SAPIENZA
UNIVERSITÀ DI ROMA

Design, Development and Evaluation of a LoRaWAN Packet Simulator for Testing Realistic Large-Scale Experimentation & LoRaWAN Deployments

Department of Computer, Control and Management Engineering
Master Degree in Engineering in Computer Science

Carlo Carugno
ID number 1999815

Advisor
Ioannis Chatzigiannakis

Co-Advisors
Domenico Garlisi
Stefano Milani

Academic Year 2023/2024

Thesis defended on 25 March 2024
in front of a Board of Examiners composed by:

Prof. Domenico Lembo (chairman)
Prof. Irene Amerini
Prof. Ioannis Chatzigiannakis
Prof. Danilo Comminiello
Prof. Marco Console
Prof. Federico Fusco
Prof. Gabriele Proietti Mattia

Design, Development and Evaluation of a LoRaWAN Packet Simulator for Testing Realistic Large-Scale Experimentation & LoRaWAN Deployments
Sapienza University of Rome

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

Author's email: carugno.1999815@studenti.uniroma1.it

Abstract

This thesis presents the design, development, and evaluation of a LoRaWAN Packet Simulator aimed at enabling realistic large-scale experimentation and deployments within the LoRaWAN protocol. Motivated by the challenges associated with modeling the complex dynamics of LoRaWAN networks, particularly in large-scale deployments, this work introduces a novel simulation tool that captures the intricacies of network behavior, including node distribution, signal interference, and environmental impacts on signal propagation. Utilizing a comprehensive approach that encompasses both standard LoRaWAN and Edge2LoRa architectures, the simulator facilitates detailed performance analysis under various conditions, thereby aiding in network planning, testing, and optimization. Extensive testing, both in a controlled laboratory environment and through cloud-based platforms like The Things Network, underscores the simulator's robustness, adaptability, and potential to significantly contribute to the efficient design and deployment of LoRaWAN networks. The findings not only highlight the simulator's efficacy in replicating real-world network conditions but also its role in advancing the understanding of LoRaWAN's operational capabilities and limitations.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) has significantly reshaped technological landscapes, merging digital capabilities with the physical world [12]. At the heart of this revolution lies Long Range Wide Area Network (LoRaWAN) [4], renowned for its low-power, long-range communication ideal for IoT applications. In parallel, edge computing has emerged as a vital component, processing data closer to IoT devices, enhancing efficiency, and reducing latency [9]. Together, LoRaWAN and edge computing technologies are crucial for a range of applications, from environmental monitoring to smart urban infrastructure.

LoRaWAN deployments, especially on a large scale, encounter challenges such as network congestion, interference, and environmental impacts on signal quality [10]. Similarly, integrating edge computing with LoRaWAN, referred to as Edge2LoRa [15], presents unique challenges in ensuring seamless communication and data processing. The lack of comprehensive simulation tools capable of accurately modeling these complexities has been a significant bottleneck, affecting the optimal design and deployment of these technologies.

1.1 Objectives and Contributions of This Research

This research aims to fill the existing gaps by developing a versatile LoRaWAN Packet Simulator, which is capable of emulating a large number of devices within LoRaWAN networks. These devices can be designated as either standard LoRaWAN devices or as part of the Edge2LoRa protocol, allowing the simulator to be utilized in both traditional LoRaWAN and edge computing-enhanced LoRaWAN scenarios. Following the development of the versatile LoRaWAN Packet Simulator, capable of emulating devices as either standard LoRaWAN or Edge2LoRa units, this research proceeded to leverage the simulator for comparative network performance analysis. This involved conducting tests to elucidate the differences between networks operating under standard LoRaWAN protocols and those enhanced with Edge2LoRa technology, providing valuable insights into their respective efficiencies and potential applications.

1.2 Structure of the Thesis

This thesis is structured to guide the reader through the research process systematically. Chapter 2 offers a review of the existing literature on LoRaWAN and edge computing technologies. Chapter 3 delves into the technologies central to this research, setting the foundation for the simulator's development. Chapter 4 outlines the methodology and development phases of the LoRaWAN Packet Simulator,

including its capabilities to simulate Edge2LoRa scenarios. Chapter 5 presents a detailed analysis of experiments conducted using the simulator, extracting valuable evaluations of both the LoRaWAN and Edge2LoRa systems. The concluding chapter summarizes the research findings and discusses potential future directions.

