idea of the list of ACKs to include in a repetition of the HELLO and FORWARD messages to acknowledge the LoRa gateways which sent the STATS message in response to the reception of a previous HELLO or FORWARD frame must be abandoned in favour of including just an ACK per message in a retransmission of the aforementioned uplinks. Of course, this negatively affects the algorithm performances because if n gateways are reached by the sensor, then n HELLO messages must be sent to open the relative receive windows, resulting in increased uplink traffic and execution time rather than the first option in which a single uplink or $m < n$ uplinks trigger n responses.

However, considering other LoRaWAN device classes as well, there exists also a third choice in addition to the two just argued since classes are not static and an end device can switch between them as it prefers through decisions made at the application layer. So, taking advantage of this, the end device can work in class B or C while running the Gateway-Device Coordination Protocol and in class A when completed for data transmission.

Analyzing the algorithm performances, class C is preferable as it is possible to receive multiple messages while the end device is listening.

Therefore the end device can switch to class C immediately before starting the protocol and operate in this class until a LoRa gateway is not selected. When finally chosen, the end device can switch to class A for sending the PAIRING_REQUEST message and receiving the subsequent reply.
If a NACK is received and no other RP satisfies the end device metrics in the current round, then it can come back to class C and continue the search for an RP. Otherwise, since the communication between the sensor and the associating gateway is henceforth one-to-one, the end device can continue working in class A to save the battery.

In conclusion, working in class A minimizes energy consumption at the cost of a greater number of messages and latency while class C minimizes the latter at the cost of increased energy consumption. Since latency is essential when an end device collects measurements and these must be processed by the best fit associated gateway, working in class A during the set up is acceptable.

6.7 Final algorithm

Based on the above observations, the algorithm to associate an end device with its best fit gateway needs to be modified as follows:

1. The sensor wakes up $t \in [0, 600]$ seconds after the OTAA method is completed and generates two nonces from which it derives the corresponding common session key. Then, it sends a GENERATE_COMMON_KEY message to the LoRa gateways in its radio range including the nonces and the timestamp when it expires,

2. Each LoRa gateway in the sensor's area that receives the message verifies if it has valid common gateway session keys (one per cluster). If the check fails, it warns the sensor to wait a bit via a GENERATE_COMMON_KEY response including a NACK, otherwise, the gateway verifies if the message is expired. In such a case, the gateway ignores the message, else derives the common session key from the nonces provided by the sensor and sends back a GENERATE_COMMON_KEY message including an ACK if and only if a sufficient number of corresponding uplinks have been received to give the opportunity to the sensor to listen a NACK. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

3. The LoRa network server receives the SYNC_COUNTER message and updates its uplink and downlink counters for the sensor of interest,

4. The sensor listens for gateway responses and if one includes a NACK, then it awaits $t \in [180, 360]$ seconds before repeating 1., otherwise it continues to broadcast the GENERATE_COMMON_KEY message until a threshold of retransmissions is reached and at least an ACK is received. When this condition is satisfied, it sends a HELLO message to the LoRa gateways in its radio range requiring message confirmation,

5. The LoRa gateways in the sensor's area that receive the message reply with a STATS message characterized by the RP's IP address (or EUI) besides the gateway resources state (at least a gateway should be present otherwise messages from the sensor will be definitely lost). Then, they store the tuple (sensor address, request ID, 0, false) or (sensor EUI, request ID, 0, false) to keep track that they have responded to such sensor in the current algorithm execution but they have not received the corresponding ACK. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

6. The sensor collects the first response it receives and in the next transmit window repeats the submission of the HELLO message including the IP address (or EUI) of the RP to acknowledge based on the listened frame,

7. The LoRa gateways in the sensor's area that receive the HELLO message for the first time performs 5. while the others compare the RP's IP address (or EUI) retrieved from the frame with their identifiers. If the check succeeds, the RP takes note that its STATS message has been correctly delivered by setting the boolean value of the entry to true so that it never sends again the STATS message in the current algorithm execution, while if the check fails the RP has not received the ACK in the dedicated time window, so, sends the STATS message again. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

8. The sensor repeats 6. and gateways 7. until the repetitions parameter of the end device is satisfied and no STATS message is listened to in a receive window,

9. Once no STATS message is captured in the last receive window, the sensor evaluates collected responses looking for a LoRa gateway that satisfies the metrics requirements (e.g. CPU-load, RSSI, ...). Then the sensor selects the best fit one and goes to 19., otherwise asks the LoRa gateways in its radio range (by sending a FORWARD message including the next level it wants

to access) to contact some other nearby gateway in their ranges in order to evaluate STATS messages of out-of-range gateways,

10. The (level-1) LoRa gateways in the sensor's area that receive the sensor's request forward it to their available gateways (level-2) over IP encapsulated in a FORWARD_OVER_IP message. Then, they update the entry (sensor address, request ID, level number) or (sensor EUI, request ID, level number) to remember the round of the FORWARD message for such sensor in the current algorithm run. This is essential for the last level when a FORWARD for accessing the next level is broadcasted to discriminate between the round to send the STATS message and the round to forward the request to neighbors,

11. The (level-2) LoRa gateways that receive the sensor's request reply to the (level-1) gateways with a STATS message as previously done by level-1 neighbor gateways. Then, they store the pair (sensor address, request ID, level number, false) or (sensor EUI, request ID, level number, false) to remember that a STATS message for such sensor in the current algorithm execution has been sent but no ACK has been received. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

12. The (level-1) LoRa gateways receive the replies of nearby gateways but out of range for the sensor and if and only if they find a matching entry (sensor address, request ID) or (sensor EUI, request ID) in their tables, forward the first STATS message they captured about the sensor to the sensor itself,

13. The sensor collects the first forwarded STATS message it receives and in the next transmit window repeats the dispatch of the FORWARD message including the IP address (or EUI) of the RP to acknowledge based on the received frame,

14. The (level-1) LoRa gateways repeat 10.,

15. The (level-2) LoRa gateways that receive the sensor's request for the first time perform 11. while others check the RP's IP address (or EUIs) included in the message looking for a match with their own identifiers. If a match is found, the relative RP takes note that its STATS message has been correctly delivered to stop sending STATS messages for the sensor in the current algorithm execution by setting the boolean value of the entry to true, otherwise, the RP has not received the ACK in the dedicated time window, so, sends the STATS message again. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

16. The (level-1) LoRa gateways repeat 12.,

17. The sensor repeats 13., the (level-1) LoRa gateways 10 and (level-2) LoRa gateways 15. until the repetitions parameter of the sensor is satisfied and no STATS message is listened to in a subsequent receive window,

18. The sensor repeats 9. and in case of unsuccessful, a chain of delegated gateways is created until a suitable gateway is found,

19. When a LoRa gateway is finally selected, the sensor requests an association with it by sending a PAIRING_REQUEST message including the IP address (or EUI) of the RP and optionally its own device EUI,

20. Each (level-1) LoRa gateway that receives the sensor's request check if its IP address (or EUI) matches the provided one. If unsuccessful, the RP encapsulates the frame in a FORWARD_OVER_IP packet and forwards it to the final destination, otherwise, the RP is the one chosen by the end device, so, verifies if it has sufficient resources to accomplish the sensor's binding. Then, it responds to the sensor confirming or refusing the association via a PAIRING_ACCEPT message (directly via LoRaWAN or indirectly via IP) and in the former case spreads the connection event to the LoRa network server via a CONNECTION_GATEWAY message sent over IP including the sensor's address (or EUI),

21. The end device listens for an incoming PAIRING_ACCEPT response about the association request. In the case of an ACK, it sends a CONNECTION message to the network server including the selected gateway address (or EUI), while in the case of a NACK or STATS_UPDATE, the sensor repeats 9. looking for a RP in the current level (including the just contacted but considering the new state) or in the next. If none is received, then the sensor repeats the sending at 19.,

22. The LoRa network server receives the CONNECTION_GATEWAY message from the selected LoRa gateway and stores the possible association (sensor's address, RP's address) or (sensor EUI, RP EUI) in the corresponding table,

23. The LoRa network server receives the CONNECTION message from the sensor through a LoRa gateway and compares the newly received association with the temporary one received from the selected gateway and stored in the corresponding table. If they match, then the binding is confirmed, otherwise, entities are alerted that the pairing has been refused. In both cases, the

gateway is notified with a CONNECTION_GATEWAY message while the sensor with a CONNECTION message,

24. The LoRa gateway receives the CONNECTION_GATEWAY message from the network server and if it is positive, then stores the entry (sensor's address, null) in the corresponding table of paired devices, otherwise, a negative response means that the two endpoints do not agree on the pairing and it can only occur if a session key is compromised and an attack is taking place. Section 7.2 discusses possible countermeasures,

25. The sensor listens for an incoming CONNECTION message from the network server through a gateway in its radio range and if the response is positive, then generates a nonce and include it, together with the selected gateway address, in a GENERATE_ASSOCIATION_KEY message, otherwise repeats 9. to verify if some other gateway in the current round meets metrics and to eventually explore the next level of gateways. If no response is received, the sensor repeats the dispatch of the CONNECTION message and awaits the reply, the sensor is alerted an attack is taking place as an unexpected event occurs involving a session key is compromised. Possible countermeasures are discussed in section 7.2,

26. The (level-1) LoRa gateways receive the GENERATE_ASSOCIATION_KEY message from the sensor and repeat the check at 20. including the possible forwarding over IP to eventually redirect the request to the right destination. When the selected gateway receives the message, it generates a nonce, from which, together with the other retrieved from the message, derives the relative session key. After having securely stored it holding a reference with the paired device, the selected gateway includes the generated nonce into a GENERATE_ASSOCIATION_KEY message, possibly encapsulated into a FORWARD_OVER_IP packet if the message was received over IP, and sends it back. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

27. The sensor listens for an incoming GENERATE_ASSOCIATION_KEY message from the selected RP (directly or indirectly) and retrieves the gateway's nonce from which, together with the previously generated nonce, derives the relative session key. If no response is received, it repeats the dispatch of the GENERATE_ASSOCIATION_KEY message and awaits the reply,

28. The sensor sends the first fragment of the data profile that fits into the DATA_PROFILE message to the selected RP,

29. The (level-1) LoRa gateways that receive the DATA_PROFILE message, forward it to the correct destination using the FORWARD_OVER_IP message but the selected LoRa gateway which stores the fragment of the sensor's profile in the entry (sensor's address, sensor's profile) of the corresponding table of paired devices and prepares to receive the next. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

30. The sensor repeats 28. until the entire data profile is uploaded on the selected gateway,

31. The LoRa selected gateway repeats 29. and spreads the data profile to the network server via a DATA_PROFILE message once all fragments are received and stored in the relative entry,

32. The LoRa network server receives the DATA_PROFILE message from the LoRa gateway and stores it in the association entry (sensor's EUI, gateway address, sensor's profile),

33. The sensor collects measurements and broadcasts them via a DATA message,

34. The (level-1) LoRa gateways that receive the DATA message forward it to the correct destination using the FORWARD_OVER_IP message but the selected LoRa gateway which processes it and then sends the elaboration to the network server via a PROCESSED_DATA message. Then, the network server is notified about the uplink and downlink counter values of the sensor via a SYNC_COUNTER message,

35. The LoRa network server receives the message from the gateway and forwards it to the cloud for persistent storage,

36. The sensor repeats 33. until a LoRaWAN Rejoin takes place,

37. The LoRa gateways repeats 34. until the corresponding message is received from the sensor,

38. The LoRa network server repeats 35. until the corresponding message is received from the gateway.

In the description of the algorithm, it is omitted that the transmission and receiving of the sensor are compliant with LoRaWAN device class A windows as well the transmissions of the gateways. Furthermore, as for section 6.2, the continuous updates of the frame counters about an end device by the network server when a

SYNC_COUNTER message is received, are reported once at 3. to not lengthen the text further.

6.8 Performance analysis

Analyzing the performance of the algorithm, the proposed solution assumes that low-level gateway number is always preferable to high-level gateway number because level-i RPs are always closer to the sensor (and should be reflected by higher RSSIs) than level-(i+1) RPs since the possible chain of delegation is shorter. So, the algorithm can be imagined to be composed of rounds (one per level) and finds the best fit RP in few rounds as possible without starting the next round if a suitable RP is found in the current one (i.e. exploring only a subset of all RPs in the LoRaWAN network).

In the worst case, an end device contacts all RPs within the network, or to be more precise within the graph-connected component, it belongs and nobody satisfies metrics requirements. If economically viable, the first and straightforward fix consists in adding some extra LoRa gateways in denser areas to meet the needs of end devices and better distribute the load. This also helps future clients' connections to RPs. Otherwise, another possibility is to adapt the end device's metrics to be gradually more relaxed to meet the constraints easier in a subsequent execution of the association algorithm. Nevertheless, this may not solve the issue or just delay it when clients connect to RPs. As mentioned above, the last resort may be represented by looking for a gateway in an external LoRaWAN network to continue leveraging resources at the edge.

In general, the number of uplinks and downlinks (including OTAA and gateway activation) sent by an end device and a gateway to complete a protocol run is respectively reported in equations 6.1 and 6.2.

$$\begin{aligned} n_{ul} &= (1+r_1)+(r+r_2)+(lr+\sum_{i=1}^{l} a_i + r_3)+(1+r_4)+ \\ &+(1+r_5)+(1+r_6)+(f+r_7)= \\ &= r(1+l)+\sum_{i=1}^{l} a_i + f + 4 + \sum_{i=1}^{7} r_i \end{aligned} \tag{6.1}$$

$$\begin{aligned} n_{dl} &= (1+r_1)+r+(1+r_2)+m+(1+r_3)+(1+r_4)+(f+r_5)= \\ &= r+m+f+4+\sum_{i=1}^{5} r_i \end{aligned} \tag{6.2}$$

where r_i refers to the number of unexpected retransmissions per message type due to bad or loss receiving, r is the retransmissions parameter to maximize message delivery ratio of a frame, a_i are the additional HELLO and FORWARD messages sent to acknowledge gateways, l is the number of levels the protocol explored, $m <= r+a_i$ is the number of STATS messages sent (bounded by the number of uplinks sent in the level), and f is the number of fragments in which the data profile is split. The gap between uplinks and downlinks is evident because the latter is necessarily upper bounded by the former due to LoRaWAN class A device and is limited at most to be compliant with LoRaWAN specifications. This difference is calculated in equation 6.3.

$$n_{ul} - n_{dl} = lr + \sum_{i=1}^{l} a_i - m + \sum_{i=1}^{5} (r_{ul,i} - r_{dl,i}) + \sum_{i=1}^{2} r_{ul,i} \tag{6.3}$$

When no further HELLO message has to be sent than the expected number of retransmissions ($a_1 = 0$), the selected gateway is in the radio range of the end device ($l = 1$), the data profile fits in a single LoRaWAN message ($f = 1$) and no retransmission due to losses occurs, the minimum number of uplinks and downlinks is used to accomplish the association and respectively result in equations 6.4 and 6.5.

$$min\ n_{ul} = 1 + r + r + 1 + 1 + 1 + 1 = 2r + 5 \tag{6.4}$$

$$min\ n_{dl} = 1 + r + 1 + 1 + 1 + 1 + 1 = r + 6 \tag{6.5}$$

Once a gateway is activated, unless a protocol run takes place far in time so that the session key needs to be refreshed, subsequent executions before key expiration do not require sending HELLO_GATEWAY messages and therefore the minimum number of downlinks is reduced to 6 per run that represents the lower bound to accomplish the association.

Chapter 7

Security analysis

As explained in section 3.1.4, LoRaWAN guarantees confidentiality, data integrity and authentication of packets by encrypting their payloads and employing a MIC.

As no message is sent in cleartext and therefore any non-encrypted frame is ignored as well as the frame that fails the MIC validation, an attacker cannot forge a valid message or eavesdrop a valid one and alter its content by for example increasing the frame counter, switching the port field of the application layer frame header, changing the IP address of the selected gateway, inserting a different data profile and so on. So, to perform the aforementioned operations an attacker needs to compromise the victim's keys used for calculating the MIC and encrypting the payload and put in place a man-in-the-middle attack to intercept, decrypt, alter and re-encrypt a victim frame or block legitimate messages and send forged ones and more.

Thus, based on LoRaWAN cryptography and the related key management, the goal of creating fake associations and uploading fake data profiles (perhaps with the final aim of a DoS) is unfeasible.

7.1 Replay Attacks

Since forged messages are almost impossible as explained above, I should consider what may happen in case of a pure replay attack of any frame type, both over LoRa and over IP. In the following sections, the Victim label reports the party whose message has been replayed, the Target is the intended recipient of the replayed frame while the Goal and Consequences labels are self-explanatory.

7.1.1 Replay Attacks over LoRa

- **HELLO_GATEWAY**:

- Victim: Gateway
- Target: Gateway
- Goal: Share the gateways common key with unintended gateways
- Consequences: Increase traffic and radio range of a negligible distance

A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates have no effect as the objective is to build a set and it cannot have duplicates by definition.

A replay attack outside the victim's range implies that unintended out-of-area gateways can be contacted and the victim gateway is improperly inserted in the sets of nearby gateways. So, the victim is then contacted over IP by the out-of-range gateways and, for each of them, it is provided with a set of IP addresses whose intersection with the victim set can result in an empty set or not based on the location of the out-of-range gateway. In any case, this situation is not effective because the bidirectional constraint is not satisfied as the victim does not have the sender's IP address in its set. To succeed, the attack needs to replay out-of-range the packet sent from G_1 to G_2 and vice versa so that each other are fooled to believe that they are in each other range. In the end, the common session key that should be exclusively shared among gateways in the range of each other is instead shared also with out-of-range gateways.

However, the size of the intersection of the two sets (received and collected) should indirectly provide a hint about the location of the two peers. So, obtaining an almost empty set is very difficult when n retransmissions occur and the two gateways are actually in the range of each other. Hence, the victim's gateway could ignore a NEARBY_GATEWAYS message if the size of the intersection is under a certain threshold, limiting the distance to which the HELLO_GATEWAY frame can be replayed.

A better alternative to limit the replay range consists in introducing a fine-tuned timestamp in the message so that the distance reachable by the attacker is reduced or completely canceled. Furthermore, in such a limited interval of time, the message cannot be replayed without the victim receiving it because the attacker cannot change its position before transmitting otherwise the message will be of course received invalid. Indeed, time on air in LoRa is a matter of ms (from a few tens needed for SF7 to a thousand for SF12) and the sender according to the spreading factor used for the transmission decides the corresponding expiration time to limit the out-of-range replay attack of HELLO_GATEWAY. As the solution is time-based, this requires an initial

synchronization phase of the clocks of the peers that must be included in the gateway activation method. So, before creating the HELLO_GATEWAY message, the gateway synchronizes its clock with the server via a well-known centralized algorithm and then proceeds as described in section 5.

- **GENERATE_COMMON_KEY**:
  - Victim: End device
  - Target: Gateway
  - Goal: Share the end device common key with unintended gateways
  - Consequences: Increase traffic and radio range of a negligible distance

  A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area gateways are contacted and, since all the nonces are included in this message, therefore they generate the same common session key used by the victim end device to run the algorithm until it associates with a gateway.

  To avoid this inconvenience, instead of generating both nonces on the end device, the key agreement process can be inspired on the join procedure with the difference that the gateways must also agree on a nonce before sending it to the end device. However, there is the issue that gateways in the range of an end device may not be in each other range and therefore in different clusters. This implies that gateways in the end device range cannot agree on a nonce easily and neither propose a nonce per cluster is a valid solution because the sensor must select the first it receives and broadcast it, coming back to the same situation as before with additional overhead.

  A simpler solution than making out-of-range gateways miss a nonce to derive the session key consists in adopting the same approach of HELLO_GATEWAY replay prevention by inserting in the GENERATE_COMMON_KEY message the timestamp denoting its expiration. In this way, the replay range is very reduced or even completely canceled.

- **HELLO**:
  - Victim: End device
  - Target: Gateway
  - Goal: Realize bad pairings, DoS

  - Consequences: Nothing more than increasing traffic

  A replay attack in the same victim's range can reduce the rate of missed frames and duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area gateways can be contacted and tempted to respond with a STATS message, even if would neither reach the victim's device nor any end device accepts it because the addresses do not match. Indeed, if a RP receives a STATS message, it is not forwarded without previously sending a FORWARD and therefore it is ignored.

  Furthermore, the message is encrypted with a session key shared between the victim's end device and the LoRa gateways in its range such that other gateways cannot decrypt it. If not, the attack can have serious consequences because can lead to a degradation of performances or to an end device incapability to associate with a RP (DoS) in the current algorithm run, as the subsequent real FORWARD message would be forwarded to nearby gateways by gateways already reached by the attacker, assuming their responses are lost, instead of sending the STATS message. Indeed, without encryption, the frame is authenticated as from the victim's device, since it is just replayed, and therefore accepted.

- **FORWARD**:
  - Victim: End device, Gateway
  - Target: End device, Gateway
  - Goal: Realize bad pairings, DoS
  - Consequences: Nothing more than increasing traffic

  A replay attack may involve an end device or a LoRa gateway but the analysis of one of the two applies to both, so I consider the case in which the victim is an end device.

  A replay attack in the same victim's range can reduce the rate of missed frames and duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area gateways can be contacted and tempted to respond with a STATS message if it is the first frame they receive from the victim's device, although would neither reach the victim's device nor any end device accepts it because the

addresses do not match. Indeed, if a RP receives a STATS message, it is not forwarded without previously sending a FORWARD and therefore it is ignored. If this is not the first frame they receive from the victim's device, the message is detected as a duplicate if the real packet has already been received or vice versa with no negative effects.

Furthermore, the message is encrypted with a session key shared between the victim's end device and the LoRa gateways in its range such that other gateways cannot decrypt it. Without encryption, the same considerations of the replayed HELLO hold.

- **STATS**:
  - Victim: Gateway
  - Target: End device, Gateway
  - Goal: Increase traffic
  - Consequences: Nothing more than increasing traffic

  A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area end devices and gateways are contacted. End devices in receive windows can be tempted to collect the message but it would be ignored as the device address does not match theirs. Even if this check is disabled by the attacker through physical tampering of the end device, the request ID (denoting the current protocol execution) does not match. Gateways can be tempted to forward it but it is not forwarded without previously sending a FORWARD and therefore it is ignored.

  Furthermore, if the STATS message is a reply to a HELLO message, then it is encrypted with the session key shared among the RP, the end device requiring the information and the RPs in the end device range, so others cannot decrypt it. Otherwise, the STATS message is a reply to a FORWARD message and it is encrypted with the session key shared among the RP and the RPs in the same cluster, so others cannot decrypt it again.

- **PAIRING_REQUEST**:
  - Victim: End device
  - Target: Gateway

  - Goal: Create fake associations
  - Consequences: Nothing more than increasing traffic

  A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area end gateways are contacted but the message is encrypted with the common session key among gateways in the radio range, so, they cannot decrypt it. Assuming no common session key, the out-of-range gateways would simply forward the message to the proper IP address as specified in the payload.

- **PAIRING_ACCEPT**:
  - Victim: Gateway
  - Target: End device, Gateway
  - Goal: Create fake associations
  - Consequences: Nothing more than increasing traffic

  A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area end devices are contacted but the message is ignored as the device address does not match theirs. Even if this check is disabled by the attacker through physical tampering of the end device, the request ID (denoting the current protocol execution) does not match.

  Unintended out-of-area gateways are contacted as well but they are not expected to deal with such a frame. Indeed, if the selected gateway is in the range of the end device the downlink is directly sent to the sensor while if the selected gateway is out of range of the end device, then a FORWARD_OVER_IP is sent back to the first gateway who sent the FORWARD_OVER_IP including the PAIRING_REQUEST.

- **CONNECTION**:
  - Victim: End device
  - Target: Gateway
  - Goal: DoS

  - Consequences: Serious if network server routing mechanism is exclusively based on current RSSI

A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

A replay attack outside the victim's range implies that unintended out-of-area end gateways are contacted but this is a standard LoRaWAN message, so, just decrease the frame loss rate. The attack can have increased latency (till DoS) consequences if the network server routing mechanism is exclusively based on current received frames. So, if it selects as a gateway to send back the response one that is out-of-range for the end device, the answer will never arrive and the end device will send it again.

Since this message is also sent in downlink by the network server (through the selected gateway) to the end device to confirm the association is real, the same considerations of already discussed downlinks apply.

- **GENERATE_ASSOCIATION_KEY**:
  - Victim: End device
  - Target: Gateway
  - Goal: Share the association key with unintended gateways
  - Consequences: Nothing more than increasing traffic

A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

A replay attack outside the victim's range implies that unintended out-of-area end gateways are contacted but the message is encrypted with the common session key among gateways in the radio range, so, they cannot decrypt it. Assuming no common session key, the out-of-range gateways would simply forward the message to the proper IP address.

Since this message is also sent in downlink by the selected gateway to the end device to provide it with the other nonce to generate the key, the same considerations of already discussed downlinks apply.

- **DATA_PROFILE**:
  - Victim: End device
  - Target: Gateway

  - Goal: Upload profiles on unintended gateways
  - Consequences: Nothing more than increasing traffic

  A replay attack in the same victim's range can reduce the rate of missed frames and possible duplicates are detected via the frame counter (included in the packet application layer header).

  A replay attack outside the victim's range implies that unintended out-of-area end gateways are contacted but the message is encrypted with the exclusive association session key, so, they cannot decrypt it. As they don't know how to forward the frame to the associated gateway, they just drop the packet. Assuming no common session key, the out-of-range gateways would simply ignore the message as they don't have the device address in the list of associated end devices

  Since this message is also sent in downlink by the selected gateway to the end device to acknowledge it about the receiving, the same considerations of already discussed downlinks apply.

- **DATA**:
  - Victim: End device
  - Target: Gateway
  - Goal: Send data to unintended gateways
  - Consequences: Nothing more than increasing traffic

  Observations are exactly the same as the DATA_PROFILE frame but the downlink that here is not present.

In the end, the proposed protocol is not susceptible to replay attacks of any LoRa frame type by design since, as presented, consequences are negligible or null for HELLO_GATEWAY and GENERATE_COMMON_KEY messages and null for the other message types (CONNECTION included if network server routing mechanism is not exclusively based on last receive RSSI).

Anyway, the attack feasibility may be hard by itself because, in a context without replay preventions, to succeed requires location and mobility such that the attacker collects the message within the victim's range and replays it out of the area before the real algorithm reaches outside gateways for example. This means that in a matter of seconds the attacker intercepts the message, goes outside an area of 2-5 Km and retransmits it. This is easier if the attacker is located at the border of the

victim's radio range but, in such a limited time, the attacker's radio range partially overlaps the victim's one, so, its effort may be unrewarded.

As replay attack security is exclusively based on the LoRa frame counter (standard) and cryptography (proposed protocol), and by its nature, a counter will exhaust and restart after its maximum value is reached ($2^{32} - 1$ for the 32-bit LoRa counters), to prevent the attacker can succeed with a replay attack when the counter restarts since the replayed frame as counter greater than the current victim one, proper countermeasures have to be adopted. For uplinks, the end device needs to rejoin the LoRaWAN network to obtain new session keys so that old ones are no longer valid. For downlinks, since network rejoin is not feasible as gateways cover many end devices and they cannot reset each time a downlink counter runs out, they can instead negotiate new session keys with the same final aim of invalidating old ones. In this way, any attempt of replay attack is denied, even if the attacker captures the frame with counter value $2^{32} - 1$ and replays it when its value is greater than the actual one.

7.1.2 Replay Attacks over IP

- **NEARBY_GATEWAYS**:
  - Victim: Gateway
  - Target: Gateway
  - Goal: Mess up nearby gateways sets
  - Consequences: Nothing more than increasing traffic

  A replay attack towards the network server implies that an unintended recipient is contacted but is not expected to receive it, so, just drop the packet.

  A replay attack towards other gateways rather than the actual destination IP address simply drops the packet while the opportune destination can detect it as a duplicate via TCP sequence number (included in the packet transport layer header).

- **SYNC_COUNTER**:
  - Victim: Gateway
  - Target: Network Server
  - Goal: Mess up the protocol, DoS
  - Consequences: Nothing more than increasing traffic

A replay attack towards the network server can reduce the rate of missed frames and possible duplicates are detected via TCP sequence number (included in the packet transport layer header).

A replay attack towards gateways implies that unintended recipients are contacted but they are not expected to receive it, so, just drop the packet.