Chapter 2

Related and Previous works

The integration of Long Range (LoRa) technologies into the Internet of Things (IoT) domain has established itself as a fundamental solution for connecting a multitude of sensor devices across widespread geographic areas. Despite its strengths, the centralized configuration of LoRa Wide-Area Networks (LoRaWAN) gives rise to substantial challenges. The reliance on a central network server for data processing introduces issues with scalability, latency, and operational costs, esasperated by the substantial data production from IoT devices.

The current landscape of research in the LoRaWAN domain is marked by a diverse range of methodologies and implementations. Notably, the work by Yousuf, Rochester, and Ghaderi [18] significantly enhances our understanding by showcasing the potential for achieving extensive network coverage using cost-effective LoRa solutions, even in challenging environmental conditions. Their results indicate that, despite less-than-ideal gateway placement, their affordable gateways and devices maintain reliable communication across various scenarios, particularly in environments with harsh propagation conditions and over long distances. This research has been instrumental in advancing my own work for several reasons.

Firstly, it underscores the environmental impact of these experiments. In an era increasingly focused on sustainability, investigating economical and energy-efficient IoT solutions such as LoRa networks is in line with worldwide initiatives aimed at minimizing the technological environmental footprint.

Furthermore, this work highlights the potential for scalability and capacity studies in LoRa networks. As the proliferation of IoT devices continues, comprehending how low-cost LoRa networks can manage escalating loads and a multitude of connections becomes critical. Such understanding is essential for the planning and deployment of large-scale IoT systems, making these insights particularly valuable for future research and development in this field.

In the field of parallel data processing, a significant strand of related literature involves the integration of Prefect with PySpark for orchestrating data workflows, as detailed in a post by Jose D. Hernandez-Betancur [8]. This literature presents a compelling case study for using Prefect and PySpark within a Docker environment to efficiently process data, with a specific focus on fuzzy string matching in movie datasets using Jaccard distance. This approach, showcasing an advanced use of workflow orchestrators, is particularly relevant to my research due to its emphasis on managing complex data processing tasks in a scalable and efficient manner.

This method supports complex systems by offering a structured approach to handle large-scale data and intricate processing tasks, which is pivotal for big data applications. It contrasts with traditional multi-device simulations by providing a more integrated and streamlined environment for data processing. The application of Prefect, in tandem with PySpark, demonstrates an efficient way to manage data pipelines, handle resource allocation, and execute dependent tasks in parallel. This is particularly informative for my work as it highlights the critical role of workflow orchestration and parallel processing in managing the complexities inherent in big data analytics. This literature underlines the necessity of robust and scalable implementations that can adeptly support complex data systems, a principle that is central to my own research efforts

In contrast, Pbench [3] introduces a unique perspective centered on data collection from an array of tools during benchmark executions. It offers a comprehensive approach to benchmarking various aspects of system performance, including CPU, memory, disk I/O, and network throughput. However, its applicability to LoRaWAN packet testing is somewhat limited. This limitation arises primarily because Pbench is optimized for general system performance analysis rather than the specific nuances of LoRaWAN networks.

LoRaWAN networks, particularly in large-scale deployments, present unique challenges that require specialized testing and analysis tools. These challenges include evaluating the network's ability to handle long-range, low-power communication, ensuring reliable message delivery in varying environmental conditions, and managing the network's scalability and capacity under different loads. Pbench, while robust in a general system performance context, lacks the specialized features necessary to simulate and analyze these specific aspects of LoRaWAN networks effectively.

Our primary objective is to develop a LoRaWAN Packet Simulator for Realistic Large-Scale Experimentation of LoRaWAN Deployments. Such a simulator would need to replicate the unique operational conditions of LoRaWAN networks, including their long-range communication capabilities, low power consumption characteristics, and the impact of various environmental factors on signal propagation. This requires a tool that goes beyond general system benchmarking to incorporate features tailored to the specific requirements of LoRaWAN technology, which Pbench does not currently offer. Therefore, while Pbench provides valuable insights for general benchmarking purposes, its direct application to the specific needs of LoRaWAN packet testing and simulation is limited, highlighting the potential necessity for more specialized tools in this domain.

In the landscape of LoRa/LoRaWAN communication tools, the 'lora-packet' library [11], developed for Node.js, is particularly noteworthy. Its core functionality lies in the encoding and decoding of LoRa/LoRaWAN packets, with a strong emphasis on encryption and decryption at the transport/MAC level. This capability is crucial for ensuring data integrity and confidentiality in IoT applications, a necessity in sensitive deployments like smart cities and industrial IoT.

Beyond its primary function, 'lora-packet' is instrumental in providing in-depth analysis of packet components, aiding in troubleshooting and enhancing understanding of network data flows. This feature is invaluable in both the development and operational phases of LoRaWAN projects, facilitating robust debugging, testing, and network management.

The library's alignment with current LoRaWAN standards further amplifies its utility across various devices and gateways, making it an adaptable tool in the evolving IoT landscape. In application servers, it streamlines data packet processing, a critical aspect of effective IoT data management.

Overall, 'lora-packet' emerges as a key component in the toolkit for LoRaWAN communication, offering multifaceted functionalities essential for secure, efficient network operation and management, particularly in scenarios demanding high security and detailed packet analysis. This toolkit will be used only for the LoRa packet construction: frame loss calculation, packet sending and the whole process of simulation will be instead implemented and discussed in this thesis.

The burgeoning need for effective simulation of LoRaWAN networks has given rise to tools like LoRaSim [14], which is pivotal in forecasting network performance before actual deployment. LoRaSim's core functionality lies in its ability to evaluate the ratio of successfully received messages to those transmitted, offering a crucial insight into network reliability over a defined period. Additionally, it estimates network energy consumption, an essential factor in the deployment of energy-efficient IoT networks. The tool is versatile, accommodating scenarios with single or multiple gateways. Key findings from LoRaSim indicate that a single gateway, operating at a spreading factor of 12, can feasibly support approximately 120 nodes, assuming each node sends 20-byte messages every 16.7 minutes. Furthermore, studies on collision rates within loaded LoRaWAN networks reveal insights akin to the maximum efficiency observed in pure ALOHA systems, with an efficiency peak of 18% at a link load of 0.48. However, this comes with a notable caveat of a high packet drop rate, reaching around 60%, highlighting critical considerations for network planning and scalability. LoRaSim, however, is a tool for collisions and scalability analysis, this wasn't enough for my research. In order to simulate and analyze the whole process of packet generation and the entire environment in which the packets are sent I was looking for a more versatile tool that could be used as a library and adapted to my needs, and this wasn't possible with LoRaSim. For this reason this tool was discarded and I decided to start with "lora-packet" instead.

As we navigate the multifaceted domain of LoRaWAN technologies, it becomes evident that the research and development in this field are dynamic and encompass a broad spectrum of approaches and tools, each contributing uniquely to the advancement of IoT networks. The exploration of various methodologies, from the practical applications of LoRaWAN in different environmental conditions to the nuanced intricacies of data processing and workflow orchestration, reflects the depth and diversity of this research area.

The collective insights from these works underscore the complexity of designing and implementing effective LoRaWAN systems. They highlight the critical balance between operational efficiency, scalability, and security – factors that are paramount in the increasingly connected landscape of IoT. The challenges identified, such as the need for more specialized tools for specific tasks like packet testing and simulation, pave the way for future research. They call for innovation in creating solutions that are not only technically robust but also align with the evolving demands of IoT applications.

In conclusion, the existing body of work in LoRaWAN research provides a solid foundation and a rich source of knowledge for ongoing and future endeavors in this field. It underlines the importance of continuous exploration and adaptation as the IoT ecosystem expands and diversifies. The advancements in LoRaWAN technologies not only hold the potential to revolutionize the way we interact with and manage our environment but also open up new avenues for research that could further enhance the capabilities and applications of IoT networks.