- **FORWARD_OVER_IP**:
  - Victim: Gateway, Network Server
  - Target: Gateway, Network Server
  - Goal: Mess up the protocol, DoS
  - Consequences: Nothing more than increasing traffic

  A replay attack towards an unintended recipient rather than the actual destination IP address simply drops the packet while towards the opportune destination can reduce the rate of miss frames and possible duplicates are detected via the frame counter (included in the encapsulated packet application layer header). A distinction is represented by encapsulated messages that the gateway only expects to forward over LoRa. In this case, the packet is dropped by the end devices either because the device address does not match or the repetition is detected via the frame counter.

- **CONNECTION_GATEWAY**:
  - Victim: Gateway
  - Target: Network Server
  - Goal: Mess up the protocol, DoS
  - Consequences: Nothing more than increasing traffic

  Considerations are exactly the same as the SYNC_COUNTER packet.

- **PROCESSED_DATA**:
  - Victim: Gateway
  - Target: Network Server
  - Goal: Mess up the protocol, DoS
  - Consequences: Nothing more than increasing traffic

  Observations are exactly the same as the SYNC_COUNTER packet.

In the end, the proposed framework is not susceptible to replay attacks of any TCP/IP and UDP/IP packet type by design. Unintended behaviors are prevented but the system is still vulnerable to a flood attack (both on LoRa and IP) where the content of the messages clearly does not matter. This can cause serious consequences on the networking performances which are already limited in LoRaWAN [2] and can be further degraded in case of an attacker who continuously sends frames over the network without respecting the duty cycle.

Even worse, possible colliding messages can deny any possible frame sent over LoRaWAN to reach the destination resulting in a DoS [91]. Similarly, the attacker can jam and intercept end device frames, alter them and make them unavailable for recipients as the MIC check will always fail [7]. However, such malicious conducts are out of the scope of the proposed framework because should be addressed by the LoRaWAN protocol itself [94].

7.2 Compromised End Device

A different scenario (briefly mentioned above) that has to be analyzed consists of a fake pairing attempt where an attacker controlling a compromised end device X performs a fake protocol execution entering the address in the LoRaWAN frame (and also providing the device EUI if used in the implementation) of an end device Y in the network with the goal of messing up end device Y associations and possibly deny other end devices to connect to some RP.
CIA triad implementation again prevents such malicious behavior because authentication fails since the network server verifies the MIC with the session key associated with the device address included in the header of the message.

Therefore, to bypass authentication, the attacker needs to compromise the session key shared between the victim's end device and the RP in its range and once in possession of it can completely behave in charge of the victim.
In fact, the RP replies with an encrypted STATS message but the attacker can intercept and decrypt it. If the RP does not generate a nonce to include it in the corresponding STATS response together with usual information and to challenge the next PAIRING message, then the compromised end device X can even ignore the reply and directly request the association with the RP.

To mitigate these issues and make the attack attempt more difficult, an additional security control can be added in compliance with the well-known cybersecurity principle called defense in depth. This consists of a redundant control that introducing a small overhead strengthens security because the attack to succeed needs to break

multiple defense levels.
In this case, a check at the network server-side can be added in which using the session key exclusively shared with the victim's end device and therefore different from the compromised key, it can query the victim for confirmation of the association with the RP. The network server can use as a relay node the same gateway from which it received the CONNECTION_GATEWAY message but if the victim's end device is not in the gateway range no reply will ever arrive. So, a more robust solution consists in taking advantage of the network server message routing capabilities to select the best LoRa gateway to reach the victim's end device with high probability. The network server can therefore create a new PAIRING_ACCEPT message specifying the LoRa gateway IP address (or EUI) to which the sensor is trying to connect and inserting in the downlink frame the device address of the victim's end device. As the victim did not request the pairing with such RP, its negative response, represented by a PAIRING_REQUEST message with one-bit payload denoting positive (1) or negative (0) acknowledgment encrypted with the key shared between the end device and the network server, let the network server interpret the previous CONNECTION_GATEWAY request as an attack with the effect of:

- providing a negative reply to the RP to possibly delete the fake association
- alert the victim's end device and the RPs in its radio range to generate a new session key because the current one has been compromised

Therefore, to succeed the attacker needs to break LoRaWAN security but it is crucial managing keys securely, paying attention when they are generated/shared between devices and to their lifetimes, refreshing them regularly.

Nevertheless, although such a network server verification prevents fake associations if the session key between the network server itself and an end device is not compromised, it can be exploited by the attacker to achieve a DoS of the victim's end device due to LoRaWAN transmission limitations. Indeed, if the network server continuously challenges the end device for checking the pairing attempt, the end device can either send in a transmitting window the ACK/NACK to the network server or data as it operates in LoRaWAN device class A.
To avoid this situation, the network server that already has an entry (end device EUI, RP IP address) can simply ignore a CONNECTION_GATEWAY message addressing the same end device. This behavior is acceptable because to accept a new CONNECTION_GATEWAY message the network server must be previously notified as it could be suspicious. In fact, in case of a gateway failure or disconnection with the end device, the sensor using the key exclusively shared with the network server can send a RECONNECTION message to the network server (through another

gateway in the radio range) to remove the old association and thus expecting to receive a new CONNECTION_GATEWAY packet from the new possible selected gateway as a result of a new protocol run.
Indeed, by adapting to the proposed solution the original LoRaWAN specifications where every ADR_ACK_LIMIT uplink frames, the end device expects to receive a downlink frame to confirm the network server is correctly receiving the frames, the end device can be aware of any selected gateway failure or disconnection. On the contrary, to avoid the RECONNECTION message, a mechanism over IP to detect the failure of the gateways is needed.

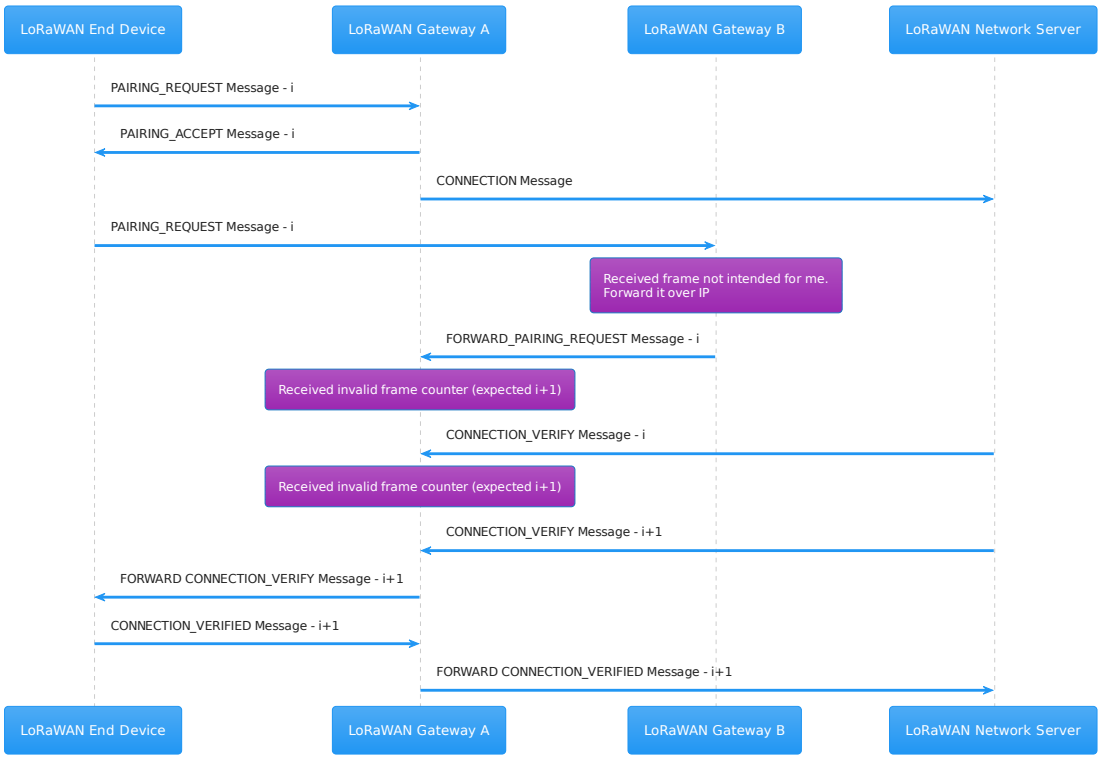
The same issue applies to the scale-out process if the network server does not store only the main RP association but creates a list of connected gateways because needs to verify if the additional RP association is real. Instead of contacting the end device, to avoid DoS, the network server can contact the additional gateway over IP on a secure channel. In fact, only the main RP sends to the network server the message that another RP must be added to the list of connected gateways, so the attacker needs to compromise two keys again instead of just one.

Furthermore, as stated in sections 3.1.3 and 6.6, this solution does not comply with LoRaWAN device class A specification where a downlink for an end device must be buffered and sent out after receiving an uplink from the recipient device. Figure 7.1 represents the just discussed situation and helps to understand why the described solution is only suitable for class B or class C devices.

Nevertheless, such a solution may be harmful also for class B and class C devices because if a CONNECTION_VERIFY frame is lost (either over IP or over LoRa), then the network server transmits a new message increasing the downlink counter, as expected by LoRaWAN specification. Although this allows receiving the message at the end device, this breaks the mechanism of synchronizing the downlink counter with the end device uplink counter as the statement $FCntUp >= FCntDown$ is no longer valid. So, a loss CONNECTION_VERIFY message increase latency because subsequent messages need to be retransmitted at least $FCntDown - FCntUp$ times. As latency must be minimized, this requires designing an alternative more performing method to create a shared downlink counter in case of end devices not belonging to LoRaWAN class A.

Considering these observations and the goodness of class A devices (as described in the aforementioned sections), an alternative verification method for a new association has been designed. Instead of notifying the network server via the gateway on behalf of the end device, it can be directly warned by the sensor itself in order to open the standard receive windows after an uplink transmission. By splitting the previous

Figure 7.1. LoRaWAN network server verification



stage into two, the server verification can be accomplished by performing a simple comparison between information received by the two endpoints. Therefore, the gateway sends to the network server a CONNECTION_GATEWAY message over a secure channel when a PAIRING_REQUEST is accepted including the end device address, and once the end device receives a positive PAIRING_ACCEPT, transmits a CONNECTION message to the network server including the selected gateway address through the session key exclusively shared with network server. If the provided IP addresses match, then the pairing is verified and the two endpoints are acknowledged, otherwise, they are alerted about a mismatch that can only happen if one of the two is misbehaving or a session key is compromised. As discussed above, if the common session key of an end device is compromised, the attacker cannot send the CONNECTION message without having also the other session key shared with the network server but the CONNECTION_GATEWAY message can be triggered. This means that the victim run will result in a mismatch and thus in an alert to regenerate keys. Instead of restarting the protocol execution, a GENERATE_COMMON_KEY exchange can be performed to update the session keys and secure communication to complete the association. In this way, the network server listens and replies to requests as expected in the LoRaWAN standard, without

actively initiating any communication and thus avoiding to ask verification for an association to an end device with related issues as explained above.

Nonetheless, a compromised end device may result in a DoS by blocking any sending, data leak and fake data generation with serious consequences on privacy and consumption measurements. However, device DoS can be easily detected as no data about such a sensor is received over time while fake data may raise an alert about a possible anomaly.

7.3 Compromised Gateway

The opposite situation to the last one discussed concerns a compromised gateway. The impact is greater than before because can be compared to server tampering as it handles multiple requests of nearby end devices and is potentially harmful if not properly managed.

For example, it can start to drop any packet aiming to a DoS that is effective only for end devices that are not in the radio range of at least another gateway. Since bad associations can be performed, as a possible neighbor gateway is unavailable, and peers may be overloaded, this results in negative performance consequences.

Fake CONNECTION_GATEWAY messages to perform fake associations are detected again due to mismatches occurring when the victim end devices select their gateways. As explained above, new keys are generated although this is only successful if the gateway is not physically compromised but exclusively the session key it uses to communicate with the network server because a compromised gateway continues to misbehave independently from the session key in use.

In this case, an effective solution consists of a temporary ban of the gateway IP address after a sufficient number of fake messages is detected to deny its actions which negatively affects the protocol performance.

Therefore, as for compromised devices, the malicious gateway can be noted both if no message or fake message is sent and requiring physical access to restore their state.

Chapter 8

Gateway-Device Coordination Protocol: Implementation

The Gateway-Device Coordination Protocol has been implemented using OMNeT++ [55], a well-known modular simulation software based on C++ for building network simulators. It provides a framework based on events where simulation models are developed by assembling modules, connected with each other through gates that communicate via message passing. OMNeT++ allows to run a simulation model using a graphical user interface, provided with animations, or a command-line user interface and it is available for the most common O.S. (Linux, Mac OS/X, Windows). Last but not least, it supports parallel distributed simulation and especially offers a reliable simulation of real-world scenarios and comprehensive, easy-to-read documentation which, together with simple event programming, enables fast development.

Since the implementation of the Gateway-Device Coordination Protocol requires full flexibility as innovations involve both end devices, gateways and the network server, instead of using a pre-existing OMNeT++ module called *FLoRa* that simulate a standard LoRaWAN network deployment, a new simulation model has been realized from scratch leaving out some aspects that are not essential for the evaluation like power consumption and radio antenna types. In this way, any component of the model has been developed according to the needs of the framework without reworking existing source code.

First of all, the topology of the LoRaWAN network is reproduced in a NED (Network Description) file. Here, the structure of the simulation model is described and simple modules can be declared, connected and assembled into compound modules as networks. Thus, the main components involved in a LoRaWAN network (the network server, the gateway and the end device) are reproduced by corresponding simple modules declared in three separated NED files. The Join and Application

servers are not included in the simulator because they do not participate in the Gateway-Device Coordination Protocol and therefore are not needed to evaluate it. For each module, parameters and gates are defined, so, the radio interface (LoRa) of the end device and gateways is represented through a vector gate able to communicate with multiple parties as well as the IP connectivity of gateways and the network server. Modules are then connected in the architecture NED files by defining networks and specifying gates connections. Two different networks have been developed to realize real and stochastic LoRaWAN deployments: a network reproduces real deployments based on values collected in the LoED dataset [10] while the other network consists of a random number of end devices and gateways positioned in a random way. Unlike networks describing real deployments, the stochastic network only defines IP connections between gateways and the network servers and among all gateways but nothing is said about radio connections to avoid creating links between end-devices and gateways not in the radio range.

Once the structure of the simulation model is defined, the simple modules previously defined and included in it need to be programmed by creating a C++ class for each LoRaWAN component participating in the Gateway-Device Coordination Protocol. In addition to these, custom messages are declared through .msg files to define the LoRaWAN Join-Request frame, Join-Accept frame and physical, data link and application layer frames, assuming the minimum payload available of 11 bytes to support any data rate across all region specifications (section 3.1.2), the IPv4 packet and the TCP and UDP segments.

To evaluate the entire proposed protocol, the initial boot phase including the OTAA Join procedure is implemented as it triggers the gateway activation. However, as stated before, the Join server and application servers are not part of the simulation to focus the implementation exclusively on the new protocol for enabling edge computing over LoRaWAN. This involves compliance to transport protocol specifications and LoRaWAN specification, so, TCP and LoRaWAN verifications and management are also partially implemented (without dwelling on details not important for the purposes of the simulation) in order to obtain a more realistic scenario.

Finally, the network that has to be simulated and corresponding input data for the simulation are specified in a .ini configuration file usually called omnetpp.ini. This allows to separate models and experiments, therefore, the same model can be executed several times with different parameters without having to change the model itself. The input parameters for the stochastic model are the regional parameter (specified through the plan ID [49] denoting where the LoRaWAN network is deployed and corresponding bandwidth, channel frequencies, spreading factors, transmission

powers and duty cycle), the number of end devices and the number of gateways. Additionally, the input parameters for the real deployment model include the RSSIs and spreading factors collected by gateways and reported in the LoED dataset.

Going into detail, in the next subsections, the implementation of the three LoRaWAN components and the evaluation is presented.

8.1 LoRaWAN end device

End devices are mainly provisioned with

- **joinEUI**, **devEUI** and **networkID** identifiers for OTAA,
- **NwkKey** and **AppKey** standard root keys,
- **CommonKey** and **AssociationKey** protocol root keys,
- **NwkSKey** and **AppSKey** standard session keys,
- **CommonSKey** and **AssociationSKey** protocol session keys,
- a unique **device address** assigned at the end of OTAA,
- **fCntUp** and **FCntDown** frame counters for managing retransmissions and detecting duplicates,
- a variable to store the last generated **nonce** for deriving session keys,
- a variable to handle the **transmission and receive windows** of LoRaWAN class A devices,
- a variable to receive a single message after a transmission of LoRaWAN class A devices,
- a variable to handle the **state of the protocol**,
- a **retransmissions** parameter to repeat sending of some messages to minimize the loss,
- the **request ID** and **level number**,
- a variable to store the **selected gateway IP address**,
- the **data profile** that consists of a location identifier string,
- a **set of gateway IP addresses** denoting the sender of STATS messages to resend only the ACK in case of a repetition and not process again the message,

- a **list of gateway IP addresses** to acknowledge in the current round,
- a **map indexed by the score of a received STATS message** and value the list of tuples (corresponding IP addresses having such score, timestamp at which the message was received) and sorted by score in ascending order,
- three **timeouts for handling the transmission and receive windows** of LoRaWAN class A devices,
- **last message sent** and corresponding expected **time on air**,
- the **spreading factor** to use to transmit frames,
- the map of **available bandwidths** and corresponding **transmission powers** according to the regional parameter where the network is deployed,
- the map of **available channel frequencies per bandwidth** according to the regional parameter where the network is deployed,
- the **duty cycle** limit and total time on air usage together with the duty cycle time interval begin,
- **geographic coordinates** about the end device location only necessary in the simulation to respect the LoRa radio range limits,
- a **set of pointers to neighbor end device modules** only necessary in the simulation for handling interferences,
- a **set of pointers to neighbor gateway modules** only necessary in the simulation for handling interferences,
- a **list of neighbor messages** sent by neighbor end devices and corresponding preamble and frame time on airs only necessary in the simulation for handling interferences.

The end device is the key element of the Gateway-Device coordination Protocol because it manages all the stages of the algorithm and transitions among one state and another.

To comply with LoRa radio range limits and simulate a real deployment in every run of the stochastic model, every end device is placed in the range of at least a gateway by selecting a random gateway from the corresponding module vector included in the NED file, obtaining its geographic coordinates and locating the device in the gateway radio range randomly. Then, based on this position, gates connections are exclusively created with gateways in the radio range of the device

by calculating the euclidean distance between the end device and gateway positions. The previous location mechanism ensures that at least one connection with a gateway is set up.

On the other hand, since the position of the end devices is not included in the LoED datasets but the Received Signal Strength Indicator (RSSI) is collected at the gateways together with the device address, the position of the end devices in the real scenario models is estimated by deriving the euclidean distances between the end device and the gateways and calculating the intersection of resulting radio ranges (simplified as circumferences). With two gateways receiving end device messages, the intersection consists of two points and the simulator takes one of the two as both comply with distances from gateways.

When a message is detected by the end device, the Received Signal Strength Indicator (RSSI) is estimated by leveraging on such a distance regarding the sender and receiver in the context of the path loss model [86] as described by equation 8.1

$$PL = -10\ n\ \log_{10}(d) + C \tag{8.1}$$

where n is the path loss exponent, whose value depends on the environment where the wireless transmission occurs and the greater it is, the greater the loss of the environment, d is the distance between the transmitter and receiver and C is the received power at one-meter distance. In the simulation, C is equal to -30 dBm [61] while the path loss exponent varies from 2.4 to 5 and is specific to the configuration.

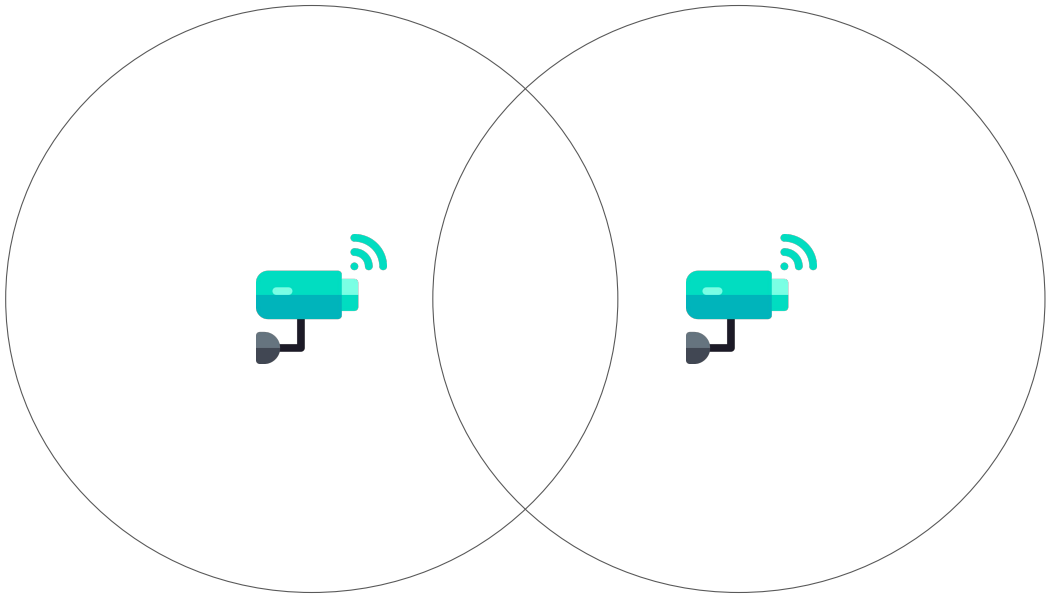
Then, to simulate fluctuations due to environment-specific changes and variables other than the device mobility as it does not fit with a smart water metering scenario, the resulting RSSI value associated with an input signal is picked from a normal distribution characterized by a mean value μ, equal to the estimated RSSI value according to the path loss model, and a small standard deviation σ, avoiding to make this environment variability predominant with respect to the location of LoRaWAN nodes.

LoRa interference is reproduced in a realistic way and is handled by end devices and gateways modules in charge of the role of the physical medium where the radio waves propagate but that is not explicitly present in the simulation. During the initialization phase, the end device collects references to modules of nearby peers in the interference range and gateways in the communication range.

To achieve this, gateways are positioned first and, as sensors are placed progressively, the i-th end device can only collect references about already positioned peers $0, ..., i-1$ and must share its reference to them at the same time for bidirectionality. As smart water metering application is mainly employed in urban

scenarios, the communication range is assumed to be about 5 Km [72, 40] while the interference range covers a larger area that is in particular exactly the double of the communication range because consists of any overlapping between two radio transmissions. In picture 8.1 the radio interference is represented by the intersection of the circumferences and shows why end devices cannot limit to gathering references of peers in the radio range.

Figure 8.1. LoRa interference



According to the region deployment specified in the .ini file, during initialization the end device retrieves the corresponding bandwidths, channel frequencies, transmitting powers, spreading factors and duty cycle. About the spreading factor, in the simulation models, Adaptive Data Rate (ADR) is always on to improve the network scalability but the mechanism is not explicitly implemented to speed up development. In fact, while modules references are collected and shared with nearby gateways, the end device calculates the appropriate spreading factor that should be used to communicate with every gateway based on the corresponding distance. Once a spreading factor per neighbor gateway is collected, the final spreading factor is derived by computing the median value, similarly to the selection performed by the LoRaWAN network server in ADR. The associated transmission power is obtained by the spreading factor following the specific guidelines of LoRaWAN regional parameters [49]. The relative data rate R is therefore calculated as in 8.2.

$$R = SF \frac{\frac{4}{4+CR}}{\frac{2^{SF}}{BW}} 1000 \tag{8.2}$$

where SF is the spreading factor, CR is the coding rate (whose value ranges from 1 to 4 to obtain the actual coding rate values of $\frac{4}{5}$, $\frac{4}{6}$, $\frac{4}{7}$ and $\frac{4}{8}$) and BW is the utilized bandwidth in kHz (this motivates the multiplication by 1000 that is needed to convert the data rate from bits/ms to bits/s) [40].

When a device sends a message, it calculates the expected time on air of the preamble (via equation 8.4) and of the entire frame (summing to the previous airtime the duration resulting through 8.6 equation). Figure 3.2 represents the structure of a LoRaWAN frame.

$$T_s = \frac{2^{SF}}{BW} \tag{8.3}$$

$$T_{preamble} = (n_{preamble} + n_{syncword})\, T_s \tag{8.4}$$

$$n_{payload} = 8 + \max\{\lceil \frac{8L - 4SF + 28 + 16 - 20H}{4(SF - 2DE)} \rceil (CR + 4), 0\}; \tag{8.5}$$

$$T_{payload} = n_{payload}\, T_s \tag{8.6}$$

where T_s is the symbol duration, $n_{preamble}$, $n_{syncword}$ and $n_{payload}$ are respectively the number of symbols of the preamble (8), of the synchronization word (4.25) and of the physical frame payload, 8 is the number of symbols of the LoRa physical header that is summed up with the number of symbols of the LoRa physical frame, L is the length of the datalink frame (including the datalink header, the app layer frame and the MIC), SF is the spreading factor, H is 0 if the header is explicit and 1 otherwise, DE is 1 if low data rate optimization is enabled and CR is the coding rate (in the range 1-4) [49, 45].

Then, the end device updates the consumed duty cycle and notifies neighbor peers by calling a public class method to provide them with the new message and relative time on airs.

Finally, the end device verifies if its message is transmitted during the time on air of at least a neighbor message received through the notification mechanism and used to advise the nearby sensors of a new transmitted message. So, for each possible interfered neighbor frame, the end device checks if the possible interference occurs during the preamble transmission of the peer or not and produces a list that is passed to nearby gateways via the corresponding public class method. From this moment on, the gateways are in charge of handling the interferences, therefore, the other half of the interference implementation is explained in the next gateway section.

To simulate the different awakenings of end devices, they start to run the protocol with a variable delay randomly picked from a uniform distribution on [0, 600] seconds.

As explained in section 3.1.4, once an end device wakes up, it sends a Join Request message to the network server to join the LoRaWAN network including the joinEUI, the network ID and a nonce. It is important to notice that for each message broadcasted by a sensor, according to regional parameters specifications it choices a random bandwidth based on the ADR spreading factor selected during initialization and a random frequency channel on the principle of frequency hopping. Moreover, the appended MIC to the LoRaWAN datalink frame is implemented as specified in the LoRaWAN specifications using the C++ OpenSSL library for calculating the AES-CMAC through AES 128 CBC algorithm. The key involved in the process depends on the transmitted frame, so, in the case of the Join Request, the message is authenticated via the nwkKey. On the other hand, in the simulation, the payload of the LoRaWAN datalink frame is not encrypted since the evaluation is not focused on security as no attacker or compromised node is included in the model. Thus, the overhead of encrypting and decrypting messages is currently not considered although for a maximum 11 bytes payload should be neglectable even for end device constrained resources.

To reproduce LoRaWAN class A device behavior, every transmission is followed by two receive windows implemented by firing timeouts at DELAY_RX1 (1 second) and DELAY_RX2 (2 seconds), where the second window is not opened by canceling the corresponding timeout if a frame is received in the first receive window. To minimize latency, there is no delay after the last receive window is closed, hence the duty cycle constraint is not applied between two transmissions but is taken into consideration in the entire interval of transmission.

The end device message is forwarded by gateways in the radio range to the destination which, if it accepts the request, replies with a Join Accept message properly delayed in the first receive window of the sensor. When the end device listens to the response, it evaluates the MIC for data integrity and authentication using the nwkKey and the OpenSSL library as mentioned above. If the check succeeds, the sensor retrieves the network ID, the assigned address and derives the nwkSKey and appSKey. Then, it cancels the second receive windows and goes to sleep until the next message is transmitted.

From now on, any subsequently captured frame is evaluated exclusively if the MIC is valid, the address specified in the LoRa app layer frame matches and the port corresponds to the expected one based on the protocol stage.

As described in section 6, once activated, the end device wakes up after $t \in [180, 360]$ seconds and starts to transmit GENERATE_COMMON_KEY messages

to generate a common session key with the gateways in its radio range. In the implementation, the device generates two nonces and shares them with the nearby gateways as a mechanism to collaborate between gateways has not been implemented. Furthermore, neither timestamp is introduced in the message because it does not fit in 11 bytes payload if nonces are elaborated as 32 bits. Indeed, as the message includes at least a nonce, it is important that the sender and the receiver agree on the endianness to properly interpret the received bits. Since LoRaWAN transmits bytes as little-endian [50], the implementation respects this convention.