Chapter 3

Technologies Adopted

3.1 LoRaWAN 1.0.4

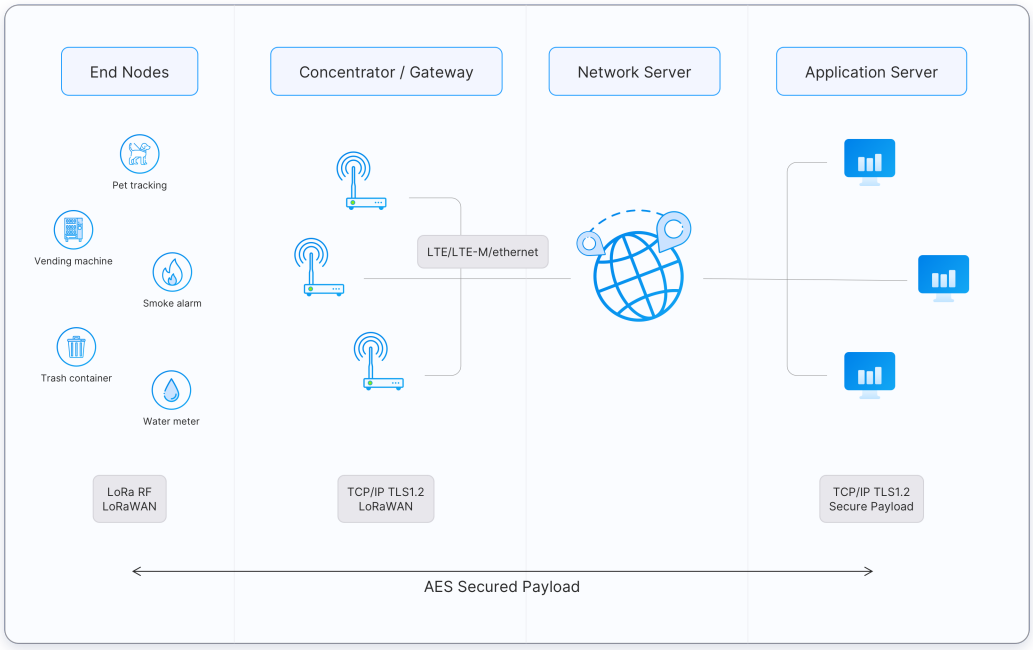


Figure 3.1. A typical LoRaWAN Architecture [2]

Derived from the official LoRaWAN 1.0.4 Specification by the LoRa Alliance [4], LoRaWAN network architecture is fundamentally a star-topology system. It involves sensors (or end-devices) transmitting data packets to a gateway, which then relays these packets to a Network Server. The Network Server, in turn, forwards the packets to an Application Server tailored for specific applications. While the gateway-server communication employs standard TCP/IP internet protocols, the sensor-gateway communication utilizes LoRa technology.

LoRaWAN categorizes devices into three classes, each with distinct communication behaviors:

- **Class A:** This class follows a pattern where every uplink transmission from the device is succeeded by two downlink receiving windows. Consequently, the

devices in this class are only receptive during these specific periods.

- **Class B:** Building upon Class A's structure, Class B devices have the added capability of opening extra downlink receiving windows at scheduled times. This scheduling is synchronized through beacons emanating from the gateways.
- **Class C:** These devices maintain an almost continuous receiving window, closing it only when transmitting data.

Energy consumption varies across these classes, with Class C being the most energy-intensive and Class A being the most efficient, thereby offering extended battery life. LoRaWAN operates on unlicensed radio frequencies, which are region-specific. For example, in Europe, it uses the 863-870MHz ISM Band.

3.1.1 Communication Mechanism for class A devices

Class A devices follow a unique communication protocol that prioritizes low energy consumption. The mechanism is based on an asynchronous transmission model:

1. **Uplink Transmission:** A Class A device initiates communication by transmitting an uplink data packet to the gateway. This transmission occurs based on the device's internal schedule or event triggers, without the need to synchronize with the gateway.
2. **Receiving Windows:** Following each uplink transmission, the device opens two short receiving windows. The first window opens exactly 1 second after the uplink transmission, and the second window opens 2 seconds after the uplink. These windows are the only periods during which the device can receive downlink messages from the network.

Advantages: The primary advantage of Class A devices is their minimal energy consumption. Since the receiving windows are brief and occur only immediately after an uplink transmission, the device spends most of its time in a low-power idle state. This feature significantly extends the battery life of the device, making it suitable for long-term deployments in remote or inaccessible locations.

Limitations: However, this communication model also imposes certain limitations. Class A devices cannot receive downlink messages outside of their receiving windows, meaning that real-time or on-demand communication from the network to the device is not feasible.

Applications: Given their low power usage, Class A devices are widely used in various IoT applications, including environmental monitoring, smart agriculture, and asset tracking.

In the next section, we delve into the specifics of uplink communication for Class A LoRa devices, highlighting the protocol's nuances and operational efficiency.

3.1.2 Uplink communication

This section delineates the messaging framework for Class A LoRa devices, which are characterized by their energy-efficient communication protocol.

Physical Message Format

The communication uses the LoRa radio packet in explicit mode, comprising various components to ensure payload integrity.

Preamble	PHDR	PHDR_CRC	PHYPayload	CRC
n bits	1 byte	1-2 bytes	1-255 bytes	4 bytes

Figure 3.2. Radio PHY structure (CRC is only available on uplink messages)

- **Preamble**: It's the initial part of the message, used for synchronization and to indicate an incoming transmission, configurable length, typically 6 to 65535 bits.
- **PHDR** (Physical Header): Contains information about the packet's properties (spreading factor, coding rate, bandwidth, CRC presence), fixed-size, typically 1 byte (8 bits).
- **PHDR_CRC** (Physical Header CRC): cyclic redundancy check for the PHDR, ensuring its integrity 1-2 bytes (8-16 bits).
- **PHYPayload**: The actual data being transmitted, which includes both the MAC header and payload, varies significantly, ranging from 1 byte to 255 bytes: the max payload size varies depending on the spreading factor.
- **CRC**: A cyclic redundancy check for the entire uplink message, ensuring overall data integrity, typically 4 bytes (32 bits).

MAC Message Formats

All LoRa uplink and downlink messages carry a PHY payload starting with a single-octet MAC header (MHDR), followed by a MAC payload and ending with a 4-octet message integrity code (MIC).

PHYPayload: LoRa messages include a PHY payload with a MAC header, payload, and integrity code.

MHDR	MACPayload	MIC
1 byte	0-255 bytes	4 bytes

Figure 3.3. PHY payload structure

- **MHDR** (MAC Header): A single-octet field defining the message type (e.g., join request, data message) and its format, 1 byte (8 bits).
- **MACPayload**: The variable-length field carrying the actual data or MAC commands, 0 to 255 bytes.
- **MIC** (Message Integrity Code): A 4-octet field used for message authentication and integrity, 4 bytes (32 bits).

MACPayload: The MAC payload consists of a frame header, port, and frame payload.

FHDR	FPort	FRMPayload
7-23 bytes	1 byte	0-242 bytes

Figure 3.4. MAC payload structure

- FHDR (Frame Header): Contains addressing and control information. Its size varies based on the fields included, 7 to 23 bytes (56 to 184 bits).
- FPort: A single byte identifying the application port, 1 byte (8 bits).
- FRMPayload: The application-specific data. Its size can vary greatly depending on the application's needs, 0 to 242 bytes.

FHDR: The frame header includes device address and control information.

DevAddr	FCtrl	FCnt	FOpts
4 bytes	1 byte	2 bytes	0-14 bytes

Figure 3.5. Frame header structure

- **DevAddr**: A device address field, identifying the end device, 4 bytes (32 bits).
- **FCtrl** (Frame Control): Contains control flags and options, typically 1 byte (8 bits).
- **FCnt** (Frame Counter): Used for keeping track of frame counts for security, 2 bytes (16 bits).
- **FOpts**: Optional field for MAC commands, variable in size, 0 to 15 bytes (0 to 120 bits).