If a NACK is returned to the end device, it falls asleep for $t \in [120, 240]$ seconds by canceling the timeout related to transmission and scheduling it according to t. When retransmissions of the GENERATE_COMMON_KEY message are exhausted, the end device begins to send HELLO frames including confirmations for gateways that reply with a STATS message by removing them from an acknowledgments list of IP addresses. Clearly, since the end device operates in LoRaWAN class A, the size of the list cannot exceed one. The gateway confirmation is preceded by the request ID (randomly generated for each protocol run) and the level number (zero in the case of HELLO message).

The processing of STATS responses represents the core of the protocol because the selection of the gateway to which the end device will ask to associate is based on it. So, for each round of the collection phase (i.e. HELLO and FORWARD) the end device extracts and inserts received IP addresses in the aforementioned list for ACKs (always) and in a set for avoiding processing twice a STATS message from a gateway in case an ACK is loss. Then, the end device takes the rest of the payload and computes a score, including the RSSI value, if and only if every resource load value does not exceed its threshold. This means that if a resource value does not meet metrics requirements, the calculation is aborted and the gateway IP address is not included in the future selection. The score is calculated as a simple sum where all resources have the same weight and the RSSI is added using the opposite of its value for preserving positive values for valid score computations. In this way, lower scores are preferable as denote low resource loads and good signal strengths that should map to shorter distances. Based on this score, the IP address and the corresponding sending timestamp estimated via airtime calculations are joined together in a tuple that is entered at the head of the list of gateways having the same score which therefore represents an entry pointed out by the score itself. The decision of entering the more recent STATS message in the first position is motivated by its proximity in time because minimizes the possibility of receiving a STATS_UPDATE response to a PAIRING_REQUEST message.

Once retransmissions have reached a threshold, the end device only continues

broadcasting the message if a STATS message is detected in the last receive window. If none, the end device gets the first entry from the map of IP addresses indexed by the score in ascending order and pops the head of the corresponding list together with the associated timestamp. Then, includes the retrieved IP address and timestamp in the PAIRING_REQUEST message and broadcasts it. If no gateway IP address is present in the map, then no gateway in the current level satisfies the metrics requirements and the next level is explored via a FORWARD message sent following the same approach of the HELLO message.

Once a PAIRING_REQUEST is transmitted, the end device awaits a response and if none is listened to during the receive windows, it repeats the transmission. If a STATS_UPDATE is returned, the payload after the one-byte flag is elaborated as a STATS message as mentioned above and a new gateway selection takes place. If a NACK is received, a new gateway is possibly chosen while in the case of an ACK a CONNECTION message is broadcasted.

A response is awaited following the same schema of the previous message to move to generate an association session key with the paired gateway. Here, a procedure similar to OTAA takes place, therefore a device nonce is created and included in the payload after the request ID and selected gateway IP address. After possible retransmissions, from the response, the gateway nonce is retrieved and the session key derived.

Unlike the last messages that are all transmitted in a simple and similar way, the DATA_PROFILE frame transmissions need to be properly handled as the data profile must be fragmented in multiple frames.

As the end device expects an ACK about the receiving of the previous packet before sending the next and in case of an ACK loss, it cannot send multiple times the same frame with the same uplink counter otherwise if already received by the gateway, this will never send the ACK to the device, the receiver needs a mechanism to know if the incoming DATA_PROFILE packet has already been received or not. To deal with this issue, the end device includes the starting index from which the data profile is copied.

In the end, once the final ACK about the data profile is received, the end device starts to collect sensor data that are sent to the selected gateway with the associated timestamp.

To have a detailed view of how the end device handles the states of the protocol, the reader is referred to the code available on the following GitHub repository.

8.2 LoRaWAN gateway

Gateways are mainly provisioned with

- **devEUI** and **networkID** identifiers for OTAA,
- **commonEndDeviceKey**, **commonGatewaysKey** and **AssociationKey** protocol root keys,
- **nwkSKey** protocol session key for secure communication over IP and different from the 128 bit key shared between the network server and an end device,
- a **map of common session keys** indexed by an end device address,
- a unique **IP address** dynamically assigned by the network server,
- a **set of variables for monitoring hardware resources** (CPU load, GPU load, RAM load, occupied storage, network I/O stats),
- a variable to handle the activation of the gateway,
- a **set of IP addresses of neighbor peers**,
- a **map of cluster session keys** indexed by cluster session keys and value the corresponding set of IP addresses included in the cluster,
- a **map for handling end device messages** indexed by end device addresses and value a tuple composed of:
  - last message sent to the address,
  - corresponding timeout,
  - request ID,
  - level number,
  - fCntUp,
  - fCntDown,
  - IP address of the associated gateway,
  - a bool denoting if the data profile has been spread to network server,
  - IP address of the peer who forwarded the end device message for retrieving the cluster session key
- a **map of associations** indexed by end device addresses associated with the gateway and value a tuple composed of:

  - symmetric session key,
  - data profile,
  - time at which the association is performed

- a **map for handling TCP/IP retransmissions**,
- a **routing table**,
- three timeouts for monitoring resources, sending the NEARBY_GATEWAYS message and retransmit packets over TCP/IP,
- **geographic coordinates** only necessary in the simulation to respect the LoRa radio range limits
- a **set of IDs of neighbor end device modules** only necessary in the simulation to implement interferences,
- a **map for handling interferences** only necessary in the simulation and indexed by the message ID and value the tuple consisting of:
  - the SINR resulting after applying external noise to the message,
  - the probability that the message is dropped

The gateway is the complement part of the end device in the Gateway-Device coordination Protocol which like a server mainly responds to end device requests. To comply with radio ranges limits and simulate a real deployment in every run of the stochastic model, every gateway is placed in the range of at least a gateway by selecting a random gateway among the already located ones from the corresponding module vector included in the NED file, obtaining its geographic coordinates and positioning the concentrator in the peer radio range randomly. Then, based on this location, gates connections are exclusively created with other gateways in the radio range by calculating the euclidean distance between positions.

On the other hand, the position of the gateways is not included in the LoED datasets but the Received Signal Strength Indicator (RSSI) of every collected end device message together with the end device address. Since in the real scenario models all end devices are connected to the same two gateways, the first is placed randomly in the city while the position of the second is chosen after the euclidean distances between the end devices and the gateways are derived. In particular, assuming the simplification of radio ranges as circumferences, for each end device the distances from the two gateways are summed and the module of the difference is calculated to respectively obtain the max and min distance between the peers and therefore constrain the position of the second gateway in the interval $[|d1 - d2|, d1 + d2]$ where

the circumferences intersect. Thus, from a distance value d randomly picked in the interval and a random angle α, the gateway geographic coordinates are calculated through trigonometric functions as $(x_2, y_2) = (x_1 + d\ cos\ \alpha, y_1 + d\ sin\ \alpha)$.

To monitor gateway resources, a timeout is fired every second to update the CPU load, GPU load, RAM load, available storage and networking statistics. Based on the number of devices associated with the gateway the resource loads are calculated as a sum of values picked from a uniform distribution as described by equation 8.7

$$\frac{1}{b-a} + \sum_{i=1}^{n} \frac{1}{b'-a'} \tag{8.7}$$

where a and b parameters depend on which load has to be estimated and represent the initial interval of values due to run the O.S. and similarly a' and b' denote the impact of each device connected to the gateway for a given resource.

While resource loads are approximately estimated without effectively simulating the number of processes running on the gateway, storage occupancy reflects the actual state of the gateway assuming the availability of an 8 GB size ROM (e.g. Raspberry Pi 4) and an initial occupancy equal to 847 MB due to required installed software such as native Raspberry Pi Operating System (last Lite release January 28th, 2022 has size 482 MB) [66] and Apache Flink stream processing engine (1.14.2 version size is 384 MB) [26]. Every time an entry is persistently stored on the gateway, its actual bit size is obtained and added to the variable in charge of maintaining the total number of occupied bits.

In the same way, network statistics in terms of input and output Kb/s are effectively collected by storing in opportune variables the number of bits received and sent in the last second.

As described in the end device implementation section, LoRa interference is simulated in a realistic way and is handled by end devices and gateways modules in charge of the role of the physical medium where the radio waves propagate but that is not explicitly present in the simulation. During end device initialization, the module ID of the end device is shared with gateways in the radio range and collected in a set.

When an end device notifies the gateway of possible interference, it first verifies if the module ID is included in the set of module IDs of nearby end devices and calculates the RSSI associated with the message. Since a LoRaWAN deployment is not an isolated entity, every transmitted signal is affected by a background noise due to the environment where end devices and gateways are located. In particular, the noise components that disturb a wireless transmission are the thermal noise, the receiver noise and out-band device communication operating on the same unlicensed

band (e.g. Wi-Fi and Bluetooth can negatively affect LoRa signals) [62]. Thermal noise is the electronic noise generated by the thermal agitation of the charge carriers and is therefore present in all electrical circuits. It is directly proportional to the temperature (regardless of the applied voltage) and can affect the sensitivity of radio receivers as the noise can drown out weak signals. Usually it is approximated by an Additive White Gaussian Noise (AWGN) [71] with zero mean and variance its noise power defined by equation 8.8

$$P = k_b \, T \, B = N_0 \, B \tag{8.8}$$

where $k_b = 1.38 \; 10^{-23} J/K$ is Boltzmann's constant, T is the resistor's absolute temperature in K, the resulting product N_0 is the thermal noise power spectral density and B is the receiver bandwidth in Hertz over which the noise is measured. As signal power is often expressed in dBm (decibels relative to 1 milliWatt), equation 8.8 results as described by equation 8.9

$$P_{dBm} = 10 \; \log_{10}(\frac{N_0 \, B}{1mW}) \tag{8.9}$$

and at room temperature (300 K), the thermal noise power is approximately 8.10

$$P_{dBm} = -174 + 10 \; \log_{10}(B) \tag{8.10}$$

where $N_0 = -174$ dBm/Hz is the related thermal noise power spectral density.

Furthermore, in wireless communication, the amplifiers and mixers at the receiver are noisy and the thermal noise power is therefore increased [62]. This noise component is called noise factor or noise figure when expressed in dB and represents the measure of the degradation of the signal-to-noise ratio (SNR) defined by the following equation

$$F_{dB} = SNR_{in} - SNR_{out} \tag{8.11}$$

where F is the noise figure and the SNRs are expressed in dB. In particular, a realistic value for practical applications of the noise figure is $F = 3dB$ for LoRaWAN gateways [71].

Additionally, the more considerable noise component is represented by the out-band device communication operating on the same unlicensed band. For LoRa signals, this is addressed by calculating an external interference probability determined as the complement of the probability of all Bernoulli trials not resulting in interference [88]. The probability takes into consideration the bandwidth of external interference and of a LoRa transmission that is wider in frequency and in most cases shorter in time. The probability an external interference occurs is high and its power is based

on an experimental generalized extreme value (GEV) distribution scaled according to the ratio between the interference unit and the radio signal.

In the simulation, all these components are applied to the notified message by retrieving the physical parameters of the message as spreading factor, frequency channel and bandwidth. Once the thermal power including the noise figure is calculated, the external interference probability is computed, and if this occurs its value is sampled by the GEV distribution. Since an implementation of a GEV type II distribution is not available in C/C++ but GEV type I and in the evaluation only Lora collisions are analyzed, the distribution empirically derived is approximated using a GEV type I distribution defined by a location value of -105 and a scale value of 1.65.

So, the external interference power is possibly added to the noise power in mW and the signal-to-interference-plus-noise ratio (SNIR) of the LoRa message is calculated. Based on this value and a corresponding threshold considering the message bandwidth, a Bit Error Rate (BER) is calculated if the SNIR is below a specific threshold [67].

Now, for every possible interfered message communicated by the possible interferer end device, the gateway verifies if the module ID is included in the set of module IDs of nearby end devices to only assert possible interference for messages in the gateway radio range. In other terms, the gateway checks whether it is located in the intersection of the radio ranges of the two end devices (Figure 8.1). Then, the physical parameters of the message as spreading factor, frequency channel and bandwidth are retrieved and if no external interference has been applied to the possible interfered message, this is enforced, otherwise, the resulting SNIR is retrieved from the map used for handling interferences. Finally, the bandwidths of the possible interferer and interfered frame are compared and if they do not match no interference takes place, otherwise, the gateway checks if the messages are transmitted using the same channel frequency. If this is the case, the Signal Interference Ratio (SIR) between the interfered and interferer messages is calculated and spreading factors are compared. If equals a strong interference is present such that the interferer is always discarded while the interfered is lost only if the SIR exceeds a threshold based on the fact the interferer is transmitted during interfered preamble or not [41]. If instead messages are transmitted using different spreading factors, the interference is weaker and due to the imperfect orthogonality of spreading factors only if the second transmission occurs during the preamble transmission of the other signal. In this case, a SIR threshold and a maximum Bit Error Rate (BER) are assigned according to the spreading factor. If the SIR threshold is exceeded, then a BER is calculated based on an experimental curve [19]. When two messages are transmitted

in different channels, the interference may occur if the interferer is transmitted during the preamble transmission of the interfered and the spreading factor used by the former is at least 10. Based on the channels adjacency, the probability of packet loss is estimated [56].

In LoRaWAN architecture, the gateway is the only device that communicates on two different technologies: LoRa and TCP or UDP over IP. This means that the gateway must appropriately process a packet according to the interface where it is received. During its lifetime, the network server listens for incoming requests on predefined frequencies channels based on the region parameter specified in the .ini file and specific UDP and TCP ports. For simplicity, the gateway is assumed to be full-duplex so that latency is minimized as it can receive and transmit over LoRa at the same time. When a UDP or TCP segment is listened to, the gateway verifies the destination address, the port and the TCP sequence number are valid while security is not implemented as it represents just overhead in the protocol and is not included among the aims of the evaluation. An encapsulated LoRa frame is eventually extracted from the packet if the goal is to forward the message to an end device, otherwise, the message is processed at the gateway. Unlike downlinks, uplinks are validated in terms of MIC, port and frame counter; again the payload is not encrypted because confidentiality is not implemented in the simulation. Based on the LoRa frame application layer port, the gateway knows if the frame must be elaborated at the gateway or simply forwarded to the network server after being included in a UDP segment.

The gateway activation is triggered by a JOIN_REQUEST sent by a nearby end device; the LoRa frame is forwarded to the network server after having been encapsulated into a UDP/IP packet, the clock is synchronized with the network server one and a HELLO_GATEWAY message is broadcasted over LoRa to contact nearby peers and scheduled to be retransmitted a constant number of times if and only if the gateway is not already activated or is not activating. Clock synchronization is not implemented in the simulation as it employs a well-known centralized algorithm to accomplish this goal. The device address used for the transmitted downlink frame is 0.0.0.0 which is never assigned to an end device so that no activated sensor can receive it while the payload includes the gateway IP address and the expiration time calculated on the expected time on air considering a tolerance. Once the network server returns the Join Accept frame, this is decapsulated from the upper layer packet and immediately forwarded over LoRa. When a peer receives a HELLO_GATEWAY message, the frame is validated based on the current time and the expiration time. If it is not valid, then the gateway synchronizes its clock with the network server if not already accomplished in the last seven days, otherwise, IP address is inserted

in the set of neighbor addresses and if the gateway is not already activated or is not activating, then starts the transmission of the HELLO_GATEWAY messages too. Once the message is repeated a sufficient number of times, the gateway cancels the timeout and awaits two transmission windows before evaluating to send NEARBY_GATEWAYS messages or not to maximize the collection of neighbor IP addresses. When this last timeout is fired, if no IP address is received, the gateway terminates the activation as it participates alone to a single cluster otherwise a NEARBY_GATEWAYS message per neighbor is sent over TCP/IP including the list of collected IP addresses. First, the peer that receives such a message checks if the sender IP address specified in the IP packet belongs to the collected set of IP addresses to determine if they are in each other radio range. If this is true, the second step consists in calculating the intersection between the provided set including the sender and destination addresses, and the collected set. Finally, the gateway verifies if the intersection is already present among clusters and if not, then it is added to the corresponding map through a temporary key. Once all NEARBY_GATEWAYS messages are received, all clusters are detected and the key agreement stage over IP begins. In the implementation, the key agreement per cluster and with the network server is not included as a traditional schema is employed and it is not essential for the evaluation.

As mentioned above, the gateway reacts to end device requests transmitted over LoRa, and in the simulator, this communication is fully implemented. When a GENERATE_COMMON_KEY message is listened to, the gateway looks up the map when it keeps track of end device uplinks and if no entry is found, one is initialized and the device nonces are retrieved from the payload and the related common session key is derived and securely stored. Then, the gateway checks its state and if it is still activating, it responds with a NACK otherwise replies with an ACK only after a threshold of incoming retransmissions is exceeded to give the end device the possibility to receive a NACK.

For each incoming HELLO message, the gateway looks up the map storing information about received frames so far and decides if it is valid accordingly. If it is valid, the frame is processed and if the request ID does not match, then the gateway infers that a new protocol run is started and prepares a STATS response. Otherwise, looks for an ACK by comparing its IP address with the one included in the HELLO payload, and it is evaluated if a new STATS message must be sent back to the device. Anyway, the timeout for resending the STATS message is canceled. The response is transmitted in the device receive window that is calculated by considering the uplink transmission air time that is in the order of ms and delaying the response a few moments after the start of one of the two receive windows of the end device,

generally the first. The resource states included in the STATS payload reflect the monitored values in the last second and are followed by the gateway IP address. MIC calculations are performed using the common session key previously generated.

Similarly, for each received FORWARD frame, the gateway looks up the map to compare the received request ID and the stored one. If no entry is found, a STATS message is sent back over UDP/IP to the peer sender, otherwise, the request ID is checked as for the HELLO message and if it is valid the next action depends on the level number. If it is equal to the stored one, then the gateway looks for an ACK and cancels the timeout for retransmissions, evaluating to send or not a new frame. Instead, if the level number is greater than the stored one, the request is forwarded to neighbor peers over IP reusing the set of collected IP addresses since the sensor wants to access the next level of gateways. In particular, this kind of gateway updates the level number value to actual-1 to allow relaying of FORWARD messages including the current level number but not those containing smaller values while a gateway recruited in the current level stores the level number as it is to respond with a STATS message. Furthermore, a STATS message is forwarded by a gateway only if a FORWARD frame has been sent and this is easily achieved by looking for a matching end device entry in the map of received messages and the corresponding level number value. When a gateway has to forward a frame, it must decrypt the message with the proper session key, encrypt it for the other side with the related session key and recalculate the MIC.

When a PAIRING_REQUEST arrives at a gateway, this compares its IP address to the one included in the payload and if it does not match, then the message is encapsulated in a UDP/IP packet and sent to the right destination. Otherwise, the gateway checks its resources by assessing that all resources variables are below a given resource threshold. If a threshold is exceeded, then a PAIRING_ACCEPT with a NACK payload is sent to the device in the corresponding receive window. Otherwise, the gateway extracts the timestamp from the received message which denotes an approximation about the time the STATS packet was sent, and verifies if its status changed by scanning the map of associations performed with the end device looking for one that occurred next in time. Based on this check, the gateway sends a STATS_UPDATE or ACK message as specified in section 6.1 and initializes an entry in the association map. In case of success, a CONNECTION_GATEWAY message including the end device address is transmitted to the network server. It is important to notice that the latter message is sent once with possible TCP retransmissions but not again when a further PAIRING_REQUEST about the same end device due to possible frame loss is listened to minimize traffic. Furthermore, a previously accepted request, in case of frame loss and retransmission may be refused

and in this case, the association stored at the network server must be deleted by sending a CONNECTION_GATEWAY message including a NACK for the given end device address. Again, PAIRING_ACCEPT messages about an end device are forwarded by a gateway exclusively if the PAIRING_REQUEST has been forwarded to the destination that is now the sender. When the gateway sends a CONNECTION_GATEWAY message, it listens for an incoming response to know if the new pairing is acknowledged or refused and therefore countermeasures need to take place.

Uplink and downlink CONNECTION and MAC COMMANDS frames are simply relaid on the other direction as in the standard behavior of the gateway in a LoRaWAN network, so, unlike aforementioned "more complex" forwarding, the frames are dispatched as they are.

The uplink GENERATE_ASSOCIATION_KEY is forwarded to the right destination previously-stored if the end device is not associated with the gateway receiving the frame while the corresponding downlink is only forwarded back if the uplink was previously forwarded. The associated gateway retrieves the device nonce, generates a new one, and derives the association key providing the end device with the nonce.

As an association key is now generated, non-associated gateways forward subsequent messages to the selected gateway in the same way they forward frames between the end device and the network server.

A final remarkable point is represented by the data profile that is fragmented in multiple packets as described above and that expects an ACK from the associated gateway for correct delivery. In case of a lost ACK, to avoid collecting a message twice or more, the gateway must compare the length of the stored data profile with the index provided by the sender and accept the message only if they match. Once it is completely received, when the first DATA message is received, the gateway appends the entire data profile to the common PROCESSED_DATA payload to propagate it to the network server. This means that if the sensor does not send the next message type, its data profile is not communicated to the network server and a task submission of a client in such a time interval (between last DATA_PROFILE sending and first DATA) through the location string would be rejected. Depending on the needs of the IoT application, if it is crucial to make immediately available the data profile on the network server, although nobody can exactly forecast the moment when an end device will be associated with a gateway due to random delays, an alternative implementation consists in employing a byte of the DATA_PROFILE payload to denote its termination and therefore allow the gateway to send it to the network server as soon as possible. In contrast, current implementation privileges dedicate such a byte to a character of the location string to minimize the fragments needed

to transmit it since it is not necessary to immediately propagate the information to the server.

To have a detailed view of how the gateway logic is carried on in the simulator, the reader is invited to refer to the code available on the GitHub repository aforementioned.

8.3 LoRaWAN network server

The network server is mainly provisioned with

- **joinEUI** and **networkID** identifiers for OTAA (simplification without join server),
- a **map of nwkKeys** indexed by devEUIs and value the corresponding standard root keys (simplification without join server),
- a **map of appKeys** indexed by devEUIs and value the corresponding standard root keys (simplification without join server),
- a unique **IP address** (based on the ID of the OMNeT++ module in the simulation run),
- a variable to assign addresses to end devices,
- a **map of EUIs** indexed by end device and gateway EUIs and value a tuple composed of:
  - the corresponding address,
  - devNonce included in the Join Request
- a **map of nwkSKeys** indexed by gateway IP addresses and value the corresponding session keys,
- a **map of appSKeys** indexed by end device addresses and value the corresponding session keys (simplification without join server),
- a **map for handling end device messages** indexed by end device addresses and value a tuple composed of:
  - request ID,
  - fCntUp,
  - fCntDown,

  - a boolean denoting if the CONNECTION_GATEWAY message from the gateway was received,
  - a boolean denoting if the CONNECTION message from the end device was received,
  - IP address of the gateway to associate with

- a **map of associations** indexed by end device addresses and value the associated gateway addresses,
- a **map of data profiles** indexed by data profiles (location strings) and value the corresponding end device addresses,
- a **routing table**,
- a **map for handling TCP/IP retransmissions**,
- a timeout for TCP/IP retransmissions,

Network server work is limited at most in the Gateway-Device Coordination Protocol as it can be deployed either at the edge or at the cloud and in the latter case, intensive communication introduces latencies. It has two main roles in the protocol, one consists in verifying new associations by a mutual verification of the endpoints statements and the other in redirecting clients to the gateway where the end device is associated to read and/or process data.

In the implementation, for simplicity, the network server also acts as Join Server and Application Server, so it stores a network root key and an application root key per end device EUI besides the network ID and joinEUI to activate end devices over the air.

During its lifetime, the network server listens for incoming requests on predefined UDP and TCP ports. When a UDP or TCP segment is listened to, the network server verifies the destination address, the port and the TCP sequence number are valid while security is not implemented as it represents just overhead in the protocol and is not included in the aim of the evaluation. When a FORWARD_OVER_IP packet is received, the encapsulated LoRa frame is extracted and LoRaWAN checks assert the MIC, the port and the uplink frame counter are valid. During end device activation, the MIC, joinEUI and devEUI of the Join Request frame are instead verified. Then, a joinNonce and a unique address are returned to the sender via a Join Accept message and the corresponding nwkSKey and appSKey are securely stored.

To mutual verify a new association, the server listens for CONNECTION LoRaWAN messages encapsulated into a UDP packet and CONNECTION_GATEWAY

over TCP. Usually, the message sent by the gateway reaches the server before the corresponding message sent by the end device. Anyway, the IP address of the gateway is retrieved from the first of the two arriving at the server and is stored in the corresponding map indexed by the end device address. When the message from the other party is listened to by the server, the gateway IP address is obtained from the message and compared with the one previously stored in the map about the same end device address. If matches the server acknowledges both parties, otherwise alerts of a possible compromised session key as endpoint requests do not match. Once an association is verified, the network server stores the new pairing in the map of associations to offer a routing service to redirect clients that require to access and process data about such a device to the appropriate destination. Furthermore, to redirect clients which provide the location string instead of the device EUI to the destination, the network server retrieves the data profile from the PROCESSED_DATA message if any, and stores it in the corresponding map with value the address of the related end device. The rest of the payload is then forwarded to a database located on the cloud but this is not implemented in the simulation.

Moreover, the network server listens for SYNC_COUNTER packets in order to detect retransmissions and be able to communicate with the end device at any moment of the algorithm execution if MAC commands need to be sent.

For details on how this is achieved in C++, the reader is referred to the GitHub repository again.

Chapter 9

Gateway-Device Coordination Protocol: Evaluation

The evaluation is carried on using two different sets of experiments: a first set aims to evaluate the correctness of implementation to assert the protocol works properly or at least as we expect and a second set aims to evaluate protocol performances. A third possible set to compare the proposed solution with related works to evaluate improvements is not conducted as the implementation does not involve stream processing on gateways. However, we can expect that the Gateway-Device Coordination Protocol reduce latency with respect to current LoRaWAN cloud computing architectures as the presented solution enables edge computing and similar works based on the edge and on the fog reduce latency and therefore improve performances [68, 54, 92].

For each simulation execution, several data are collected such as

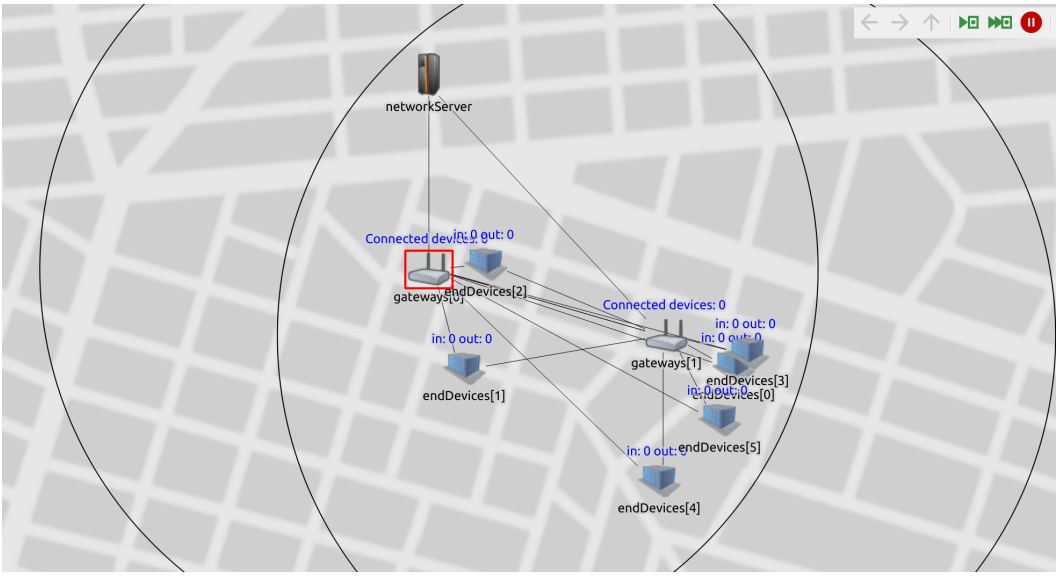
- message sent, received, lost and retransmitted by every end device,
- message sent, received and lost by every gateway (also differentiating between LoRa and IP interfaces and with associated RSSIs for incoming LoRa signals),
- interferences and possible interferences detected by a gateway,
- gateway resources state,
- number of connected end devices to a gateway

where a possible interference is defined as any concurrent transmission of two LoRa signals received by the same gateway while an interference (or collision) is recorded only if the two signals meet the conditions described in section 8. This means that overlapping of multiple signals is not collected as a single possible interference but multiple times based on the involved pairs (collisions).

As explained in the above section, end devices and gateways are located based on the scenario the simulation is reproducing and both stochastic and realistic scenarios have a set of configurations that result in different input data. Specifically, each real-world configuration reproduces a day in the LoED dataset and the selection of a subset of days available in the dataset is made according to the number of end devices and gateways involved and gateway coverages to simulate different network sizes and deployments. To speed up the process and remove human error, the selection is carried on a summary produced by a Python script which elaborates the dataset and produces detailed info about every day of collection including formatted input data ready to use in the simulator.

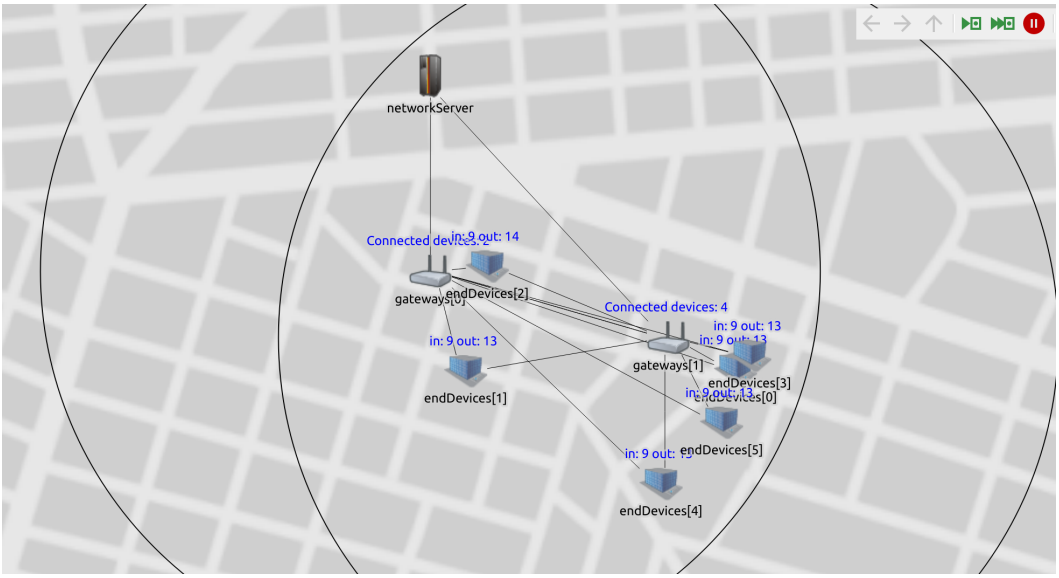
Starting from correctness, an initial evaluation is conducted by setting up simple stochastic scenarios to demonstrate the protocol properly works in different conditions. Small LoRaWAN networks composed of $n \in \{1, 2, 3\}$ gateways and $m \in \{1, ..., 6\}$ end devices are deployed to speed up the simulation and traceback message exchanges. Each test scenario is simulated 10 times reproducing different deployments due to the random location where gateways and devices are placed at each run. A deployment example composed of 6 end devices and 2 gateways is shown in figure 9.1 where lines connecting modules denote the possibility of bidirectional communication and circles represent the approximation of gateways radio ranges.

Figure 9.1. Example of a stochastic deployment execution start - six end devices and two gateways



While this figure represents the initial situation in which devices sleep before activating, figure 9.2 illustrates the opposite situation where the protocol is finished and end devices are associated with a gateway.

Figure 9.2. Example of a stochastic deployment execution end - six end devices and two gateways



Labels over end devices report the number of messages they have correctly received (according to receive windows, MIC, device address and port) and messages they have broadcasted during the experiment while labels over gateways display the number of associated end devices. According to these, the sensors distribute over gateways as expected based primarily on proximity to the gateways as their resources are certainly not so loaded as to choose a gateway further away and the number of messages sent and received by end devices is similar denoting the absence of interferences.

Analyzing a sample end device, figure 9.3 presents the distribution over time of the messages it sent.

The value is constant at one denoting the sensor broadcasts a message at a time as expected by a LoRaWAN class A device. The first message is approximately sent after 50 seconds the simulation starts and consists of the Join Request message of OTAA. After a silence lasting about 400 seconds, a burst of messages is sent at fixed time intervals consisting of the Gateway-Device Coordination protocol messages. Such a burst is more evident in figure 9.4 where the line grows over time every time a message is sent.

On the same approach, figure 9.5 about message receiving reflects the distribution of sending as expected by a LoRaWAN class A device.

The same applies when the line representing the number of received messages is plotted in figure 9.6.

Putting the charts together in figures 9.7 and 9.8 the coupling between sending

Figure 9.3. Single device - LoRa messages sending over time

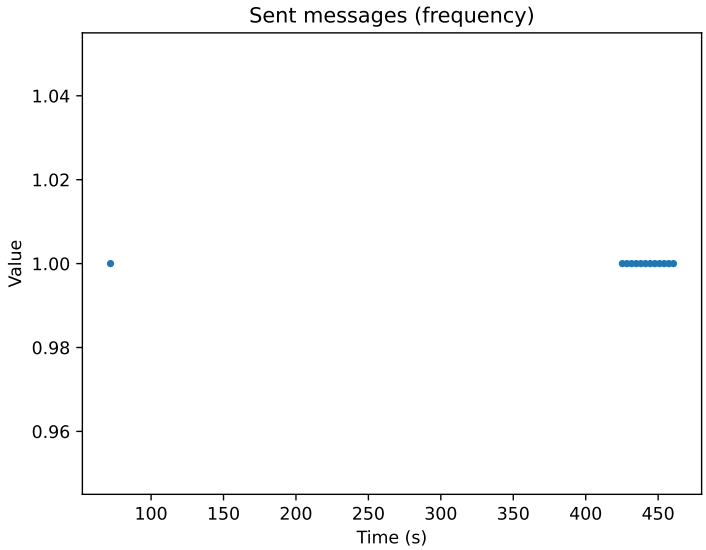
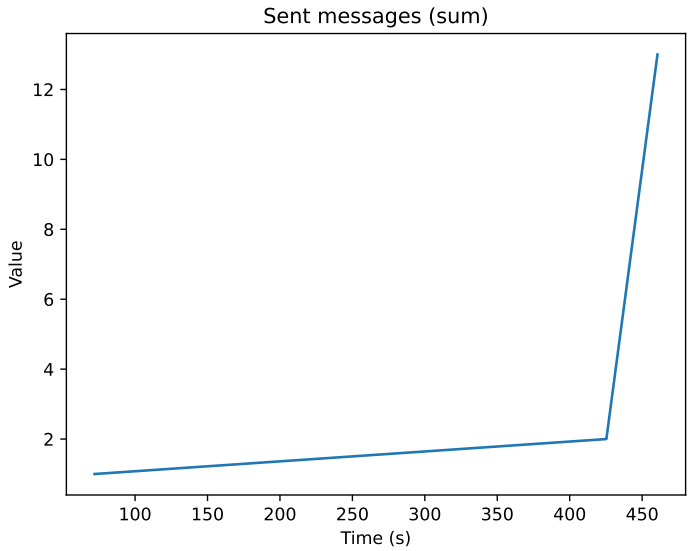


Figure 9.4. Single device - number of LoRa messages sent over time



and receiving is even more noticeable.

Focusing on the number of messages involved in the protocol, figure 9.9 shows a gap due to the number of retransmissions an end device uses to minimize the probability of lost messages at the gateways during GENERATE_COMMON_KEY, HELLO and FORWARD frames and to possible collisions that deny the sending or receiving of the response of the gateway if any.

Figure 9.5. Single device - LoRa messages receiving over time

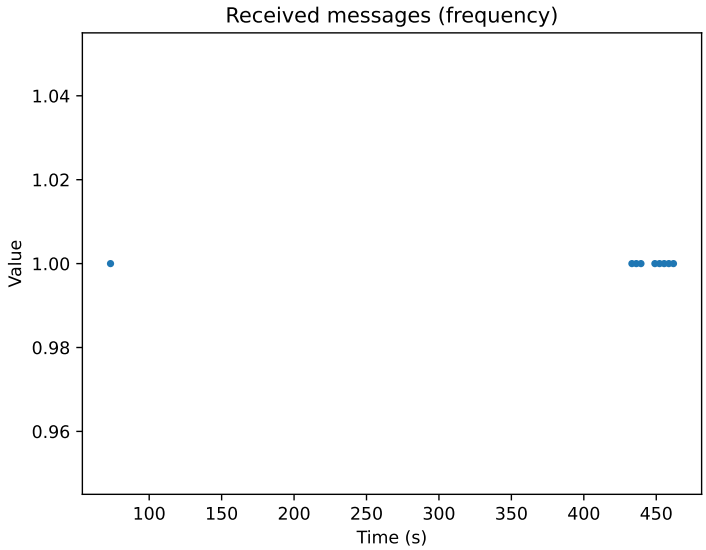
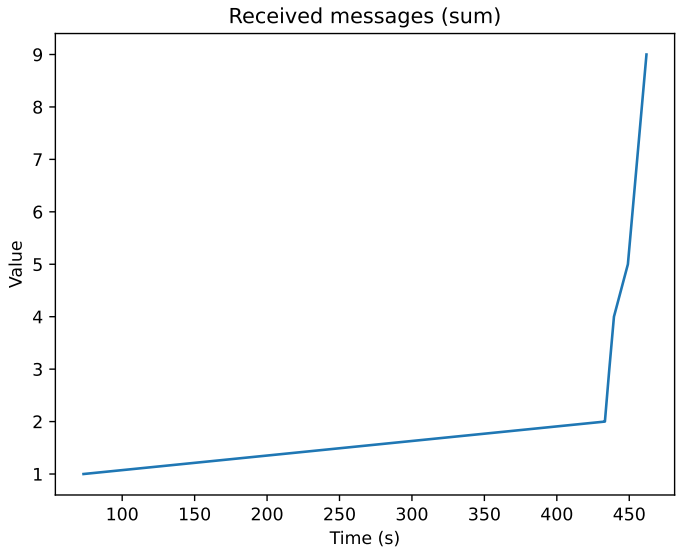


Figure 9.6. Single device - number of LoRa messages received over time



In this particular simulation run, as illustrated in figure 9.10, no message is retransmitted by the sampled end device due to a miss response by the gateway, therefore the gap is exclusively composed of expected retransmissions of a protocol run.

Analyzing one of the two gateways, figure 9.11 presents the distribution over time of the messages it sent and received on both LoRa and IP interfaces. Unlike

Figure 9.7. Single device - LoRa messages sending and receiving over time

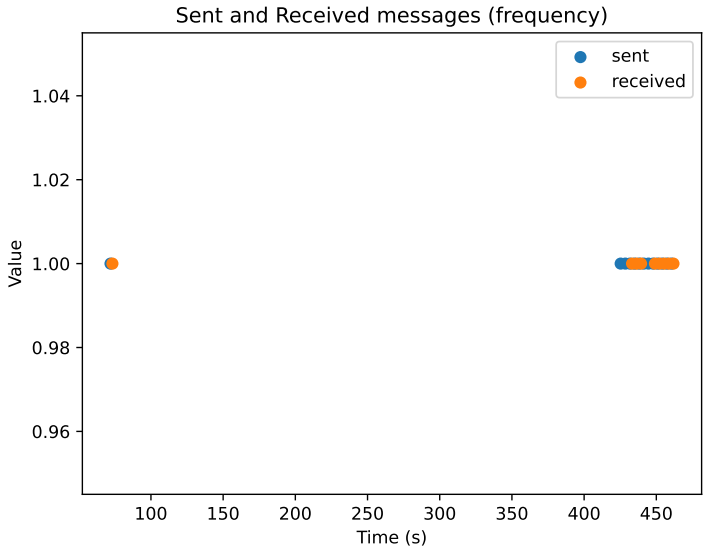
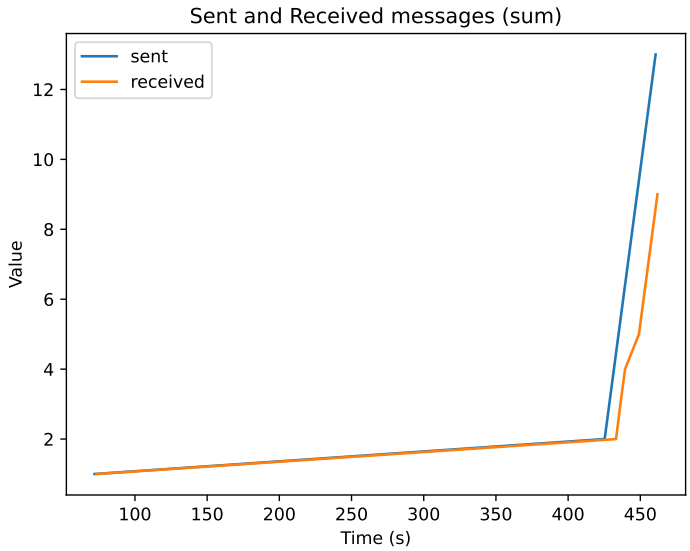


Figure 9.8. Single device - number of LoRa messages sent and received over time



end devices, the gateway value is not constant as there are both cases of ones and twos. This is because almost every time it receives a message from an end device, it notifies the network server besides replying to the sensor as described during implementation.

The chart in figure 9.12 better represents the exchange of messages where after an initial phase where sent messages dominate received due to gateway activation,

Figure 9.9. Single device - number of LoRa messages sent and received

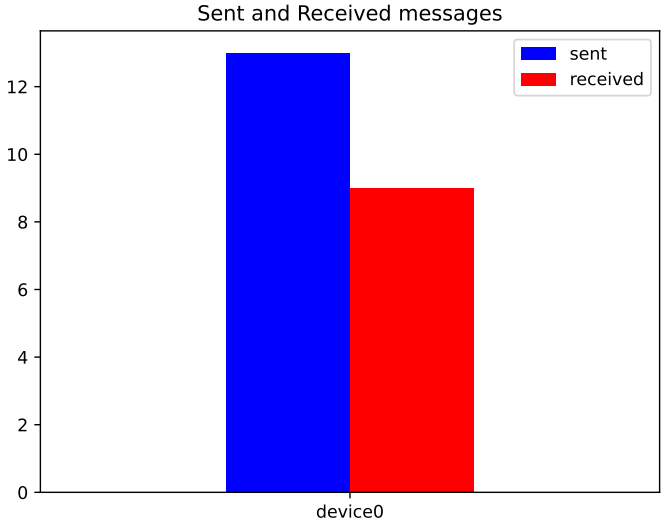
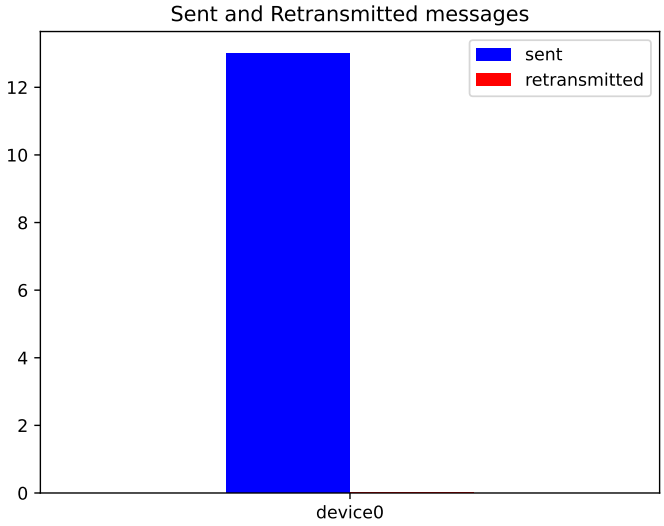


Figure 9.10. Single device - number of LoRa messages sent and retransmitted



then the trend is reversed as expected because the gateway should not respond to GENERATE_COMMON_KEY, HELLO and FORWARD messages in particular stages of the protocol when a certain condition is met (e.g. a STATS message has been acknowledged by the sensor).

Figure 9.13 helps to visualize such a gap but to understand the protocol execution even better, messages at the gateway must be differentiated based on the interface

Figure 9.11. Single gateway - messages sending and receiving over time

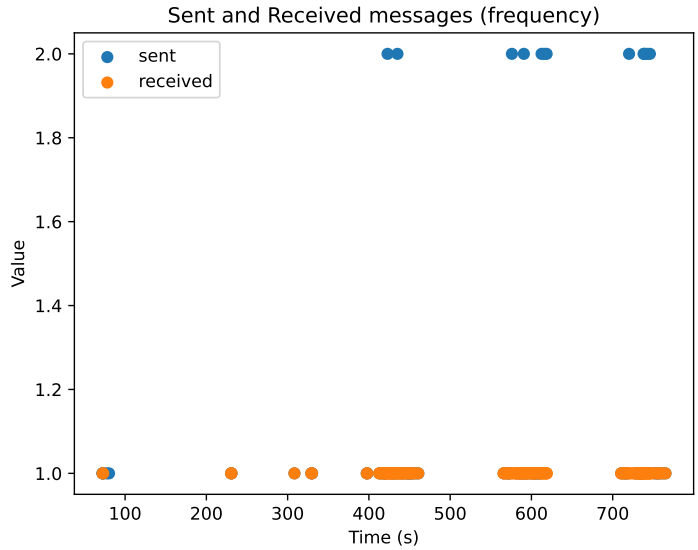
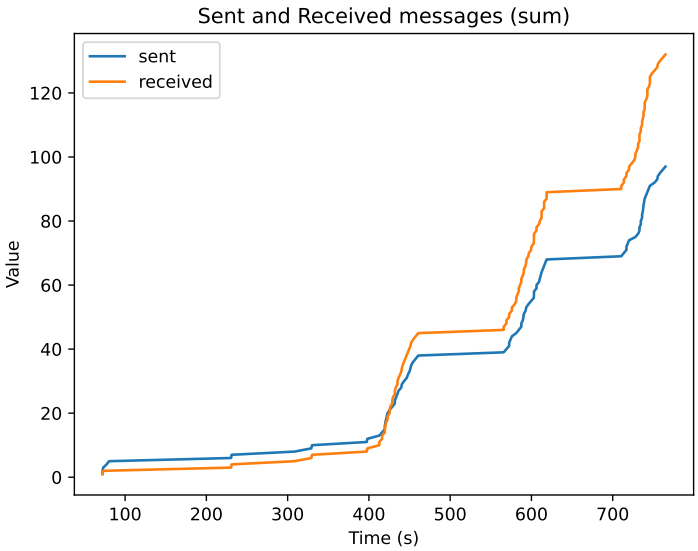


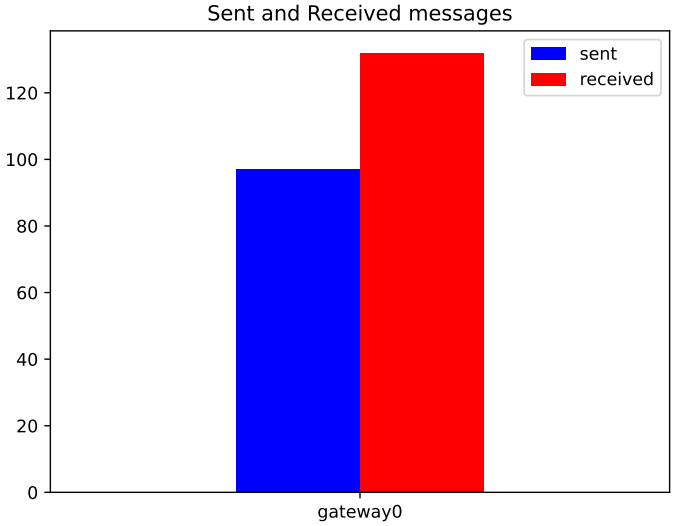
Figure 9.12. Single gateway - number of messages sent and received over time



where the frame is listened to or sent.

Such a split is represented in figures 9.14, 9.15 and 9.16. The former clearly shows an end device activation that takes place about 50 seconds after the simulation starts while the clusters of points denote the presence of at least three end devices executing the protocol. Overlapping points are better represented in the second figure through three peaks occurrences. Again it is important to notice the trend

Figure 9.13. Single gateway - number of messages sent and received



reversion of LoRa I/O where after an initial phase where the HELLO_GATEWAY is broadcasted, LoRa outputs are very limited with respect to other messages. This remarkable point is made evident in the last graph where LoRa outputs are less than half of LoRa inputs and it is very important because downlinks must be limited as much as possible. On the contrary, it is appreciable that the trend is the opposite over IP because the gap is expected to denote the absence of a response of the network server that may be very distant if located at the cloud in order to reduce the latency introduced by a possible cloud communication. Additionally, most of the messages are received over LoRa denoting a strong propensity towards an edge solution that is even more evident remembering that IP packets are also used to redirect a LoRa message to the right destination to minimize frame loss.

Lost messages are reported in figure 9.17 and as we can see a single message over about 130 is not received that represents about the 0.77% of the total messages "addressed" to the gateway.

Figure 9.18 let to visually appreciate such a value while figure 9.19 shows that such a loss regards the IP packet transmissions that has less impact on the performance of the protocol rather than a LoRa frame loss.

The complementary graphs are presented in figures 9.20 and 9.21 and frame receive percentage is therefore about 99.23% of all incoming packets.

With such an impressive frame delivery ratio is obvious that in this particular simulation run no interference affects LoRa transmission as shown in 9.22.

Moving the perspective from the protocol network traffic to the protocol load

Figure 9.14. Single gateway - LoRa and IP messages sending and receiving over time

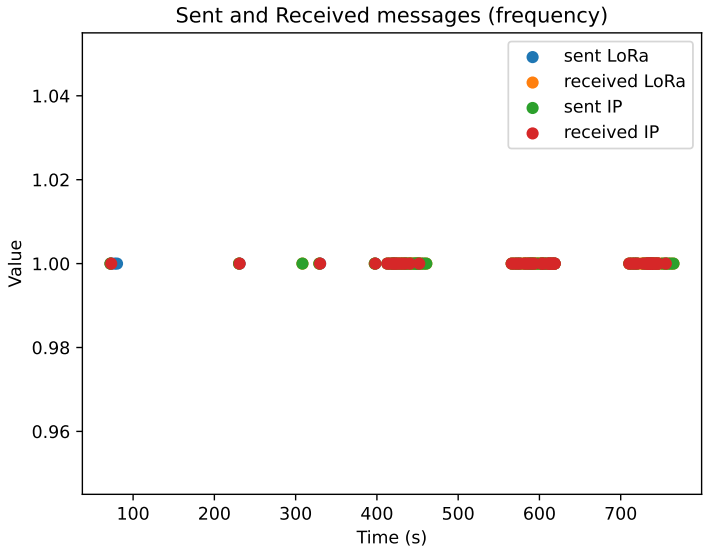
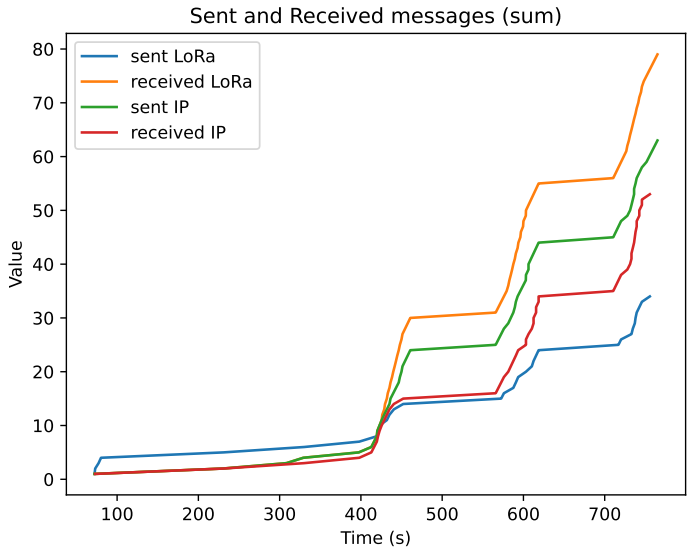


Figure 9.15. Single gateway - number of LoRa and IP messages sent and received over time



balancing, as mentioned above it is unbalanced to privilege the proximity of the source to the machine that will be in charge of processing collected data. Therefore the number of associated end devices is two as represented in figure 9.23 where the trend of pairings over time is plotted.

The last discussion regards the gateway resources which should reflect the number of associations via an increment of the workload. Although no stream processing

Figure 9.16. Single gateway - number of LoRa and IP messages sent and received

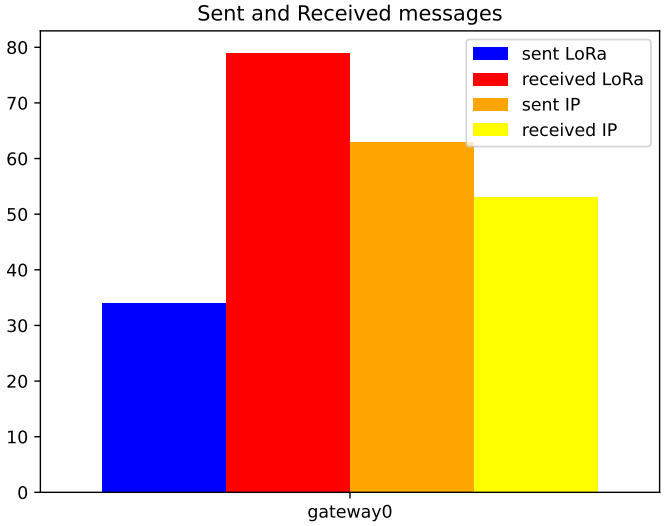
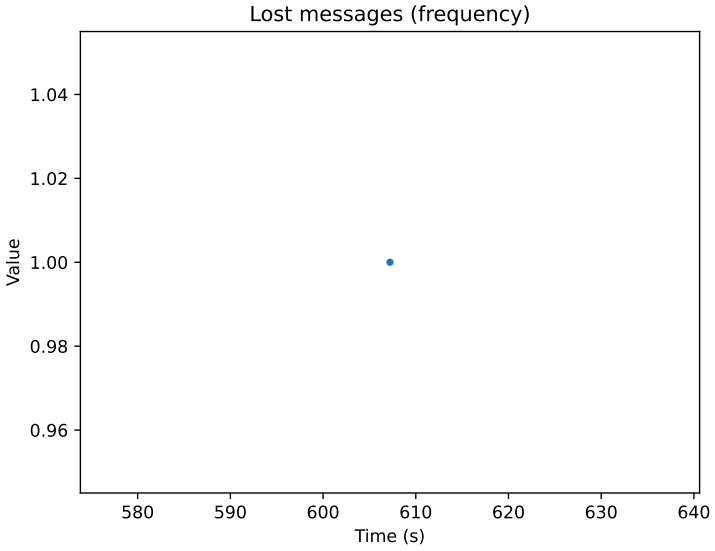


Figure 9.17. Single gateway - messages lost over time



task is simulated in the implementation, is legitimate to suppose that as soon as an association takes place the gateway prepares to receive data and therefore instantiate a default task with a negative impact on resources utilization. Figure 9.24 shows resources usage with measurements collected every second the simulation lasts.

The occupied storage space over time is faithfully reproduced in figure 9.25 where the trend of occupied percentage describes an increment of about $5.5 \cdot 10^{-6}\%$ which

Figure 9.18. Single gateway - number of messages in and lost

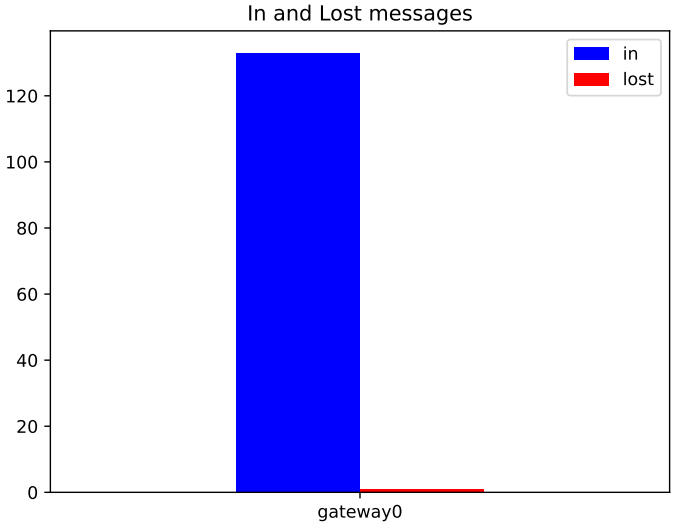
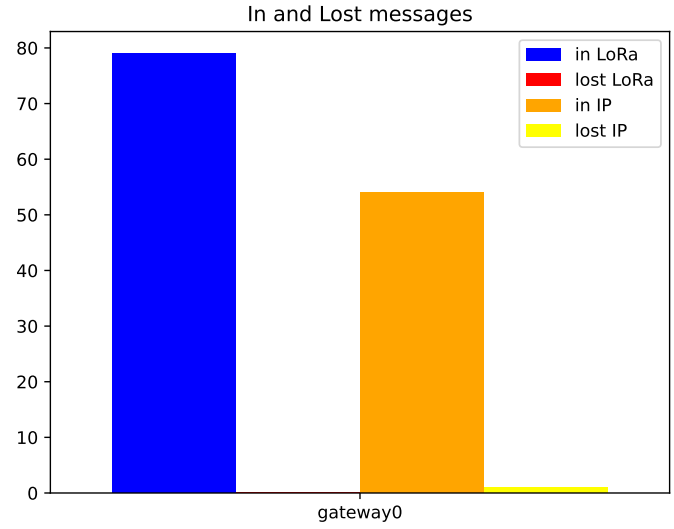


Figure 9.19. Single gateway - number of LoRa and IP messages in and lost



means the protocol has no negative impact on such a resource since the percentage is calculated over a total available storage size of 8GB as discussed in section 8. Therefore, with just 8GB of storage space, the gateway could potentially scale to a large-scale network composed of thousands of nodes where the bottleneck would be represented by the other resources affected by stream processing workload.