The message format for Class A LoRa devices is designed for energy efficiency and flexibility. The sizes of components like the MACPayload and FRMPayload are highly variable, supporting a wide range of applications. While some fields have fixed sizes, others are adjustable, reflecting the diverse requirements of different applications.

3.1.3 Activation Methods

LoRaWAN supports two primary activation methods to secure communication between the end-device and the Join/Application/Network Server:

1. **Activation By Personalization (ABP):** This method is straightforward but offers less security. Activating an end-device by personalization involves directly storing the DevAddr and the two session keys, NwkSKey and AppSKey, in the end-device rather than deriving them from DevEUI, JoinEUI, and AppKey. As a result, the device comes ready to join a specific LoRaWAN network upon startup. Each device must have a unique set of NwkSKey and AppSKey values to maintain security. The keys for one device must be secured in such a way that they do not compromise the communications of other devices, ensuring they cannot be inferred from public information like the device address or DevEUI.

2. **Over The Air Activation (OTAA):** OTAA enables the end-device and the Join Server to mutually establish session and integrity keys through a handshake mechanism. Despite being more secure due to its ability to refresh these keys, OTAA is typically more complex and slower in setup compared to ABP.

For development and testing phases, ABP is often preferred due to its simplicity and speed. However, OTAA is recommended for operational deployments due to its enhanced security features. For our testing purposes, especially since our experiments are focused predominantly on uplink messages, ABP is the chosen method. Its straightforward nature makes it suitable for performance testing in our context, without the need for the advanced security measures that OTAA offers.

3.2 Packet Forwarder

In the architecture of LoRaWAN, the role of the Packet Forwarder is both crucial and multifaceted. Operating on the LoRaWAN gateway, it serves as an intermediary, managing the transmission and reception of LoRa packets between end devices and the network server. A Packet Forwarder is a program that interacts:

1. with the LoRa chip, to receive and transmits LoRa packets.
2. with the network server, to forward the received LoRa packets for processing and to receive downlink messages intended for end devices.

Handling of LoRaWAN Packets

- **Demodulation and Decoding:** The Packet Forwarder demodulates the radio signal from the end device and decodes it into a digital packet format, converting the CSS modulated signal into baseband digital data.
- **Packet Inspection and Processing:** It inspects the packet structure, including headers, payload, and MIC, validating the packet and checking for errors.
- **Metadata Addition:** Metadata such as RSSI, SNR, gateway identification, and timestamp are attached to the packet.
- **JSON Serialization:** The Packet Forwarder serializes the packet data and metadata into JSON format with the following fields:
  - *"tmst"*: Current timestamp in milliseconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).
  - *"time"*: The packet's UTC time in ISO 8601 format, if GPS time reference is available.
  - *"tmms"*: GPS time in milliseconds since 06 Jan 1980, if GPS time reference is available.
  - *"chan"*: the concentrator channel within the Intermediate Frequency (IF) chain, where the packet was received.
  - *"rfch"*: the Radio Frequency (RF) chain on which the packet was received.
  - *"freq"*: The center frequency of the received packet in MHz.

  - *"stat"*: Packet status indicating CRC validation (1 for OK, -1 for fail, 0 for no CRC).
  - *"modu"*: Modulation technique (e.g., "LORA" or "FSK").
  - *"datr"*: Data rate, represented differently for LoRa (e.g., "SF7BW125") and FSK modulations.
  - *"codr"*: Error Correction Coding (ECC) coding rate for LoRa packets.
  - *"lsnr"*: This stands for "LoRa Signal-to-Noise Ratio" and represents the signal-to-noise ratio (SNR) in decibels (dB) specific to LoRa communication..
  - *"rssi"*: Received Signal Strength Indicator. It is a measurement of the strength of the received signal, expressed in dBm.
  - *"size"*: Payload size in bytes.
  - *"data"*: Base64 encoded payload data.

- **Packet Conversion for Network Server:** The JSON serialized packet is encapsulated in a UDP packet for transmission to the Network Server, with a UDP header containing source and destination ports, packet length, and checksum.