Figure 9.26 plots the network I/O in byte/s characterized by the 3 bursts analyzed

Figure 9.20. Single gateway - number of messages in and received over time

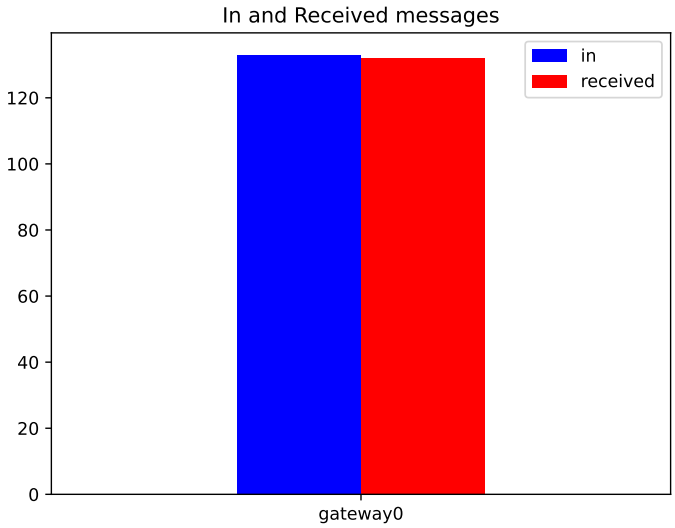
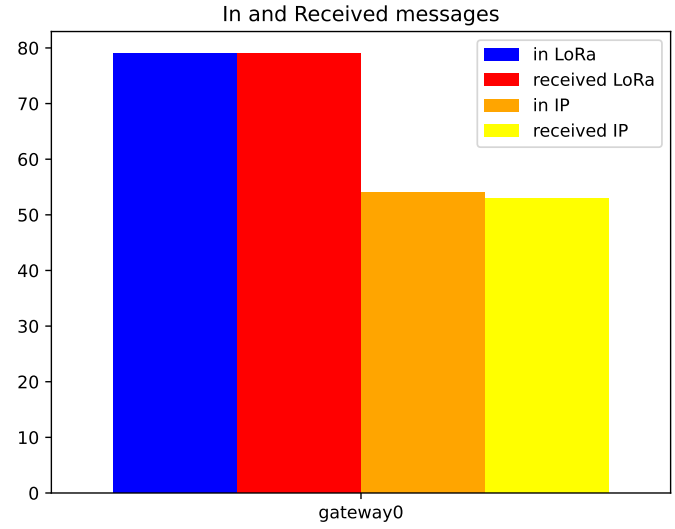


Figure 9.21. Single gateway - number of LoRa and IP messages in and received



above and the reactive nature of the gateway which exclusively reacts to listened frames. The number of bytes is considered by excess but values are acceptable as are recorded in the last second while a LoRa and IP transmission usually last a little portion of it (ms) and makes no distinction between transmissions on the two technologies.

A final notice about the distribution of the RSSIs collected at the gateway as

Figure 9.22. Single gateway - number of messages in and interferences

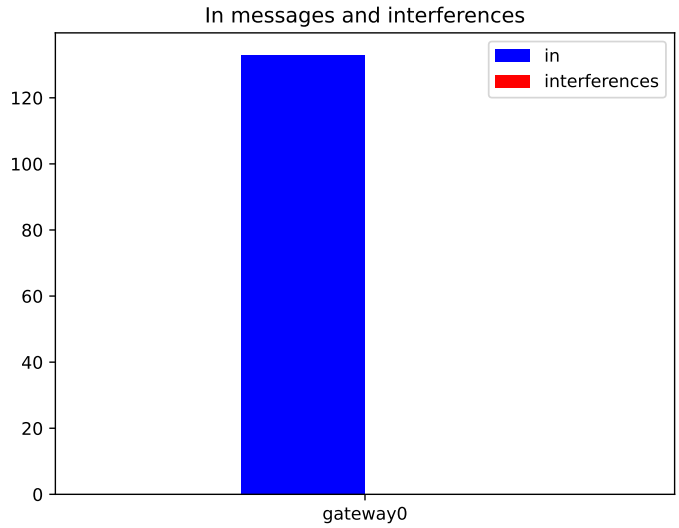
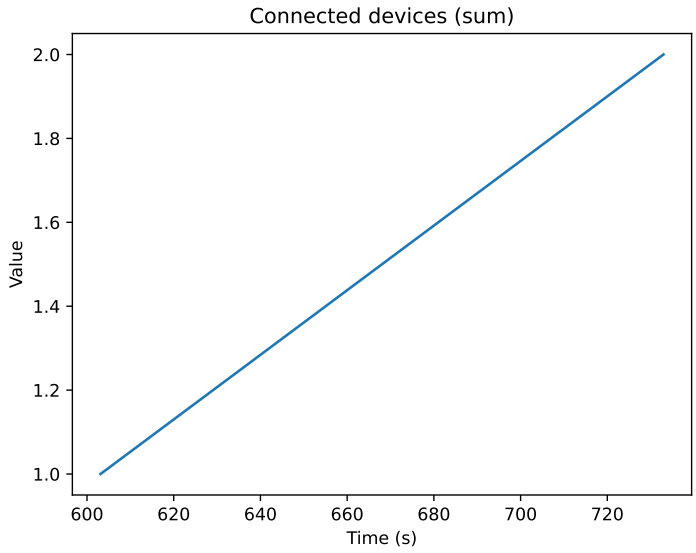


Figure 9.23. Single gateway - number of connected end devices over time



represented in figure 9.27. Most of LoRa signals have a poor strength as gateway sensitivity is defined in [-30, -120] [61] but depends on the location of devices and gateway and the environment where they are deployed. As most of the end devices are positioned at the border of the gateway radio range, RSSIs values reflect this situation and enforce to create associations with nearby sensors as in a larger network such signals can be heavily affected by interferences.

Figure 9.24. Single gateway - Resources over time

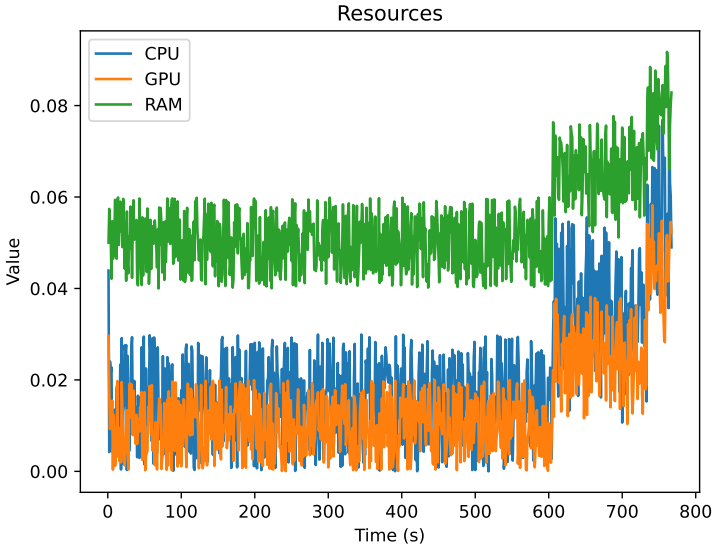
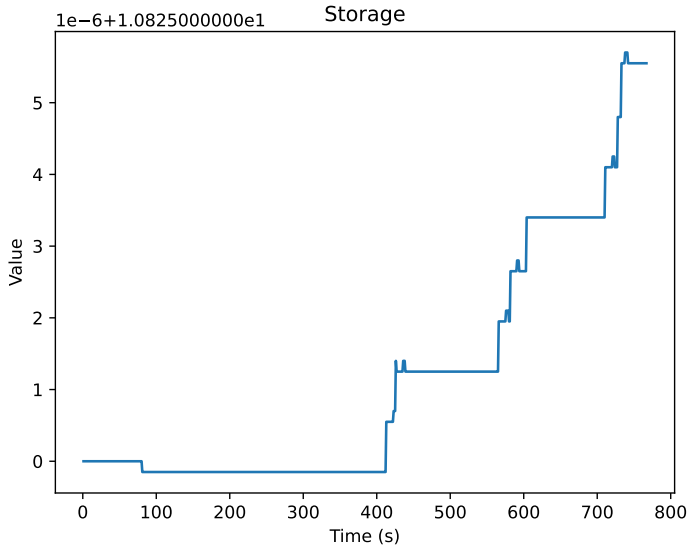


Figure 9.25. Single gateway - Storage over time



In the end, the discussion of the results shows that the protocol run is completed with success as all end devices connect to the gateways and message exchanges occur as expected. A more detailed analysis is not reported here but has been carried on for the entire duration of the protocol implementation to assert actions related to every single message occur as designed and if any possible retransmissions correctly happen (by disabling the corresponding receptions). As mentioned above, multiple

Figure 9.26. Single gateway - Network I/O over time

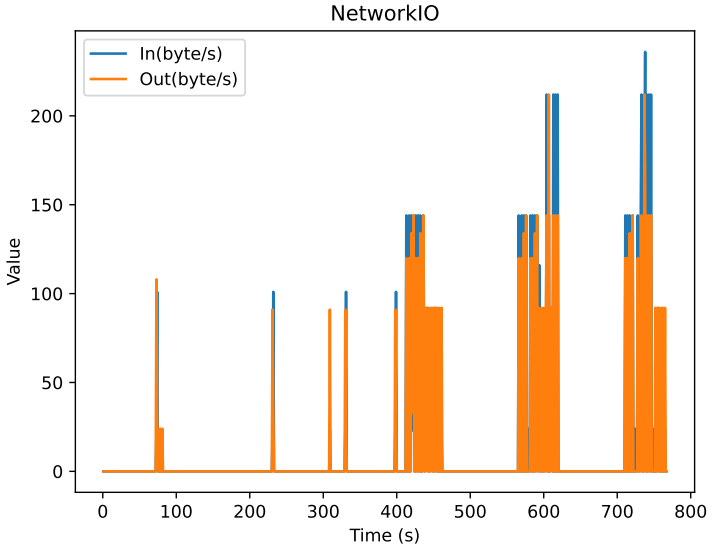
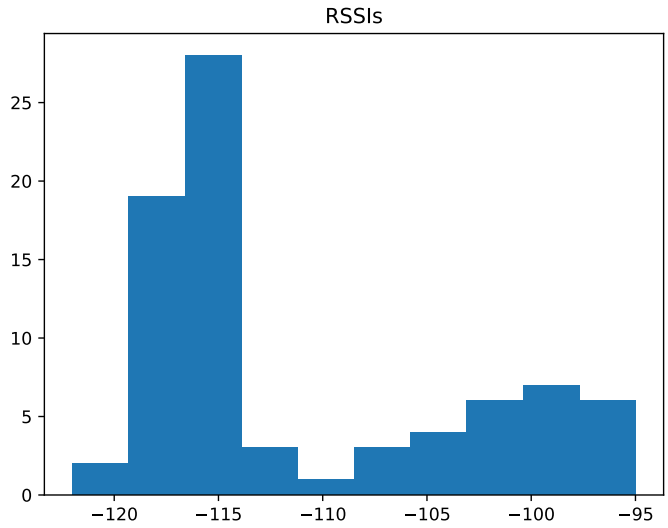


Figure 9.27. Single gateway - RSSIs distribution



runs of similar tests were used for evaluating correctness and therefore demonstrate the protocol completes with success every time.

To analyze the performances of the protocol, the continuous collection and sending of data by the sensors is removed in order to only evaluate the protocol messages and to automatically stop the execution when all end devices have completed their jobs avoiding introducing additional time delays at the end which would otherwise

be recorded. The performance analysis is carried on in terms of network utilization and network scalability (number of messages and collisions, retransmissions, gateway resource utilization, gateways load balancing and latency).

Figure 9.28 reproduces the real LoED deployment of 2019-03-01 based on collected RSSIs and spreading factors values at gateways and consists of 13 end devices and 2 gateways. Also in this case multiple gateways are positioned closer to the same gateway but since this time the focus is on performances, to have a better view, the results are obtained by the mean of 10 experiment runs.

Figure 9.28. Example of a real deployment - LoED 2019-03-01

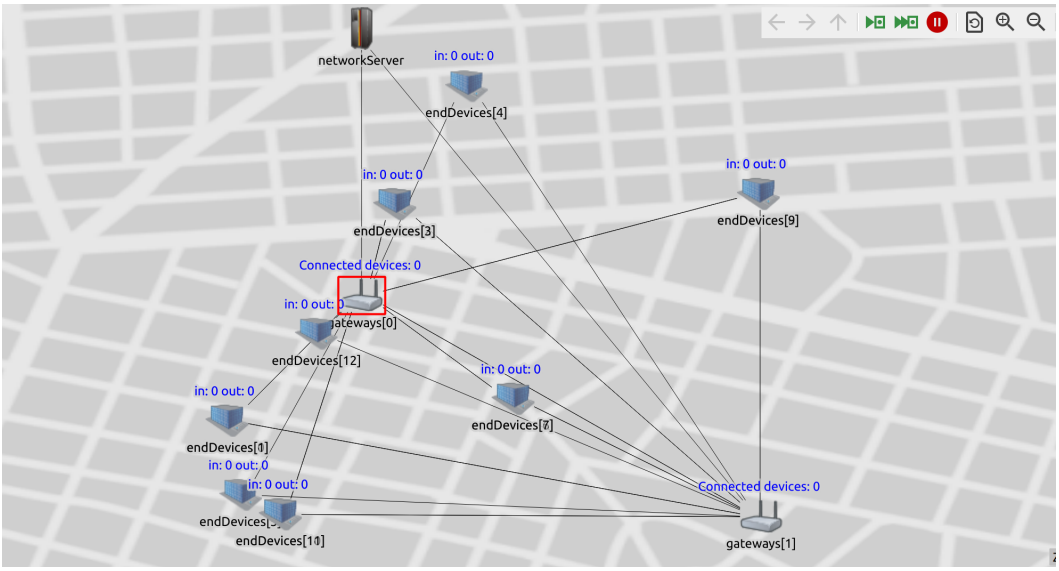


Figure 9.29 shows the distribution of messages of five end devices over time. In the first 400 seconds, all sample sensors send a Join Request as described in OTAA at different times and wake up after a random delay. After 500 seconds from the start of the simulation, 3 sample end devices perform the protocol so we can expect they may be affected by interferences although also device3 and device4 can since just a subset of all end devices are plotted in the graph.

However, figure 9.30 show no difference of trends compared to the previously analyzed scenario, therefore, the 5 devices should not be affected by interferences.

Moving to message receiving, the execution is consistent with LoRaWAN class A device behavior as presented in figure 9.31.

From 9.32, receiving are not influenced by interferences as well.

As expected by previous graphs, the number of sent and received messages by end devices is the same of the scenario with the half of end devices as shown in figure 9.33 and 9.34.

Unlike the previous stochastic scenario, here in figure 9.35 a message is retrans-

Figure 9.29. Multiple devices - LoRa messages sending over time

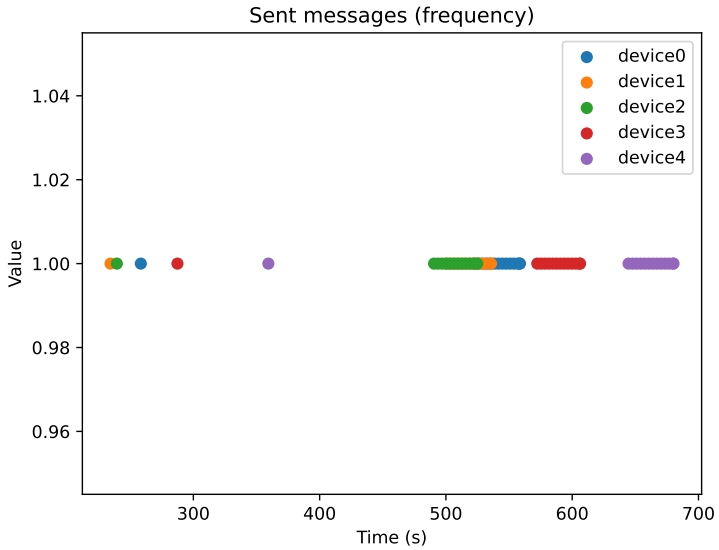
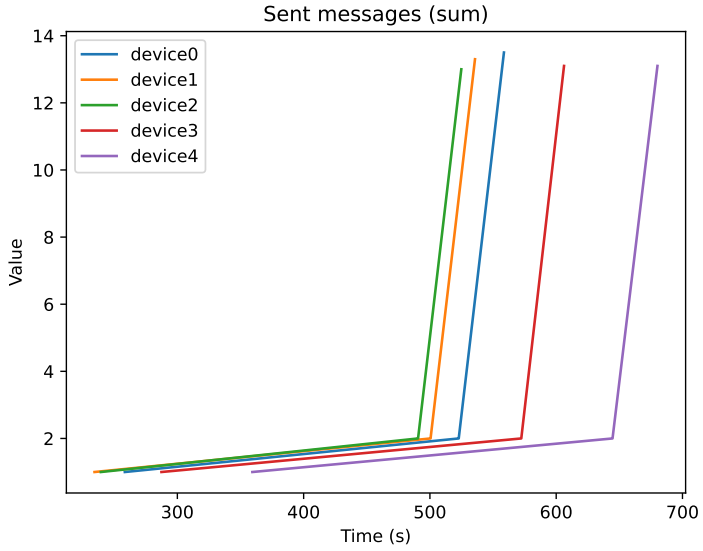


Figure 9.30. Multiple devices - number of LoRa messages sent over time



mitted once among 10 executions by an end device denoting an expected although the minimum increasing impact of interferences when the number of end devices is almost doubled from 6 to 13.

Figures 9.36 and 9.37 help to visualize the single retransmitted message over the 13 messages sent by the corresponding device in an effective way. Considering the five sample end devices, the single retransmission over the 65 total number of

Figure 9.31. Multiple devices - LoRa messages receiving over time

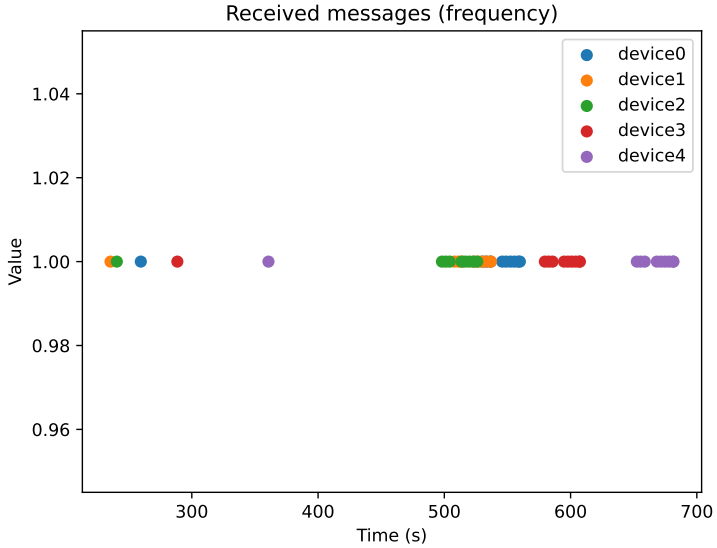
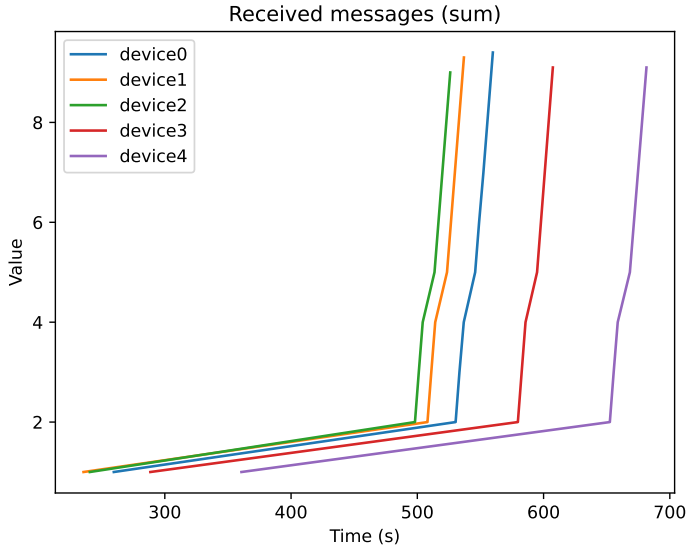


Figure 9.32. Multiple devices - number of LoRa messages received over time



messages sent corresponds to a percentage of retransmissions of about 1.54%.

Figure 9.38 confirm the goodness of the protocol in minimizing messages sent by gateways as they are much lower than those received.

This gap is explicit in figure 9.39 and it is also evident a difference of received messages between gateways although they are both in the end device ranges. This may suggest a greater data loss at gateway1 probably due to a greater distance from

Figure 9.33. Multiple devices - number of LoRa messages sent and received

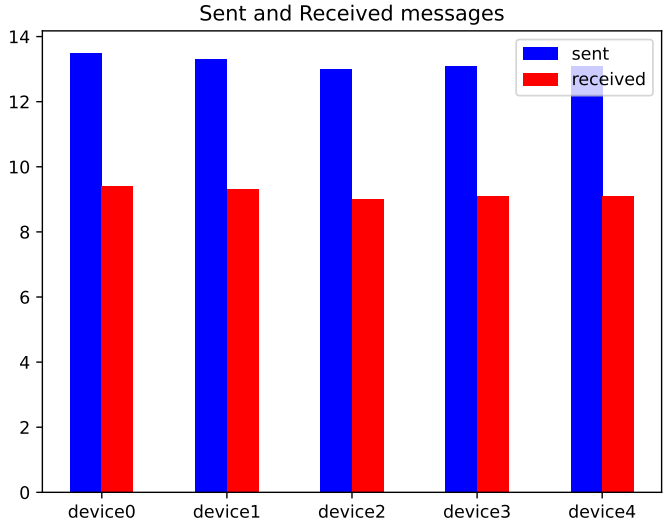


Figure 9.34. Multiple devices - number of LoRa messages sent and received

the end devices and corresponding low RSSIs.

A portion of this difference is certainly due to the slightly greater number of messages sent by gateway0 as shown by the chart with the minimum, maximum and mean values in figure 9.40.

Specifically, figure 9.41 reproducing messages exchanged over LoRa by the gateways demonstrate that there is a clear difference in messages sent by gateway0

Figure 9.35. Multiple devices - LoRa message retransmissions over time

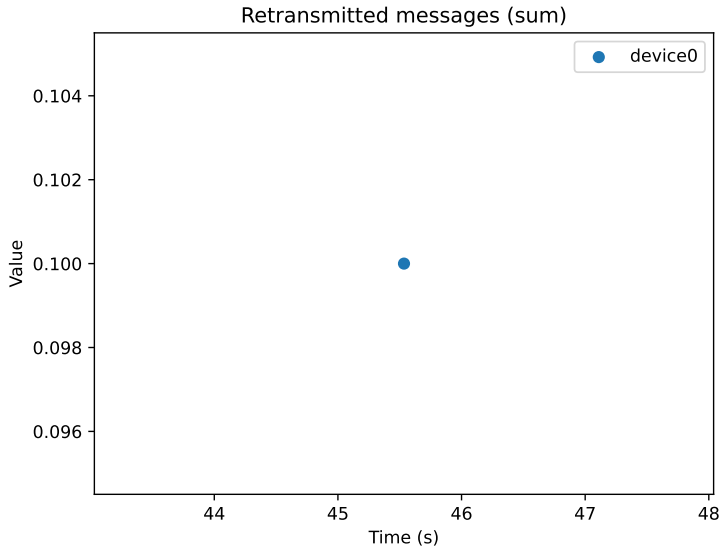
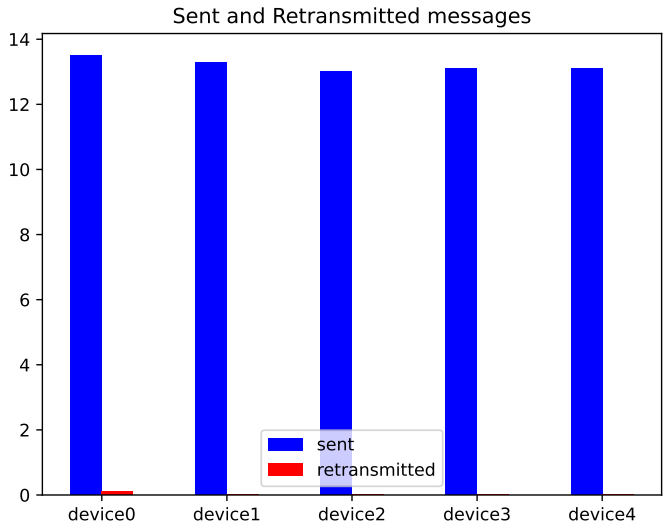


Figure 9.36. Multiple devices - number of LoRa messages sent and retransmitted



despite the same received messages.

Again, bar charts in figure 9.42 and 9.43 help to quantify such a difference which may be motivated by a difference in associations so that only the interested gateway responses over LoRa to minimize downlinks.

This suggestion is confirmed by figures 9.44, 9.45 and 9.46 where the trend is reversed in favor of gateway1 and the messages received by gateway0 reflects the

Figure 9.37. Multiple devices - number of LoRa messages sent and retransmitted

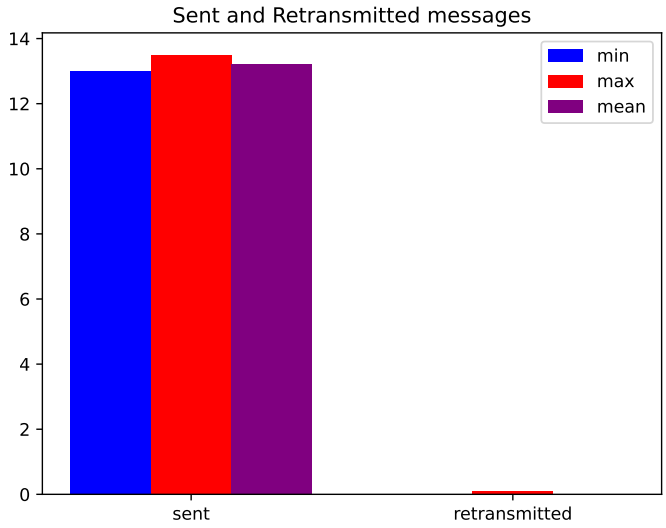
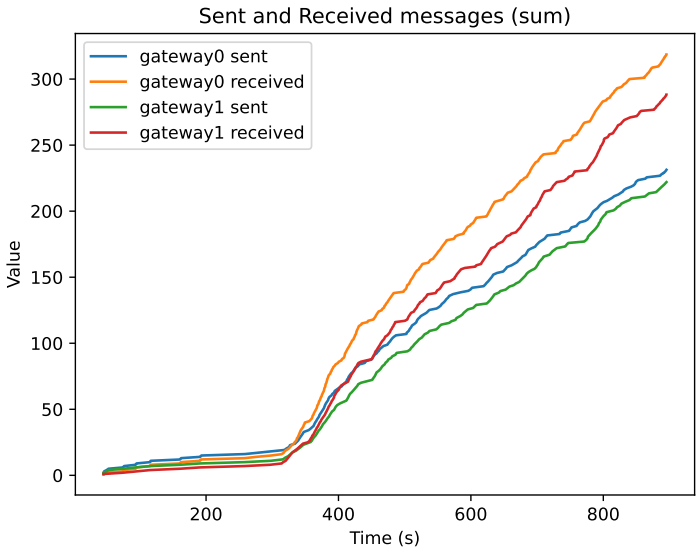


Figure 9.38. Multiple gateways - messages sending and receiving over time



gateway1 sending. This is motivated also by the big difference of listened packets over IP of the two gateways. This means that messages not intended for the current gateway are forwarded over IP to the proper destination without using LoRa, as expected by the protocol to minimize the number of downlinks and frame loss.

About packet loss, in figure 9.47 the frequency is greater than in the previous scenario but this is expected since the number of nodes is increased. A remarkable

Figure 9.39. Multiple gateways - number of messages sent and received

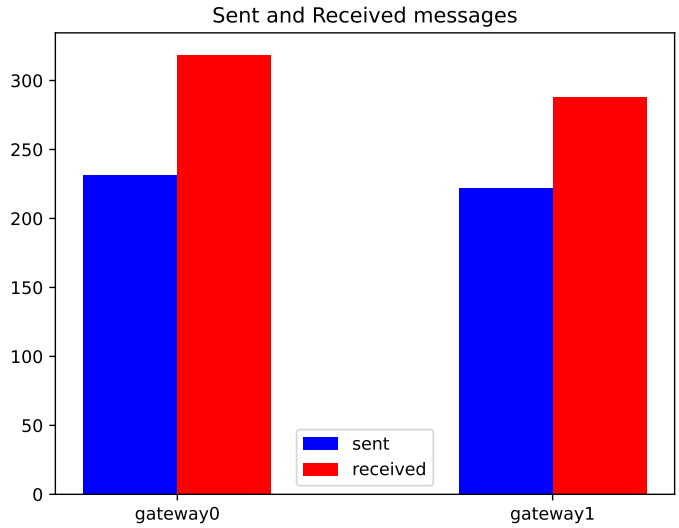


Figure 9.40. Multiple gateways - number of messages sent and received

point is that gateway0 is affected in the first 200 seconds of the simulation (probably over LoRa) while the peer starting from the second 450 (probably over IP).

The mean number of lost messages over time is presented in figure 9.48.

Anyway, the greater number of lost messages result in a slightly greater percentage of frame loss ratio for gateway0 with respect to the previous scenario having the half of nodes, as represented in figures 9.49 and 9.50 where the percentage of gateway0 is

Figure 9.41. Multiple gateways - LoRa messages sending and receiving over time

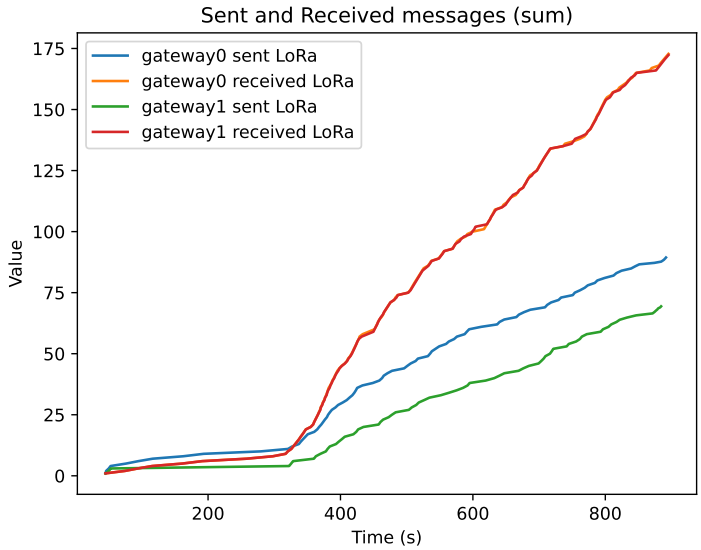
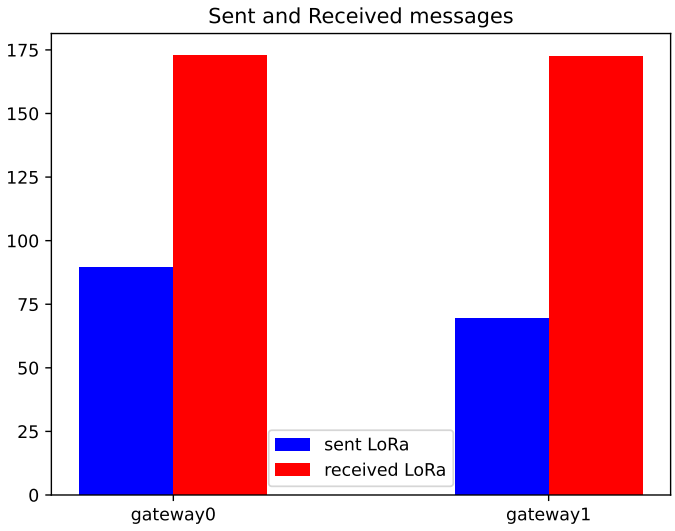


Figure 9.42. Multiple gateways - number of LoRa messages sent and received



about $3/340 = 0.0088 = 0.88\%$. On the contrary, the message loss ratio of gateway1 is about $1/310 = 0.0032 = 0.32\%$ and their mean is about $4/650 = 0.0062 = 0.62\%$ that is less than in the previous scenario. However, in this case, we are considering both gateways while in the previous the focus was on a single gateway and sensor.

As usual, to better understand why losses occur, the messages are differentiated according to the gateway interface where the frame is listened to. Figures 9.51 and

Figure 9.43. Multiple gateways - number of LoRa messages sent and received

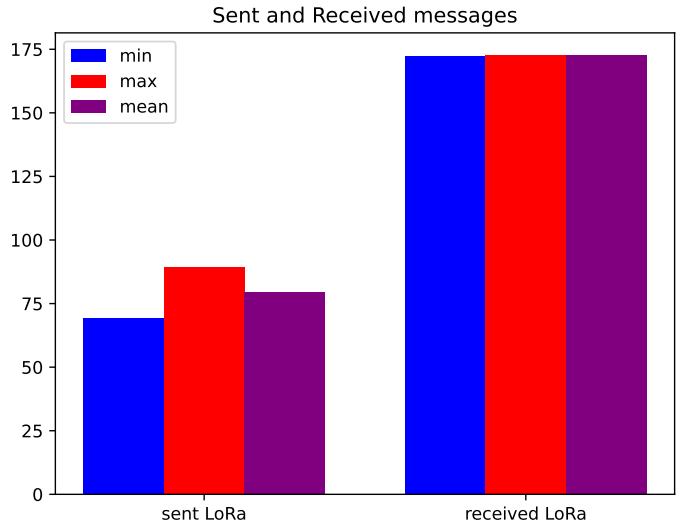
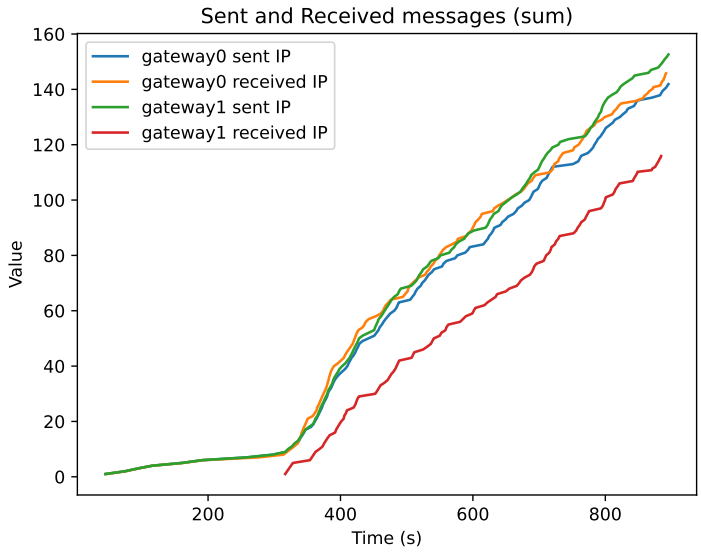


Figure 9.44. Multiple gateways - IP messages sending and receiving over time



9.52 show that LoRa frame loss percentage is almost zero as it is barely noticeable at gateway1.

Unlike LoRa, IP losses are slightly more visible in figures 9.53 and 9.54 confirming again that most of losses occur over IP because more susceptible to LoRa interference than the opposite but less problematic than LoRa losses.

Next two figures 9.55 and 9.56 show no LoRa interference takes place and

Figure 9.45. Multiple gateways - number of IP messages sent and received

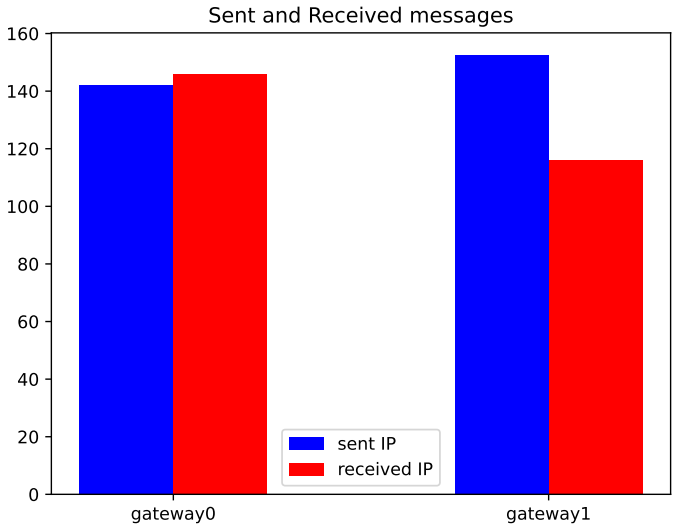
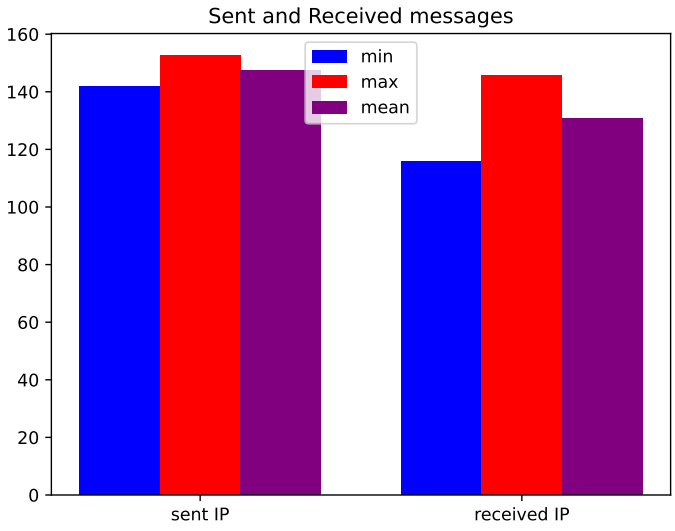


Figure 9.46. Multiple gateways - number of IP messages sent and received



therefore the few frame losses detected over LoRa were caused by background noise that the simulator reproduces.

Analyzing the load balancing of the protocol must be remembered that the protocol aims to associate end devices to gateways based on proximity and workload. As a small network is reproduced about LoED 2019-03-01, an unbalanced load is expected. Results of figures 9.57, 9.58 and 9.59 instead present a balanced situation of

Figure 9.47. Multiple gateways - messages lost over time

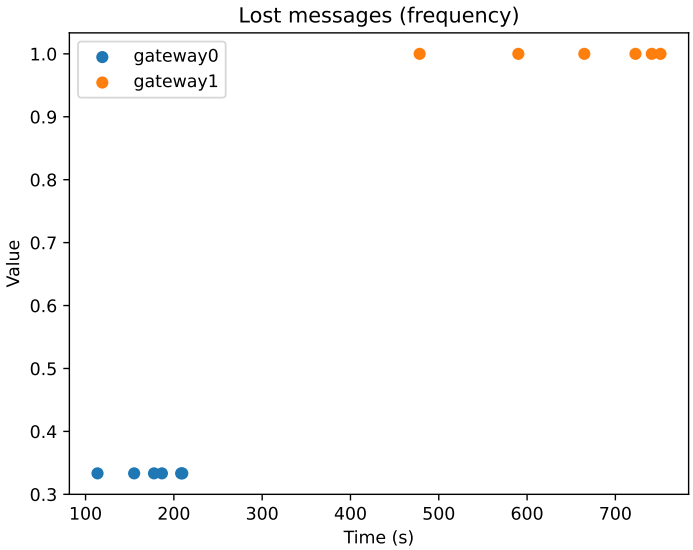
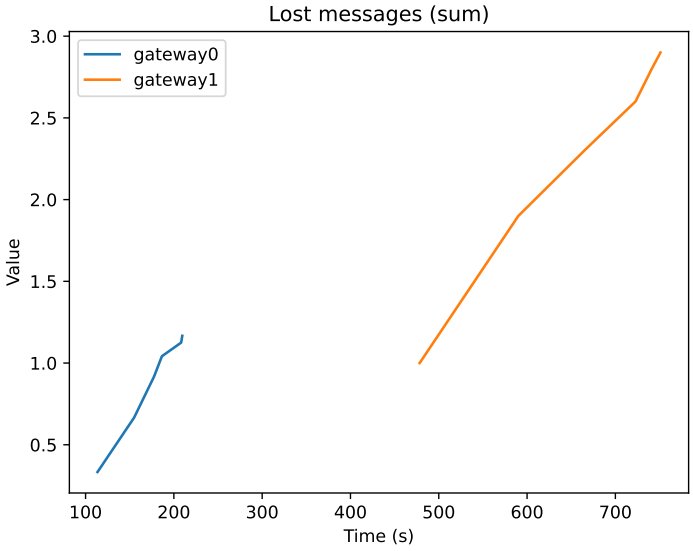


Figure 9.48. Multiple gateways - number of messages lost over time



7 devices connected to gateway0 and 6 to gateway1. This may look quite unexpected but to discover the reason we need to look at RSSIs.

As expected gateway0 receives better signals than gateway1, so, the motivation of balanced load is caused by resources workload. In the section about implementation (8), has been stated that the score is computed assigning the same weights to all factors that participate in the sum. Therefore, to privilege the RSSI, different weights

Figure 9.49. Multiple gateways - number of In and lost messages

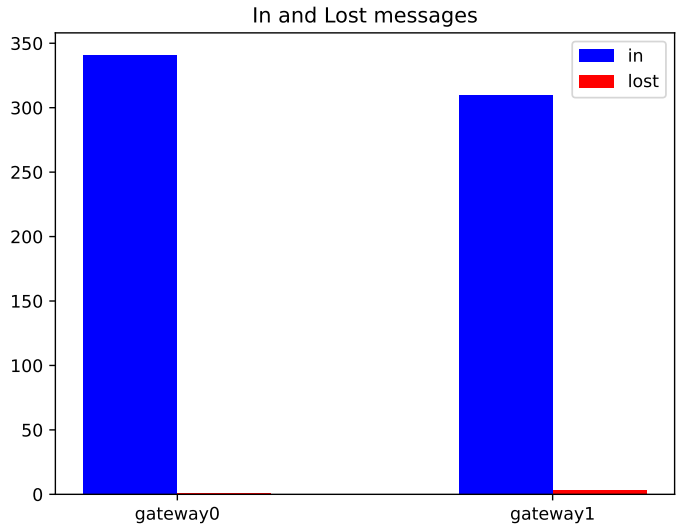
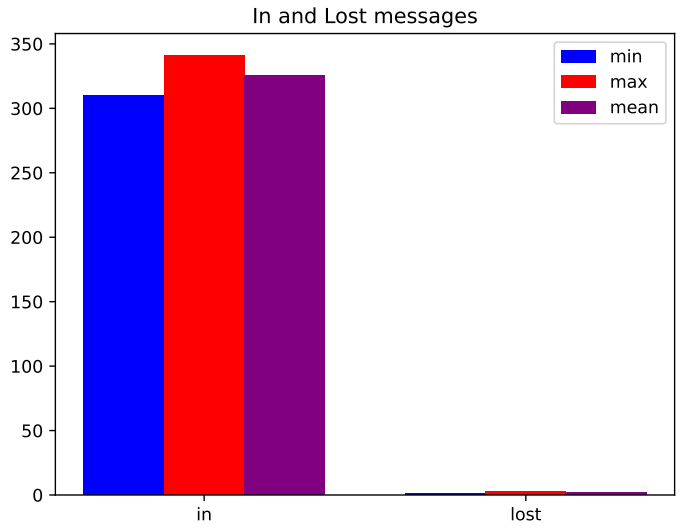


Figure 9.50. Multiple gateways - number of In and lost messages



must be applied. However, an RSSI of about -113 dBm is acceptable for LoRa.

Looking at resources usage in figure 9.61, the CPU, GPU and RAM percentages of gateway0 grows faster than those of gateway1 and for this reason, the load is balanced because these consumptions reduce the goodness of RSSI in a sum where all parameters have the same weights.

The bad trend of gateway0 in resources utilization is confirmed in the occupied

Figure 9.51. Multiple gateways - number of in and LoRa messages lost

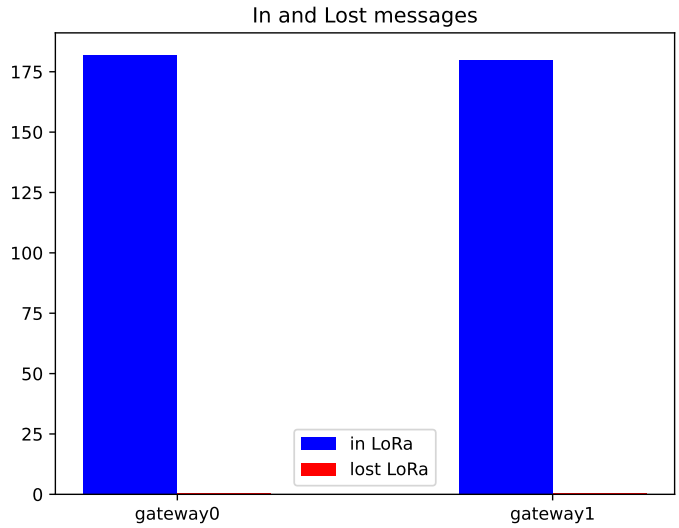
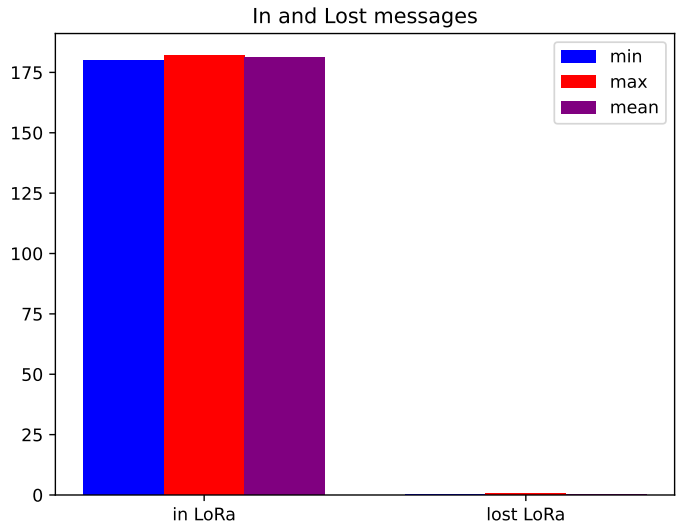


Figure 9.52. Multiple gateways - number of in and LoRa messages lost



storage graph in figure 9.62 while network statistics are very similar throughout the simulation as described in figure 9.63. Again, the storage utilization is minimal considering a storage size of 8GB.

Results show that also in a real deployment the protocol completes with success and the number of exchanged messages is very low as only 13 messages are necessary where 4 represent retransmissions of GENERATE_COMMON_KEY and HELLO

Figure 9.53. Multiple gateways - number of in and IP messages lost

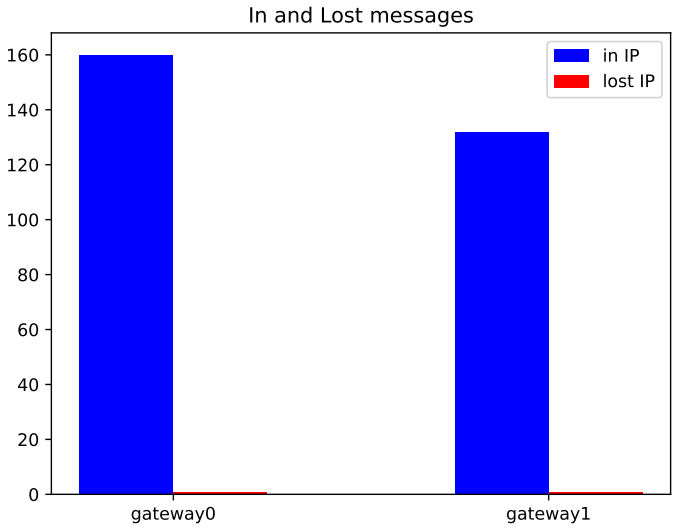
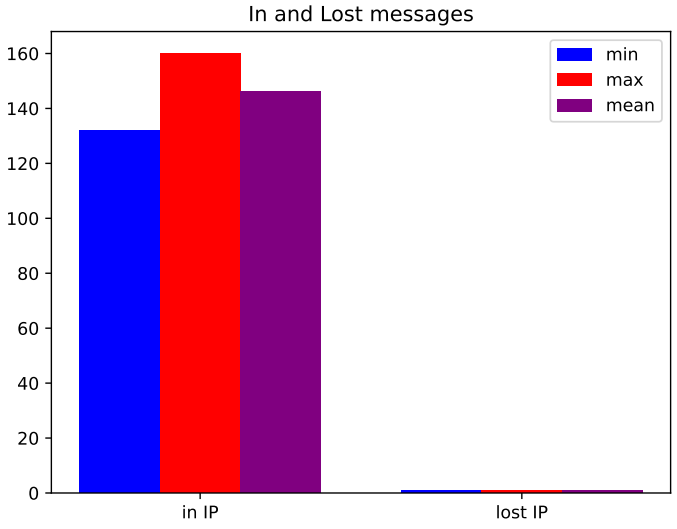


Figure 9.54. Multiple gateways - number of in and IP messages lost



messages to minimize frame loss. This means the protocol finishes with no delay because random wake-ups reduce the probability of collisions.

Figure 9.55. Multiple gateways - number of In and interferences messages over time

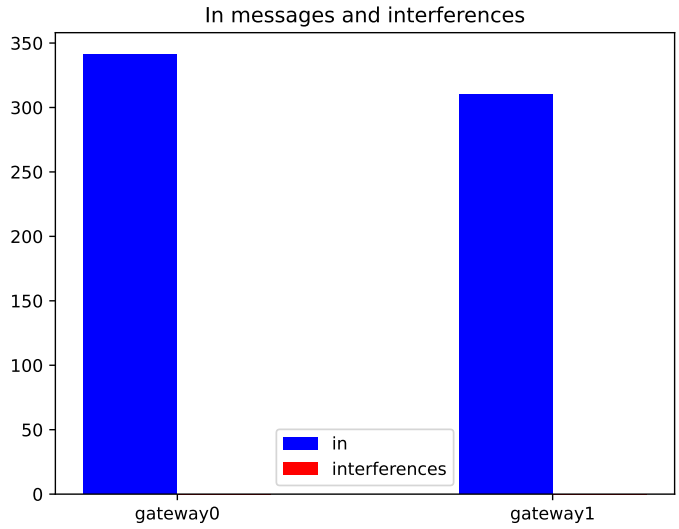


Figure 9.56. Multiple gateways - number of In and interferences messages over time

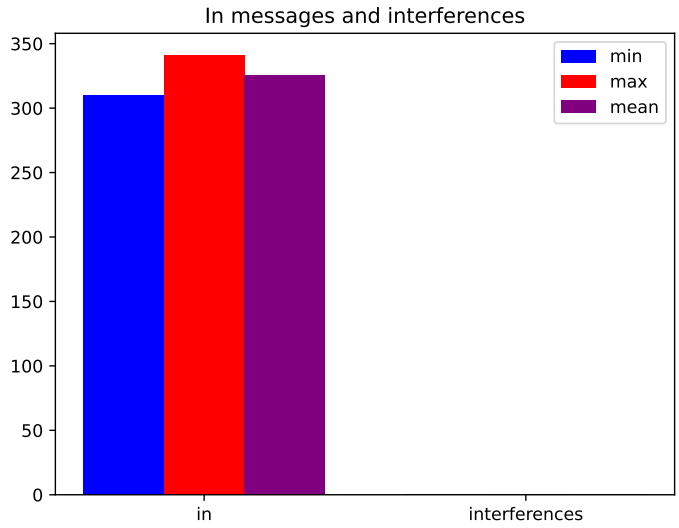


Figure 9.57. Multiple gateways - number of connected end devices over time

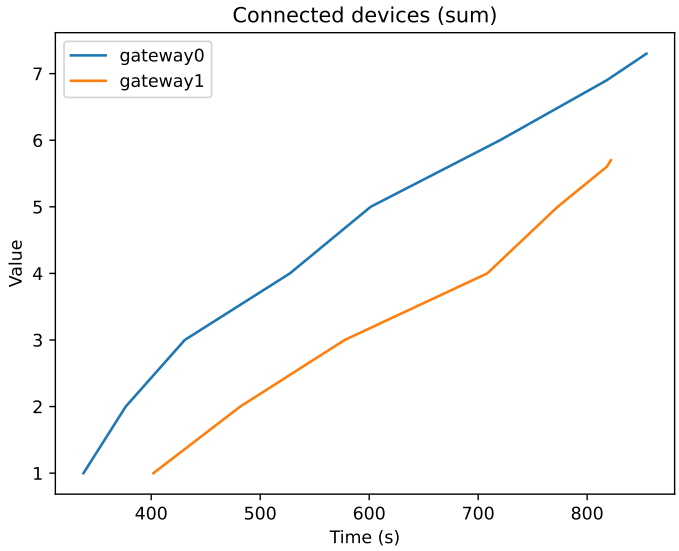


Figure 9.58. Multiple gateways - number of connected end devices

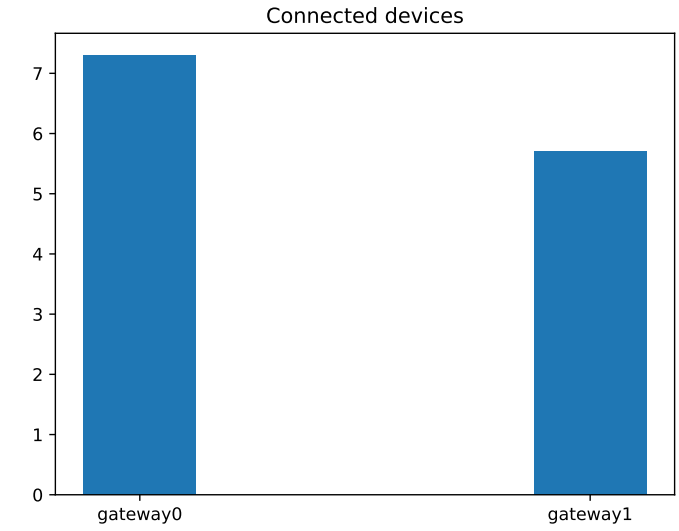


Figure 9.59. Multiple gateways - number of connected end devices over time

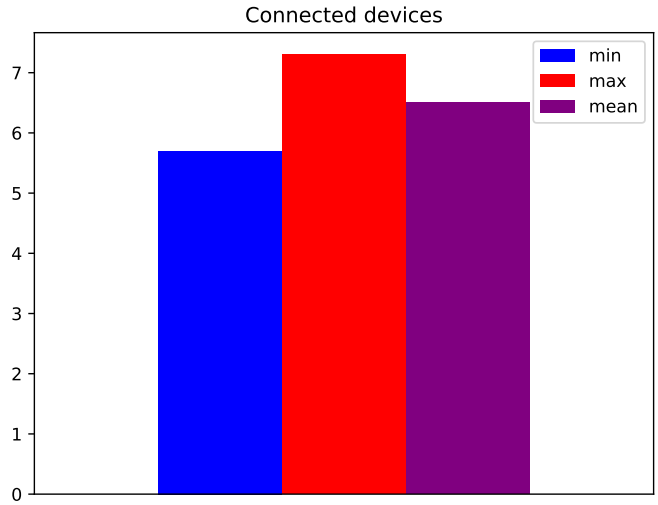


Figure 9.60. Multiple gateways - RSSIs distribution

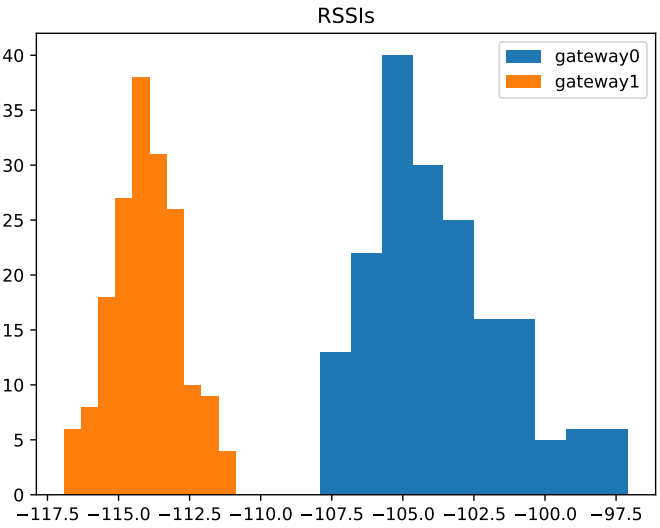


Figure 9.61. Multiple gateways - Resources over time

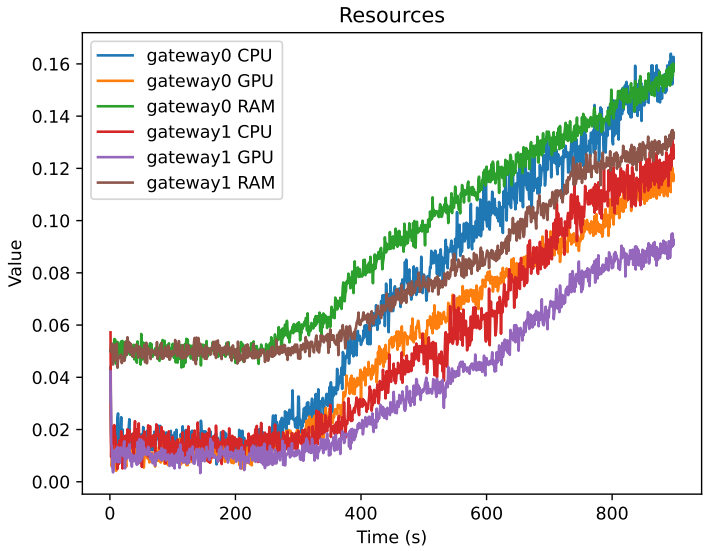


Figure 9.62. Multiple gateways - Storage over time

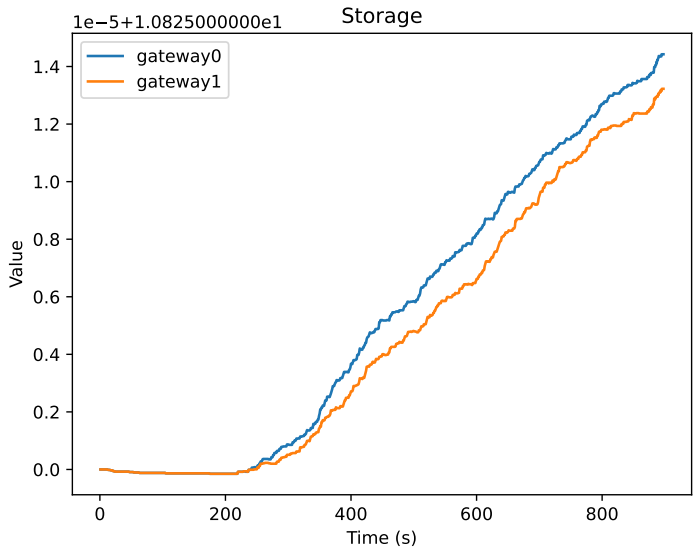
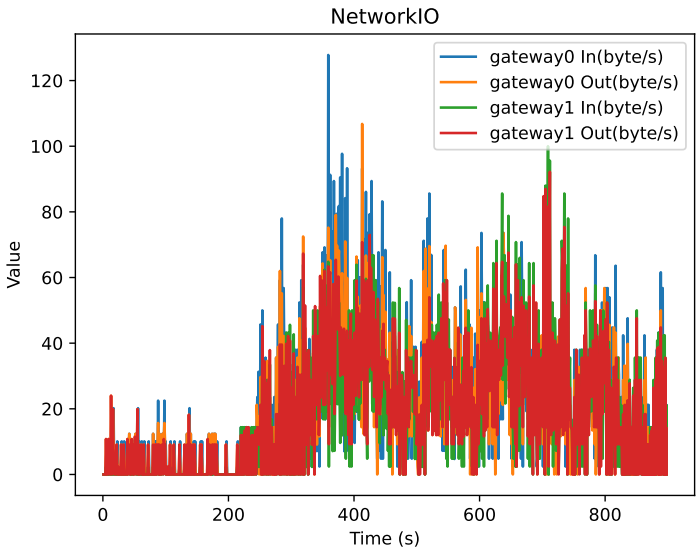


Figure 9.63. Multiple gateways - Network I/O over time



Chapter 10

Client connection to RP

Once an end device (i.e. water metering sensor) is connected to a LoRa gateway (RP), it can begin sending data stream periodically. If a client (i.e. user) is interested in values produced by a specific sensor, it usually needs to access the cloud where large amounts of data are stored because application architectures are commonly cloud-based.
However, this is fine for retrieving data in the past, for example specifying a time interval (last week, last month, last year, ...), and then performing analytics on it but, on the other hand, querying the cloud is not the optimal solution for accessing real-time data because I can take advantage of location proximity of the source and destination to reduce latency and network traffic (e.g. show real-time water consumption in the building, give direct feedback to actuators and so on).
To realize a framework able to give users the possibility to obtain data at the edge of the network, the idea is to directly connect to the RP to which the sensor is associated to read and/or process collected data. To achieve this, I have to keep in mind that the binding between a sensor and a RP is stored in the LoRa network server as well as in the RP itself (including the sensor's profile).
The LoRa network server is the point of connection between the LoRaWAN network and the Internet, so represents the entry point for an external user: without accessing the cloud, it may directly access the network server to retrieve the LoRa gateway to which the sensor is connected to.

In a smart city scenario, an extended geographical area has to be split into multiple LoRaWAN networks (e.g. north, east, south, west) because the LoRa range can be up to five kilometers in urban areas [40]. In such a case, there exist multiple entry points (one for each network) and since the user may not know where the sensor is located but the device EUI, has to contact all LoRa network servers to retrieve the associated RP.

Instead of dealing with multiple IP addresses, a single multicast IPv6 address can be employed obtaining the same simplicity of a single cloud entry point but a more efficient solution due to the user's proximity to the servers.

Hence, the user can obtain the RP to which a sensor is paired by sending a specific USER-CONNECTION packet over TCP/IP (including the port in the header and the device EUI in the payload) to the LoRa network servers which will scan their mapping tables looking for a matching entry (device EUI, RP's IP address). Only the server who has the wanted device in its LoRaWAN network will reply positively (HTTP 200) including the RP's IP address.

To evaluate performances in contacting LoRa network servers, a test can be carried out between multicast IPv6 and sequential unicast IPv4/IPv6.
While the multicast version spreads a single packet intended for all network servers of the application to approach all of them simultaneously, the unicast version sequentially scans the list of network server IP addresses and contacts the next server in the list only if the previous one has responded negatively.
In the worst case, the unicast version has to handle n one-to-one communications, with a delay between each other due to the RTT + the time to build and submit the next packet. The latency is clearly higher than building a single multicast packet and spreading it once to receive responses.
In a general case, the advantage of the sequential version is that a subset of network servers can be avoided to be reached (saving CPU for useless table scans) because the device EUI entry has already been found.
However, this CPU advantage has to be compared to network delays due to the creation of multiple single communications and to the performances of the multicast version via experiments to be demonstrated as a real benefit. Since from a theoretical point of view the worst case prevails, the multicast version (awaiting tests) is preferable.

10.1 Stream processing engine

So, when the user's client has retrieved the IP address of the LoRa gateway paired with the desired end device, to read and/or process data the sensor is collecting, the user needs to submit a QUERY-REQUEST message specifying a query and the end device EUI to the LoRa gateway. The gateway on its behalf needs to run a program able to interpret and execute the received task.
As the sensors continuously send data at regular time intervals, stream processing represents a suitable solution because efficiently performs real-time processing as

soon as new data arrives from the source to the processing node.
In recent years, many frameworks have been developed implementing this computer programming paradigm such as the famous Apache Flink, Apache Kafka, Apache Spark and Apache Storm. Unlike the others mentioned, Spark is not a native streaming engine but a micro-batching engine that means collecting data and processing it together every few seconds, thus introducing small delays. So, the stream processing engine listens on a specific TCP/IP port for new task requests and receives in input a user query denoting the operations to apply to a specific data stream, builds a corresponding dataflow graph of operations and executes them on the related data stream.
As usual, each framework has its own advantages and disadvantages and the choice has to consider the use case [42, 39]. In our particular scenario and solution, since the data processing is performed on LoRa gateways and therefore the stream processing engine must not run on the cloud but on every LoRa gateway at the edge, the constrained resources of the edge nodes take on considerable importance in making a decision. In fact, gateways have to afford such a complex software that usually runs over a Java Virtual Machine (JVM) and is particularly CPU intensive but gateways may not even be able to implement the JVM.
Taking a look at LoRa gateways technical specifications, many manufacturers produce devices equipped with a processor characterized by a MHz frequency clock, a RAM size in MB and a ROM size in KB and even the Cisco gateway (one of the best LoRa gateways at the time of writing) is provided with a 1.33 GHz single-core CPU, RAM size of 1 GB and ROM storage of 4 GB.
So, it is probable that such hardware does not satisfy stream processing engine requirements and to remedy this shortcoming, an alternative is to build our own LoRa gateways starting from more powerful boards (e.g. Raspberry PI 4 is equipped with a 1.5 GHz quad-core processor and 2, 4 or 8 GB RAM) [30].

Typically, stream processing operations are grouped into four categories:

- Single record operations: process a single event in the input
- Multiple records operations: process multiple events in input through windows
- Join operation: merge multiple data streams into one
- Split operation: separate a data stream into multiple ones

Common single record operations are Filter (removing undesirable data) and Map (transforming data) while common multiple records operations are analytics such as Count and Average and apply to data collected in a specified window. A window is

a memory to look back at recent data efficiently.
The join operation is a challenging operation for the streaming framework because multiple data streams can probably have different timestamps and therefore the engine needs to align them. This is not surprising since time is a well-known problem that has been studied by the distributed systems community for decades.
From the time of events, it is a natural consequence to derive an ordering, thus stream processing operations also include the possibility to detect patterns by correlating events based on timestamps and the "happened-before" relation. This means that anomaly and fraud detections can be easily implemented without developing complex machine learning models.
However, if the application needs sophisticated predictions, then machine learning has to be introduced. There exist two main approaches:

- Develop and learn a model on the cloud, based on time series in a classical way and then deploy it to the edge
- Develop and learn a model directly at the edge based on streaming machine learning

In the end, I decided to use Apache Flink as it is one of the most innovative stream processing engines with low latency and high throughput [46].

10.2 Scale-out

Nevertheless, when the user's client retrieves the IP address of the LoRa gateway paired with the desired end device, it may be unable to instantiate a stream processing task on such RP because its available resources could not be sufficient for executing the query. To reduce network traffic (i.e. number of messages), the RP itself can decide if the currently available resources are sufficient for the task and then reply to the client positively (including a UID generated on the fly to be used as a reference for the query) or not (similarly to the pairing algorithm when a PAIRING request is sent to a RP). Moreover, this functionality could be directly made available by the streaming engine itself without requiring additional effort to implement it.
Thus, if the RP is heavy-loaded and cannot meet the user request, a nearby RP can be selected to allow the user to still process the sensor's data through a scale-out algorithm.
In the proposed framework, a scale-out consists in adding a RP to the set of RPs to which the sensor is connected, so that the sensor's data are also read by this

additional RP. To accomplish it, the framework can reuse the pairing algorithm and associate the end device with an additional RP. However, in LoRaWAN an end device sends a message in broadcast once in each uplink window and in the proposed framework this is encrypted with a session key exclusively shared with the first paired RP. As a result, the message cannot be encrypted with different session keys for different RPs.
So, an option is to share the session key (used in the communication with the first RP) with the additional RP by generating a new session key (as expected by the pairing algorithm) to forward the other key. Anyway, instead of wasting resources and time in generating a new session key for a very limited scope, the end device can be configured to reuse the same parameters used to generate the session key with the main RP thus avoiding sending the key.
A better solution than sharing the existing session key is to use a variant of the pairing algorithm where the final stage of generating a session key is removed. Indeed, as sensor's data are readable from the main RP, the idea is to create a secure channel over IP between the two peers (i.e. the main RP and the additional RP) and forward data there.
This means that as new RPs are added through the scale-out process, more connections the main RP has to set up with those to make them available sensor data. Due to constrained resources, multiple levels of indirection can be used resulting in a path of RPs where the number of connections per RP can be limited by an upper bound.
Finally, the last resort is to send raw data to the client and perform processing on the client device instead of on a LoRa gateway. In this case, the traffic size over IP is not reduced by locality processing but the computing power and resources are potentially infinite since the framework can scale seamlessly over any client's device. So, when all RPs have reached their connections limit, this solution can be used instead of denying the user the possibility to execute its query.

So, a user interested in values produced by multiple sensors or submitting multiple queries about the same sensor may be connected to multiple LoRa gateways because either the sensors are associated with different RPs or the scale-out algorithm has selected other RPs. Obviously, if two sensors share the same RP and scale-out is not necessary, both data resulting from the relative queries are sent on the same connection to save resources.

To implement the scale-out algorithm as a path of RPs to forward data over IP, the framework needs

- a further LoRa port for communicating the end device to start the variant of

the pairing algorithm (SCALE-OUT message),

- a further TCP port for setting up a new secure connection over IP between two RPs

The downside of executing the pairing algorithm (although partially) in the scale-out procedure is that the end device cannot send collected data at the same time. Hence, when a user causes a sensor to scale out, all the other users connected to the same sensor no receive data until the algorithm is finished because no data is sent.
To avoid a long blocking time, the sensor can alternate a message for the pairing algorithm variant and a data message, giving priority to users already connected to the system in a trade-off between scale-out execution time and old users blocking time. In order not to lose data, the sensor can queue it with associated timestamps and then send it in a single message (hopefully LoRa payload size should be sufficient to accommodate two messages in one as data size should be small).
To minimize the delay due to the alternation of messages and not affecting already connected users to the same end device, the pairing algorithm for the scale-out can be revised again towards a sub-optimal choice of the additional RP by delegating the main RP to search it (i.e. the RP is found among nearby LoRa gateways of the main RP instead of nearby gateways of the sensor). So, when the end device is notified about the scale-out process via the SCALE-OUT message, instead of performing the HELLO stage, it continues to send data to the main RP. Then, in the next downlink window, the main RP directly sends the FORWARD message to nearby LoRa gateways and proceeds with the pairing algorithm acting on behalf of the sensor; only the PAIRING message is sent by the sensor itself introducing just this delay.
Even better, I can cut off the end device from the scale-out procedure (as in the building of a path of RPs nothing changes in the messages sent from the end device) so that data is continuously submitted without any delay. Thus, instead of performing the association via the PAIRING message sent by the end device, this is automatically sent by the main RP and once the binding is finished, the sensor is just notified of the new association as well as the network server.

To secure forwarded data and get a secure IP connection, the possible implementations are TLS and IPsec. While TLS works at the transport layer, IPsec works at the network layer, so it is transparent to applications representing a benefit for the framework. Since setting up Security Associations in real-time is unfeasible and always error-prone, an IKE daemon is the only real valid option. It is natively supported in the Linux kernel but also complete packages are available such as strongSwan and Libreswan.

Furthermore, a smart-resources usage plans to execute the same query on a LoRa gateway once in case of multiple users submitting it twice or more and then send processed data to corresponding clients on different connections. To achieve this, the proposed framework can take advantage of the stream processing engine architecture where a data flow resulting from a submitted query can be fed with multiple data streams as specified in the equivalent queries. Even better, a query can be decomposed into single operations so that even non-equivalent queries can use components instantiated once and fed sequentially with different data. Obviously, in doing this the system needs to find a balance between available RAM and performances (penalized by reducing parallel computing). This job is expected among the stream processing engine features.

10.3 Client authentication

From a security point of view, the framework needs to implement access control to respect the privacy of people whose data is collected by IoT sensors. In fact, usually in an IoT deployment, sensors send sensitive data that shouldn't be publicly disclosed (to avoid pattern recognition about lifestyles and activities [5, 82]), and therefore a user of the framework must be able to access only resources to which it is entitled.
To achieve this result and consequently avoid a user's client can read data not intended for him/her, I can get inspired by Linux groups and corresponding privileges such that data collected by a sensor are only available to users in the related group with execute and read privileges. Clearly, the sensitive part consists in adding users to a group because the system must verify the real identity of the user before allowing it to read and operate on the sensor's data in order to prevent possible leaks and violations.
To accomplish this, I need to authenticate users and a common approach consists in deploying a DB on the cloud as a large data storage is needed to face a huge number of accounts and to verify concurrent login attempts. In the account creation, I have to be aware of the upsides and downsides of

- Letting users be free to create accounts
- Letting admins create the accounts on behalf of the users

The main difference is that the first is a completely automatic process while the second is a human process that cannot scale as well but essentially both options suffer from the same issue mentioned above: trust a user identity. In fact, if anyone can

create an account, a simple authentication (e.g. email and password) is not sufficient to assess the real identity of the user and even if admins create user accounts, then credentials must be provided only to legitimate users, so, a method is still necessary to verify if the person is requiring such credentials is actually who is claiming to be.

However, to be precise, the validity of this issue depends on the particular context in which the framework is used and how the sensors are implemented because it is different if the framework is accessible solely as a web service rather than being integrated and employed by an organization. to which people subscribe to plans or purchase items and which therefore involves a physical or digital purchasing process with the customer (e.g. telephone company, Internet service provider). Indeed, in this situation, when a customer enters into a contract or buys a sensor from the organization, it can provide an email address (or some other communication channel) as a contact to which the company can send future credentials without the need of the framework of demonstrating the real identity of the client during the authentication phase because only the proper user will receive the credentials for its own sensor via the provided contact (their management is out of the scope of this discussion).
So, the organization can decide between the two account creation options analyzing benefits and disadvantages without worrying about the identity issue and, as stated before, the choice may be reduced to put in place either a manual or an automatic process.
In the first case, the organization admins employed in the framework maintenance have to associate the customer personal information with the related sensor(s) as specified in the purchasing process, create a user account, and then send the resulting credentials to the customer in order to be able to access the system.
In the second case, since it is impossible to have a priori knowledge of who will be interested in the data collected by a specific sensor, the system needs additional information to realize the mapping between a user and the related sensor(s). This info may be represented by a secret for accessing a determined resource which means shifting the association from $person \rightarrow resource$ to $secret \rightarrow resource$. Therefore, the organization needs to provide such a secret, that acts as a function to the range of EUIs of deployed end devices, to the customer through the given contact. Then at registration time, the customer will enter the received secret with the aim of associating the new account to the sensor(s).
The technical implementation for giving the users the possibility of creating accounts on their own consists of a table of entries (device EUI, secret, user IDs) and of an inverted indices table (secret, device EUI) automatically populated in two phases:

1. after an end device activation, the network server can request to the cloud to

add an entry for the current device EUI involved in the process. If there does not exist an entry for such EUI (primary key), then a record is added to the table with a corresponding secret (UID) generated on the fly and either one or multiple user IDs corresponding to the admin accounts for monitoring data (possible privacy concerns) or a null user ID according to the application. At the same time, an entry (secret, device EUI) is added to the table of inverted indices

2. during user's registration in the application, to add the user ID to an existing entry of the primary table, the system retrieves the device EUI corresponding to the provided secret from the inverted indices table and then updates the record of the primary table

Of course, the secret must be securely stored because if it is compromised, data leaks can occur as it is used to add a user to the group of a sensor. To prevent this, the system can run a trigger function to avoid multiple users redeeming the same secret and, as a last resort, the manual intervention of an admin can change the secret and contact the proper user through the communication channel to give the possibility to access its data again. A similar intervention can occur when a customer unsubscribes to the company.

Nevertheless, considering the framework as not integrated into a similar context or although integrated designed in another way, to retrieve the real identity of a user the system can rely on public administration authentication methods (e.g. Italian SPID) that uniquely bind a citizen to digital identity. In such a way, it is also easier to traceback illegal attempts via system logs although there is always the possibility that the digital identity credentials are compromised. However, this is beyond the scope of the framework because it is a matter of the third-party entity and the user.

However, once the user identity has been verified by a trusted third-party organization, the framework needs to map this identity to the end devices the user is allowed to access. So, the situation is similar to the latest analyzed since a priori knowledge is impossible to have.
As in this scenario, the framework is a web service independent from organizations that deal with customers, the additional information required for performing the mapping must be retrieved from a collaboration between the company managing the framework and these other businesses.
For example, in our specific scenario, sensors are collecting data about water consumption that implies collaboration with the water supplier(s); the proposed framework can take advantage of the third-party entity and in particular of the contract between

the physical person and the water supplier. Therefore, instead of generating a unique secret identifier for every deployed end device (as in the solution explained above), here an alternative idea is to generate such a secret for each contract associated with a determined number of sensors that the user must provide at registration time on the app developed by the company managing the framework. Such an identifier must be provided by the third party to the user via a given contact granted when this enters into a contract with the supplier itself. In this way, only the legitimate person who has this identifier can create an account to access data collected by the sensor associated with the contract.
Once the user has performed the registration, the system can map the secret to the device EUIs included in the contract by querying either the supplier(s) web service(s) through APIs (authenticating and specifying the secret) or an internal table containing data received from the supplier via a communication channel by manually creating an internal table with data received from the supplier.
For sure, if available, communicating with the supplier(s) web service(s) represents the best option because is more flexible than manually creating and continuously maintaining a table populated with data belonging to an external organization because additions and updates are immediate since made by the supplier itself and also the surface for privacy breaches or GDPR violations is reduced.
Moreover, when collaborating with a supplier, the secret can be replaced by the digital identity resulting in an easier approach for the user and avoiding possible misuses of the secret itself.
Of course, the smart water metering scenario and the micro-services architecture can be extended to any other scenario where a supplier or a service provider is involved in the data measurements of the IoT sensors.

When a user is finally authenticated, it is provided with a session token (either JSON Web Token or OAuth 2.0 token) [75, 44, 69] denoting which device EUIs can be accessed; therefore, it has to be attached to every subsequent request issued by the user's client to the LoRaWAN networks. So, when a new request from the Internet comes to a LoRa network server, the first action that takes place consists of verifying the validity of the token, the corresponding privileges of the user's request and the validity of the request itself (i.e. input validation). Only if these steps are true, then the network server proceeds to look up the table of associations.

Access as an admin deserves a deeper check due to the higher privileges required, hence, it may be restricted to a range or list of IP addresses (e.g. whitelisting the admin network) and/or implement an additional authentication (like proxy authentication) where just a few account credentials need to be stored on the network server itself.

Merging the scale-out approach and the privacy motivation, someone can think the scale-out is not needed because probably a user will access only its own sensor for which only it has necessary privileges. However, this is true in some cases (e.g. household utilities) but may no longer be valid in others such as for organizations having sensors deployed on multiple buildings (e.g. university) where the group having privileges to access a sensor is not limited to a single person. Furthermore, each sensor's group usually contains one or more admin accounts for injecting queries (e.g. water supplier wants to know customer consumption, detect frauds and so on) although this may be a source for privacy concerns.

Since in the micro-services scenario the cloud does nothing else than querying the external organization and generating a session token, an edge computing approach consists in performing such operation (bound to user authentication and device mapping) directly at the edge. So, instead of contacting the cloud (central server), the user can directly reach the LoRa network servers regarding the end device data it wants to access.
However, I have to consider that the framework can handle multiple LoRaWAN networks that can be independent of each other (e.g. a subset for each city where the organization deployed the sensors) and therefore just a subset of all network servers should be contacted by the user to save resources.
To achieve this, the digital identity is useful again because according to the city where the person lives or the location where the end device is deployed retrieved from the third-party supplier, the user's application can contact exclusively the IP addresses of the location of interest: either via a multicast IP address per city or via a list of IP addresses per city.

10.4 Client connection to RP algorithm

After these observations, the task assignment algorithm may consist of:

1. The user's client sends a QUERY-REQUEST message (including the query and the end device EUI) to the RP (whose IP address has been retrieved after being authenticated and communicating with the LoRa network servers) to which the end device of interest with the required device EUI is connected to, in order to find out if there's room for the new stream processing task of the user

2. The stream processing engine on the RP checks if an instance of an equivalent task is already running as may happen that multiple clients submit the same

query (e.g. default query set in the application through which users access sensor's data)

3. If an instance is found, then the RP replies to the client positively (i.e. it sends an ACK which denotes the client can connect to it since an equivalent task is already started)

4. Otherwise, the RP checks if has enough resources to accomplish the new task (such as CPU-load threshold, storage threshold, network stats, ...)

5. If successful, then the RP starts the task and replies to the client positively (i.e. it sends an ACK so that the client can connect to it since the task has been started)

6. Otherwise, the client is notified by the RP (to which the sensor is connected) that it is currently busy via a NACK. At the same time, the RP starts the scale-out algorithm

7. When the scale-out algorithm finishes, the RP sends to the client the new IP address to which request the execution of the query

8. The client receives the message and the loop restarts (it should end at 3. or 5.)

10.5 Scale-out algorithm

The scale-out algorithm may be defined as follows

1. The first (main) RP to which the sensor is connected sends to the requested end device (in the first available downlink window) the SCALE-OUT message to notify it the scale-out process is beginning

2. The sensor receives the message and continues to send data as usual expecting to be notified shortly when an association is performed

3. The main RP sends a FORWARD message to nearby LoRa gateways

4. Nearby LoRa gateways that receive the message replies to the main RP with STATS messages

5. The main RP selects a gateway among the responses based on resource utilization and sends to it a PAIRING request over IP

6. The selected gateway replies to the main RP positively or negatively

7. The main RP notifies the end device and the network server about the new association

An improvement w.r.t. the research phase of the additional RP in 5. can be that the selection considers the user query load so that the resulting RP has enough resources to perform it before contacting the client. This improvement minimizes the network traffic outside the LoRaWAN network and avoids deadlock situations in which always the same RP is chosen infinitely.

This strategy implies that a sensor can upload its profile on multiple RPs and the same sensor is mapped to multiple RPs on the LoRa network servers. In this way, the client does not retrieve just an address but a list sorted by FIFO order.

10.6 Data profile

So far, I have not discussed the data profile included in the PAIRING message once an end device has selected the RP to associate with.
A data profile consists of a set of attributes that defines the properties of the related sensor. Hence, in a generic heterogeneous deployment, it can report the type of the device, its mobility degree, the quality of its measurements and so on.
As in the proposed framework, all end devices are static water metering sensors, reporting the type of the device is redundant but useful information may be represented by the location where the sensor is installed (e.g. GPS coordinates) or some related human-readable identifier (other than the device EUI) of which a client can take advantage to remotely easy inspect a particular sensor (e.g. *sapienza_building_a* can uniquely identify a sensor placed in building A of Sapienza University of Rome). Furthermore, if the end device location is either represented by GPS coordinates or identifiers hierarchically organized, a client can also easily address multiple sensors at the same time with a single stream processing query. For example, our university can have deployed a sensor per building and each one includes in the profile a different location ID $Sapienza/Building/A$, $Sapienza/Building/B$, ..., $Sapienza/Building/I$. To address them submitting a single query, the client has to specify the $Sapienza/*$ location where * is a wildcard for all subsequent levels (equals to # in an MQTT topic).
If instead GPS coordinates are used, the client can specify a matching range for the latitude and another for the longitude (e.g. [[41.8 - 42], [12.3 - 12.5]]) for the devices it is interested in. However, this is feasible if the end devices are provided with a GPS sensor or if the coordinates are manually entered at deployment or triangulating every LoRa signal of an end device received by at least three LoRa

gateway.
As explained above, in a REQUEST-QUERY message the user needs to include the query to be executed and the device EUI(s) whose data stream(s) is (are) given in input to the query. Location identifiers are more user-friendly than entering a device EUI or even worse a list of device EUIs because intuitive strings for humans rather than alphanumeric codes result in a simpler usage.
To achieve it, the framework needs a mapping between the device EUI and the location identifier and this is obtained through the data profile. So, when a user contacts a network server providing a location ID, the server can use a table indexed by this identifier instead of the device EUI so that an entry consists of (location ID, device EUI, RP's IP address). Levels and wildcards are implemented by parsing a location ID string, in fact, for levels, simple tokenization on the '/' character is sufficient (insensibility to upper and lowercase can be added for simplicity) while for wildcards a simple character matching is enough for detecting them.

Nevertheless, the end devices associations can be distributed on multiple LoRa gateways and, although this is not a problem for a query addressing a single end device, may instead be not suitable for a particular stream processing task where multiple data streams need to be joined together.
To elaborate them, I can think to work at an upper layer than the RPs and therefore employ the LoRa network server which receives all data produced by the sensors but elaborated by stream processing at the RPs.
The client can therefore submit the stream processing task to the LoRa network server and it can notify the RPs of interest not to elaborate the requested sensors data but to pass them directly to the network server itself while simultaneously continuing the previously assigned stream processing tasks.
In this way, as the network server is the gateway towards the Internet, it may receive the same data twice, once raw and once processed (at least a query addressing multiple end devices connected to different RPs); the former will be collected until the last sensor's data of interest is received and then processed together as asked by the client, the latter will be forwarded to the cloud or a user for other goals.
Nonetheless, in such a case, if multiple clients connect and submit tasks to the LoRa network server, it can be overloaded and cannot guarantee scalability at the edge of the network.

Conversely, a better edge computing solution considers taking advantage of LoRa gateways and submitting the stream processing queries to them instead of relying on the LoRa network server, also in case of join queries regarding end devices connected to different RPs. So, taking up the previous example where the user is interested in

all Sapienza sensors if multiple Sapienza end devices are connected to multiple RPs, the freer and with more connected Sapienza sensors can be selected to receive data also from other Sapienza sensors not connected to it.
The result is similar to a scale-out of the sensor data and as the hierarchical organization should also imply proximity of sensors (e.g. school, university, business, condominium), should not degrade performances too much.
So, when a user sends a CONNECTION packet to the framework, the LoRa network server immediately knows if it addresses multiple RPs or not because when parses the location ID retrieves the corresponding end device EUIs and RP's IP addresses and if the number of RPs is greater than one means the end devices are distributed on multiple RPs. At the same time, the network server can sort the RPs by descending order of connected end devices in which the user is interested. Then, starting from the head of the list, the network server runs an algorithm for finding the suitable LoRa gateway where execute the user query (the choice can be based on the parameters used by the pairing algorithm since the query is not included in the CONNECTION packet). Thus, the network server can contact the first RP of the list over IP and ask it to connect to the remaining peers in the list. Like the scale-out process, secure connections are set up to receive other end devices' data and in the end, the RP replies to the network server which in turn responds to the user with the IP address of the RP to which submit the query. At this point, the only issue may be represented by the impossibility of the RP to execute the query and its will to scale out. As this implies also a migration of secure connections with other RPs, it is better to carefully choose the RP to avoid a similar situation. An improvement may be represented by including the query in the CONNECTION packet so that when selecting the RP the query is forwarded to calculate if the RP can sustain it. So, the CONNECTION and QUERY-REQUEST should be joined in a single packet.

Another important aspect of the data profile is that it can be used to filter incoming messages at the LoRa gateway. In fact, in LoRaWAN the end device always sends messages in broadcast but, in the proposed framework, they are encrypted with a session key shared either with the gateways in the radio range or with the gateway to which it is connected. After running the pairing algorithm and being associated with the best fit gateway, the gateway needs to store in a table the associations with the different sensors including the data profile. If no session key is generated, all gateways in the sensor's range can decrypt the frame but only the selected gateway must process the stream of data. This can be achieved by filtering messages based on the aforementioned table of associations.
Stronger filtering is represented by the generation of a session key between the

end device and the associated LoRa gateway such that CIA for transmitted data is ensured. Clearly, this requires a greater effort for the two parties involved in communication and sufficient device data storage while the overhead is constant because the common key encryption is replaced with the two-parties key encryption.

10.7 MQTT

An improvement to networking performances may be achieved by replacing HTTP with MQTT at the transport layer of the OSI model.
The Message Queuing Telemetry Transport (MQTT) is a standard lightweight messaging protocol designed for the Internet of Things. It is based on the client-server publish/subscribe message pattern and usually works over TCP/IP but can also be supported by any network protocol that provides ordered, lossless, bi-directional connections [8].
The server role is performed by the MQTT message broker that routes all incoming messages from the clients (publishers) to the appropriate destinations (subscribers) through topics. The MQTT client consists of any device (from a microcontroller up to a smartphone or a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

LoRaWAN is a Media Access Control (MAC) layer protocol built on top of LoRa modulation (physical layer), that mainly acts as a network layer protocol for managing communication between LoRa gateways and end devices, where TCP/IP is only available for the data exchanged between the LoRa gateways and the LoRa network server. Nevertheless, the entire network implements mechanism for frame ordering, acknowledgments and, of course, provides bi-directional communication between end devices and the network server. So, it should be theoretically feasible to map the LoRa network server or the LoRa gateways to the message broker(s) and the end devices to the clients.
However, the MQTT specification suggests using the version for sensor networks (MQTT-SN) if devices are deployed on non-TCP/IP networks because it is designed to extend the MQTT protocol beyond the reach of TCP/IP infrastructure. Therefore, I can opt for one of the following architectures:

- extend MQTT to end devices through Static Context Header Compression (SCHC),
- extend MQTT to end devices through MQTT-SN,

- limit MQTT to the IP connection between the LoRa gateways and the LoRa network server without altering the communication between end devices and gateways.

About the first option, at the time of writing SCHC only supports the Constrained Application Protocol (CoAP) but maybe in the near future, MQTT will be introduced and might be compared with the second alternative of the list (i.e. MQTT-SN) [60, 59]. Unlike MQTT, CoAP provides a request/response interaction model between application endpoints and it is more similar to REST for IoT constrained devices [76].
The main difference is that MQTT is a one-to-many protocol while CoAP is a one-to-one protocol, so for the proposed framework, the former is suitable.

Indeed, the idea is to deploy the MQTT clients on the end devices and make the LoRa gateways or the LoRa network server act as message brokers. However, in the original SCHC proposal, following the general architecture and mode of operation of LoRaWAN, compression and decompression of packets are exclusively performed by end devices and the LoRa network server. So, if MQTT would be supported, it should be easy to place the message broker on the network server while some adaptations to SCHC are needed to deploy multiple message brokers on the LoRa gateways. Careful evaluations may be carried on to justify an eventual effort to such modifications.

Taking a look at other proposals in the same research field, there are related works (such as The Things Network, ...) in which the MQTT message broker is placed at the cloud and the LoRa network server performs the role of the MQTT client.
Moving to an edge computing solution and leaving out the SCHC idea, I can reflect on the other aforementioned alternatives for implementing MQTT in the proposed framework.
Using MQTT-SN, the end devices can implement the clients, the LoRa gateways the MQTT-SN gateway and the LoRa network server the MQTT broker.
Limiting MQTT to available IP connectivity, the end device can send data over LoRaWAN to the gateways as usual while the gateways publish it using MQTT. In this case, I can have the gateways run the MQTT clients and the network server the MQTT broker again.

In both approaches, a user interested in just reading data produced by a specific sensor, instead of submitting a query to be executed in a stream processing engine, can simply subscribe to a topic (denoted by the location ID string rather than the

device EUI as explained before) at the network server (broker) and receive data via MQTT instead of HTTP with the related advantages. The CPU of the LoRa gateway also benefits from this implementation because an MQTT sending is lighter than running a stream processing query.
On the other hand, the query is still necessary for stream processing tasks other than just reading data, so, the user needs to publish it on a specific MQTT topic at the network server that denotes the location ID and then subscribe to a topic where the results will be published. The topic and relative message broker is provided by the network server remembering that a scale-out may be necessary.

Therefore, with this improvement, the network server can open a port for MQTT and another for the pairing algorithm or just the MQTT one if also the algorithm is executed over MQTT. For this reason, is probably better to set up MQTT at the very beginning rather than doing this after the end device association with the gateway so that the stream of data is immediately published on an MQTT topic.
Although the message broker is able to accept multiple connections by nature, it is usually deployed on the cloud which provides apparent seamless scalability. Since this is not the case of the network server, to linearly increase the scalability capabilities of the message broker, I can deploy multiple brokers at the LoRa gateways. A comparison can be run out to evaluate the resulting performances.
So, the network server can implement an MQTT client instead of a message broker, subscribe to all topics generated by the application by using wildcards in order to receive sensor's data over MQTT, and forward them to the cloud for enabling massive storage and performing historical analysis by publishing such data on a topic addressed to an MQTT message broker on the cloud. The resulting topic can be the same received at the network server by prepending a level denoting the LoRaWAN network identifier (e.g. $East\_net/Sapienza/Building/A$ using location ID strings).

To avoid the situation in which a user has to subscribe to multiple (or all) LoRa gateways to receive multiple (or all) sensor's data within the network, instead of relying on the broker located on the cloud for performing a single subscription, privileging the edge scenario the LoRa network server can act as MQTT client and broker at the same time by running as a bridge.
In this way, the framework has a fine-granularity control on incoming connections and can decide if allocate a stream processing query in the network server or a LoRa gateway based on user topics (i.e. location identifiers).

10.8 Client disconnection and query cancellation

Just as a user can connect to a RP and submit queries for a given end device, it can also perform the opposite actions: cancel active queries and disconnect from the RP.
To remove a submitted query, the user can send a CANCEL packet specifying the ID received when the query was previously sent. The implementation is indeed quite simple since the RP binds each query for an end device data stream with a session token, the end device EUI and a corresponding ID (generated when the query is positively received from the RP) and stores the entry in a table. If the cancel request matches an entry (session token and query ID), then the RP calls the corresponding streaming engine API to cancel the task. When removed, the RP sends an ACK to the client. When a user has no active query, a timeout on the RP is started so that if the user does not submit queries for a determined interval of time, then it is automatically disconnected to free resources for other potential users.
If instead, a user wants to explicitly disconnect from the RP, then it must send a DISCONNECTION packet specifying the device EUI of the sensor or a zero payload to disconnect from any end device to which it is connected. This is achieved similarly to the cancel request; since the client provides the session token, then a matching entry in the table of connected clients (session token, end device EUI, query ID) must be found. If the user has not explicitly canceled previous queries, then the RP scans the list of active queries retrieved from the table and calls the stream processing engine API to remove the tasks. When removed, the RP deletes the entry from the table of connected clients and sends an ACK to the client.

10.9 Scale-in

Once a query has been canceled or a user has been disconnected, the framework needs to take care of the scale-in process. In fact, one of the two aforementioned operations can be subsequent to a previous scale-out. As the framework implements the scale-out as a path of RPs over IP, then the scale-in process needs to remove this path; nothing simpler. However, to save resources the same channel between two RPs can be shared by multiple end devices data, so before deleting it, the main RP needs to ensure that no other data is sent over this connection. So, the Security Association is deleted from the corresponding database and the entry in the table to forward data from a specific end device EUI to a RP's IP address too.

Chapter 11

Conclusions

Large-scale IoT applications produce massive data volumes and edge computing solutions including stream processing represents a suitable solution to efficiently elaborate it. Low-Power Wide Area Network (LPWANs) is a standard choice to realize a large-scale IoT deployment able to support IoT application requirements: low-power, long-range and low-cost transmission.

Starting from a smart water metering use case and the related importance of implementing data analysis on collected measurements to extract information about water consumption in order to react in real-time to anomalies, let end users monitor consumption in real-time and prepare ready-to-use data for further analysis on the cloud, this master thesis proposes a framework based on LoRaWAN to generate, process and consume data collected by IoT devices at the edge. LoRa gateways, geographically distributed in an urban environment rather than in a rural area, compose a layer of rendezvous points that constitutes the core of the system. Through an innovative Gateway-Device Coordination Protocol over LoRaWAN, every end device in the network binds to the most suitable gateway according to location and resources load where the computational tasks about the generated data stream must be executed. Many observations have been reported and taken into consideration throughout the entire writing to design, implement and evaluate the protocol to the best of my skills acquired in the two-year Master's Degree study plan. The contribution includes a gateway activation method based on the LoRaWAN join procedure that is essential to provide CIA of communications. To enable end users' data access and processing, two algorithms are presented to directly connect to a LoRa gateway and submit stream processing tasks.

The implementation of the Gateway-Device Coordination Protocol has been performed using OMNeT++ and different scenarios are reproduced via two simulation models: one stochastic and the other based on values collected about a real LoRaWAN deployment in the city of London throughout several days.

The evaluation conducted on the implementation of the protocol in terms of correctness, network utilization and network scalability (gateway resource utilization, associations load balancing and latency) shows that an effective load balancing is achieved based on gateway proximity to the end devices and gateway resources utilization through a small number of LoRaWAN messages (about 13 per end device to complete a protocol run including OTAA and 3 retransmissions for specific message types in order to maximize frame delivery ratio) where the network server contribution is limited to verification of the two entities that are associating in order to reduce latency. Besides the small size of the network deployments (6 and 13 end devices and 2 gateways), the random wake-ups reduce the number of message collisions resulting in a 99.28% delivery ratio. Finally, resources utilization suggests that the storage size of gateways will not represent a bottleneck in large-scale networks as the impact of the protocol on gateway storage of 8GB results in an imperceptible increment of occupied storage of about $2.5 \cdot 10^{-6}\%$ per associated end device.

As future works, I would like to implement the protocol in Riot-OS, Chirpstack or LoRa Basics and then test it in the real world (e.g. using IoT-Lab testbed platform) to evaluate its actual performance and effectiveness as suggested by simulations. It is also important to develop a robust alternative to timestamps against out-of-range frame replays of HELLO_GATEWAY and GENERATE_COMMON_KEY messages, respectively used in gateway activation and in the generation of the common session key between an end device and the gateways in the radio range. Additionally, I would like to implement the entire framework to first evaluate the client connection and scale-out algorithms and then the overall performance of the system, which is currently focused only on the setup of the associations between devices and gateways. Then, a final evaluation against existing cloud computing or edge computing solutions should be performed to compare the results and expected improvements, possibly through experiments on real IoT application scenarios.

Bibliography

[1] 3GPP. *3GPP Specification Set: 5G*. URL: https://www.3gpp.org/dynareport/SpecList.htm?release=Rel-15&tech=4&ts=1&tr=1.

[2] Ferran Adelantado et al. "Understanding the Limits of LoRaWAN". In: *IEEE Communications Magazine* (2017).

[3] Arif Ahmed et al. "Fog Computing Applications: Taxonomy and Requirements". In: *arXiv:1907.11621* (2019).

[4] Gayashan Amarasinghe et al. "A Data Stream Processing Optimisation Framework for Edge Computing Applications". In: *IEEE 21st International Symposium on Real-Time Distributed Computing* (2018).

[5] Dimitrios Amaxilatis et al. "A Smart Water Metering Deployment Based on the Fog Computing Paradigm". In: *Applied Sciences* (2020).

[6] Dimitrios Amaxilatis et al. "Sparks-Edge: Analytics for Intelligent City Water Metering". In: *European Conference on Ambient Intelligence* (2019).

[7] Emekcan Aras et al. "Exploring the Security Vulnerabilities of LoRa". In: *3rd IEEE International Conference on Cybernetics (CYBCONF)* (2017).

[8] Andrew Banks et al. *MQTT Version 5.0*. Tech. rep. OASIS Message Queuing Telemetry Transport (MQTT) Technical Committee, 2019.

[9] Olivier Bernard, André Seller, and Nicolas Sornin. *Low power long range transmitter*. Tech. rep. Patent EP2763321A1, 2014.

[10] Laksh Bhatia et al. "Dataset: LoED: The LoRaWAN at the Edge Dataset". In: *The 3rd International SenSys+BuildSys Workshop on Data: Acquisition to Analysis (DATA '20)* (2020).

[11] Thomas Boyle et al. "Intelligent Metering for Urban Water: A Review". In: *Water* (2013).

[12] Thilina Buddhika and Shrideep Pallickara. "NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments". In: *IEEE International Parallel and Distributed Processing Symposium* (2016).

[13] Anderson Carvalho et al. “Edge Computing Applied to Industrial Machines”. In: *Procedia Manufacturing* (2020).

[14] Bharat S. Chaudhari, Marco Zennaro, and Suresh Borkar. “LPWAN Technologies: Emerging Application Characteristics, Requirements, and Design Considerations”. In: *Future internet* (2020).

[15] Bin Cheng, Apostolos Papageorgiou, and Martin Bauer. “Geelytics: Enabling On-demand Edge Analytics Over Scoped Data Sources”. In: *IEEE International Congress on Big Data* (2016).

[16] Sanket Chintapalli et al. “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016).

[17] European Commission. *Smart grids and meters*. URL: https://energy.ec.europa.eu/topics/markets-and-consumers/smart-grids-and-meters_en.

[18] David Corral-Plaza et al. “A stream processing architecture for heterogeneous data sources in the Internet of Things”. In: *Computer Standards & Interfaces* (2020).

[19] Daniele Croce et al. “Impact of LoRa Imperfect Orthogonality: Analysis of Link-Level Performance”. In: *IEEE Communications Letters* (2018).

[20] Roshan Bharath Das, Gabriele Di Bernardo, and Henri BalEduard Renart. “Large Scale Stream Analytics using a Resource-constrained Edge”. In: *IEEE International Conference on Edge Computing* (2018).

[21] Roshan Bharath Das, Nicolae Vladimir Bozdog, and Henri Bal. “Cowbird: A Flexible Cloud-Based Framework for Combining Smartphone Sensors and IoT”. In: *5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)* (2017).

[22] Shilpa Devalal and A. Karthikeyan. “LoRa technology-an overview”. In: *Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)* (2018).

[23] Stephen Farrell. *Low-Power Wide Area Network (LPWAN) Overview*. Tech. rep. RFC 8376; Internet Engineering Task Force (IETF), 2018.

[24] Cheng Feng et al. “Smart grid encounters edge computing: opportunities and applications”. In: *Advances in Applied Energy* (2021).

[25] The Apache Software Foundation. *Apache Flink*. URL: https://flink.apache.org/.

[26] The Apache Software Foundation. *Apache Flink Downloads*. URL: https://flink.apache.org/downloads.html.

[27] The Apache Software Foundation. *Apache Kafka*. URL: https://kafka.apache.org/.

[28] The Apache Software Foundation. *Apache Spark*. URL: https://spark.apache.org/.

[29] The Apache Software Foundation. *Apache Storm*. URL: https://storm.apache.org/.

[30] Xinwei Fu et al. "EdgeWise: A Better Stream Processing Engine for the Edge". In: *USENIX Annual Technical Conference* (2019).

[31] Oscar García et al. "A Serious Game to Reduce Consumption in Smart Buildings". In: *International Conference on Practical Applications of Agents and Multi-Agent Systems* (2017).

[32] Óscar García et al. "CAFCLA: A Conceptual Framework to Develop Collaborative Context-Aware Learning Activities". In: *Workshop on Learning Technology for Education in Cloud (LTEC'12)* (2012).

[33] Vijay K. Garg. *Principles of DISTRIBUTED SYSTEMS*. Springer, 1995.

[34] Haralampos Gavriilidis et al. "Scaling a Public Transport Monitoring System to Internet of Things Infrastructures". In: *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT)* (2020).

[35] Haralampos Gavriilidis et al. "Scaling a Public Transport Monitoring System to Internet of Things Infrastructures". In: *EDBT* (2020).

[36] Hêriş Golpîra, Syed Abdul Rehman Khan, and Sina Safaeipour. "A review of logistics Internet-of-Things: Current trends and scope for future research". In: *Journal of Industrial Information Integration* (2021).

[37] Heitor Murilo Gomes et al. "Machine learning for streaming data: state of the art, challenges, and opportunities". In: *ACM SIGKDD Explorations Newsletter* (2019).

[38] Aurora González-Vidal, Jesús Cuenca-Jar, and Antonio F. Skarmet. "IoT for Water Management: Towards Intelligent Anomaly Detection". In: *IEEE 5th World Forum on Internet of Things (WF-IoT)* (2019).

[39] Darshankumar Vinubhai Gorasiya. "Comparison of Open-Source Data Stream Processing Engines: Spark Streaming, Flink and Storm". In: *Technical Report* (2019).

[40] Jetmir Haxhibeqiri et al. “A Survey of LoRaWAN for IoT: From Technology to Application”. In: *Sensors* (2018).

[41] Jetmir Haxhibeqiri et al. “LoRa Scalability: A Simulation Model Based on Interference Measurements”. In: *Sensors* (2017).

[42] Haruna Isah et al. “A Survey of Distributed Data Stream Processing Frameworks”. In: *IEEE Access* (2019).

[43] Shikha Jain et al. “Internet of medical things (IoMT)-integrated biosensors for point-of-care testing of infectious diseases”. In: *Biosensors and Bioelectronics* (2021).

[44] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. Tech. rep. RFC 7519; Internet Engineering Task Force (IETF), 2015.

[45] Ji-Young Jung and Jung-Ryun Lee. “Throughput and Packet Loss Probability Analysis of Long Range Wide Area Network”. In: *Applied Sciences* (2021).

[46] Jeyhun Karimov et al. “Benchmarking Distributed Stream Data Processing Systems”. In: *IEEE 34th International Conference on Data Engineering (ICDE)* (2018).

[47] Kanitkorn Khanchuea and Rawat Siripokarpirom. “A Multi-Protocol IoT Gateway and WiFi/BLE Sensor Nodes for Smart Home and Building Automation: Design and Implementation”. In: *10th International Conference on Information and Communication Technology for Embedded Systems* (2019).

[48] Sandra Khvoynitskaya. *The IoT history and future*. 2019. URL: https://www.itransition.com/blog/iot-history.

[49] D. Kjendal. *RP002-1.0.3 LoRaWAN® Regional Parameters*. Tech. rep. LoRa Alliance Technical Committee, 2021.

[50] T. Kramp and O. Seller. *LoRaWAN® L2 1.0.4 Specification (TS001-1.0.4)*. Tech. rep. LoRa Alliance Technical Committee, 2020.

[51] Rajalakshmi Krishnamurthi et al. “An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques”. In: *Sensors* (2020).

[52] Abhaykumar Kumbhar. “Overview of ISM Bands and Software-Defined Radio Experimentation”. In: *Wireless Personal Communications* (2017).

[53] Hussein Kwasme and Sabit Ekin. “RSSI-Based Localization Using LoRaWAN Technology”. In: *IEEE Access* (2019).

[54] Hochul Lee et al. “iEdge: An IoT-assisted Edge Computing Framework”. In: *IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2021).

[55] OpenSim Ltd. *OMNeT++*. URL: https://omnetpp.org/.

[56] Jinshan Luo et al. "A Study on Adjacent Interference of LoRa". In: *Eighth International Symposium on Computing and Networking Workshops (CANDARW)* (2020).

[57] Praveen Kumar Malik et al. "Industrial Internet of Things and its Applications in Industry 4.0: State of The Art". In: *Computer Communications* (2021).

[58] Kais Mekki et al. "A comparative study of LPWAN technologies for large-scale IoT deployment". In: *ICT Express* (2019).

[59] Ana Minaburo, Laurent Toutain, and Ricardo Andreasen. *Static Context Header Compression (SCHC) for the Constrained Application Protocol (CoAP)*. Tech. rep. RFC 8824; Internet Engineering Task Force (IETF), 2021.

[60] Ana Minaburo et al. *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*. Tech. rep. RFC 8724; Internet Engineering Task Force (IETF), 2020.

[61] Mobilefish. *LoRa*. URL: https://lora.readthedocs.io/en/latest/.

[62] Andreas F. Molisch. *Wireless Communications*. John Wiley & Sons Inc, 2010.

[63] Nicholas Palmer et al. "SWAN-song: a flexible context expression language for smartphones". In: *PhoneSense '12: Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones* (2012).

[64] Srinath Perera. *Stream Processing 101: A Deep Look at Operators*. 2019. URL: https://medium.com/stream-processing/stream-processing-101-from-sql-to-streaming-sql-44d299cf38aa.

[65] Tobias Pfandzelter and David Bermbach. "IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?" In: *IEEE International Conference on Fog Computing (ICFC)* (2019).

[66] Raspberry Pi. *Raspberry Pi Operating system images*. URL: https://www.raspberrypi.com/software/operating-systems/.

[67] Ladislav Polak and Jiri Milos. "Performance analysis of LoRa in the 2.4 GHz ISM band: coexistence issues with Wi-Fi". In: *Telecommunication Systems* (2020).

[68] Eduard Renart, Javier Diaz-Montes, and Manish Parashar. "Data-driven Stream Processing at the Edge". In: *IEEE 1st International Conference on Fog and Edge Computing (ICFEC)* (2017).

[69] Justin Richer. *OAuth 2.0 Token Introspection*. Tech. rep. RFC 7662; Internet Engineering Task Force (IETF), 2015.

[70] David del Rio Astorga et al. “Paving the way towards high-level parallel pattern interfaces for data stream processing”. In: *Future Generation Computer Systems* (2018).

[71] Joerg Robert et al. “IEEE 802.15 Low Power Wide Area Network (LPWAN) PHY Interference Model”. In: *IEEE International Conference on Communications (ICC)* (2018).

[72] Semtech. *LoRa and LoRaWAN: Technical overview*. URL: https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan/.

[73] Kinza Shafique et al. “Internet of Things (IoT) for Next-Generation Smart Systems: A Review of Current Challenges, Future Trends and Prospects for Emerging 5G-IoT Scenarios”. In: *IEEE Access* (2020).

[74] Saeed Shahrivari. “Beyond Batch Processing: Towards Real-Time and Streaming Big Data”. In: *Computers* (2014).

[75] Rifaat Shekh-Yusef, Christer Holmberg, and Victor Pascual. *Third-Party Token-Based Authentication and Authorization for Session Initiation Protocol (SIP)*. Tech. rep. RFC 8898; Internet Engineering Task Force (IETF), 2020.

[76] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. Tech. rep. RFC 7252; Internet Engineering Task Force (IETF), 2014.

[77] Inés Sittón-Candanedo et al. “Edge Computing, IoT and Social Computing in Smart Energy Scenarios”. In: *Sensors* (2019).

[78] Jack Steward. *The Ultimate List of Internet of Things Statistics for 2022*. 2022. URL: https://findstack.com/internet-of-things-statistics/.

[79] George Suciu et al. “IoT time critical applications for environmental early warning”. In: *9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)* (2017).

[80] Ministero dello Sviluppo Economico. *Nuova TV digitale: Refarming*. 2022. URL: https://nuovatvdigitale.mise.gov.it/refarming/.

[81] Ministero dello Sviluppo Economico. *Nuova TV digitale: Roadmap*. 2022. URL: https://nuovatvdigitale.mise.gov.it/road-map/.

[82] Lo’ai Tawalbeh et al. “IoT Privacy and Security: Challenges and Solutions”. In: *Applied Sciences* (2020).

[83] Yuuichi Teranishi et al. “Dynamic Data Flow Processing in Edge Computing Environments”. In: *IEEE 41st Annual Computer Software and Applications Conference* (2017).

[84] Lilia Tightiz and Hyosik Yang. “A Comprehensive Review on IoT Protocols’ Features in Smart Grid Communication”. In: *Energies 2020* (2020).

[85] Ralf Tönjes et al. “Real Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications”. In: *European Conference on Networks and Communications (EUCNC)* (2014).

[86] Phyo Thu Zar Tun. “PATH LOSS PREDICTION BY USING RSSI VALUES”. In: *International Journal Of All Research Writings* (2018).

[87] Pal Varga et al. “5G support for Industrial IoT Applications— Challenges, Solutions, and Research gaps”. In: *Sensors 2020* (2020).

[88] Benny Vejlgaard et al. “Interference Impact on Coverage and Capacity for Low Power Wide Area IoT Networks”. In: *IEEE Wireless Communications and Networking Conference (WCNC)* (2017).

[89] Alexandra Voit et al. “Demo of a Smart Plant System as an Exemplary Smart Home Application Supporting Non-Urgent Notifications”. In: *Proceedings of the 10th Nordic Conference on Human-Computer Interaction* (2018).

[90] Shiqiang Wang, Murtaza Zafer, and Kin K. Leung. “Online Placement of Multi-Component Applications in Edge Computing Environments”. In: *IEEE Access* (2017).

[91] Kia C. Wiklundh. “Understanding the IoT technology LoRa and its interference vulnerability”. In: *International Symposium on Electromagnetic Compatibility - EMC EUROPE* (2019).

[92] Fatos Xhafa, Burak Kilic, and Paul Krause. “Evaluation of IoT stream processing at edge computing layer for semantic data enrichment”. In: *Future Generation Computer Systems* (2019).

[93] Shusen Yang. “IoT Stream Processing and Analytics in the Fog”. In: *IEEE Communications Magazine* (2017).

[94] Xueying Yang et al. “Security Vulnerabilities in LoRaWAN”. In: *IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)* (2018).

[95] Steffen Zeuch et al. “NebulaStream: Complex Analytics Beyond the Cloud”. In: *Open Journal of Internet of Things (OJIOT)* (2020).

[96] Steffen Zeuch et al. “The NebulaStream Platform: Data and Application Management for the Internet of Things”. In: *arXiv:1910.07867* (2019).

[97] Haibo Zhou et al. “Evolutionary V2X Technologies Toward the Internet of Vehicles: Challenges and Opportunities”. In: *Proceedings of the IEEE* (2020).