3.2.1 Security and Reliability Enhancements

To enhance the reliability and security of the communications:

- Tools like *ChirpStack Gateway Bridge* [1] are used to implement more reliable protocols such as MQTT over TCP.
- These tools support SSL/TLS encryption and authentication for secure data transmission.

3.3 LoRaWAN Servers

After a LoRa packet is transmitted by the Packet Forwarder, it enters the Things Stack, a comprehensive system in LoRaWAN architecture responsible for intricate network management and data processing.

3.3.1 Network Server: Initial Packet Processing

Upon receiving the packet, the Network Server undertakes several critical operations:

- **MIC Verification:** The server first verifies the Message Integrity Code (MIC) to confirm the packet's authenticity and integrity.
- **Duplicate Packet Filtering:** Since LoRa packets may be received by multiple gateways, the Network Server employs algorithms to identify and eliminate duplicate packets.
- **Rate Optimization (ADR):** For devices using Adaptive Data Rate (ADR), the server assesses the metadata (RSSI, SNR) to adjust the data rate and transmission power dynamically, enhancing network efficiency and device battery life.

3.3.2 Join Procedure and Device Session Management

When a device initiates a join procedure, the Join Server manages the following:

- **Join-Request Processing:** The server evaluates Join-Requests, checking device credentials and generating Join-Accepts with unique session keys (Network Session Key and Application Session Key) for secure communications.
- **Session Context Management:** It maintains a session context for each device, tracking parameters like frame counters and channel configurations, crucial for ongoing communication.

3.3.3 Application Server: Advanced Data Handling

The Application Server, upon receiving data from the Network Server, performs the following tasks:

- **Payload Decryption and Decoding:** Using the Application Session Key, the server decrypts and decodes the FRMPayload to extract application-specific data.
- **Application Data Routing:** It routes the decoded data to designated applications or endpoints, employing services such as MQTT, HTTP, or custom APIs for integration.
- **Downlink Communication Management:** The server orchestrates the downlink queue, scheduling and prioritizing data or commands to be sent back to the devices, accounting for network constraints and device communication classes (Class A, B, or C).

3.3.4 Ensuring Security

The Things Stack incorporates several mechanisms to bolster security and reliability:

- **End-to-End Payload Encryption:** Leveraging the Application Session Key, payload data remains encrypted throughout the network, from the device to the application server. This level of encryption is a fundamental aspect defined by the LoRaWAN protocol, ensuring secure communication across all LoRaWAN servers.
- **Secure Transport Protocols:** Communication between the Network Server and Application Server within The Things Stack can be secured using TLS/SSL, ensuring data protection during transmission. This specific security feature is a distinctive attribute of The Things Stack, enhancing the overall security posture of the system.

3.4 Edge2LoRa

The Edge2LoRa [5] architecture represents a significant enhancement to the traditional LoRaWAN system. It is specifically designed to integrate edge processing capabilities, addressing the scalability challenges inherent in the conventional LoRaWAN architecture.

3.4.1 Key Components

- **Device Registry:** Manages device information, which is critical for ensuring secure and efficient communication within the network.
- **Gateway Selection Algorithm:** Optimizes the usage of gateways to balance load and improve network efficiency.
- **Group Key Establishment:** Plays a vital role in enhancing the security of the system, which is essential for maintaining data confidentiality and integrity.
- **Data Processing at Edge-to-Gateway (E2GWs):** Enables localized data processing, which significantly reduces the need for data transmission to distant servers, resulting in lower latency and reduced bandwidth requirements.

3.4.2 Benefits

Edge2LoRa maintains compatibility with traditional LoRaWAN while bringing significant performance improvements, especially in terms of latency and scalability. Its implementation in real-world situations highlights these benefits. Edge2LoRa stands out as a smart and effective choice in LoRaWAN technology. It's important to note that my work played a crucial role in proving these improvements and making them practical.

Chapter 4

Implementation

4.1 Architecture of the system

In my thesis work, before even start modelling a solution for the implementation of the simulator my first thought was about reproducing the architecture of a Standard and an Edge2LoRa Network. As you can see from the figure 4.1, LoRaWAN communication happens between three distinct layers.

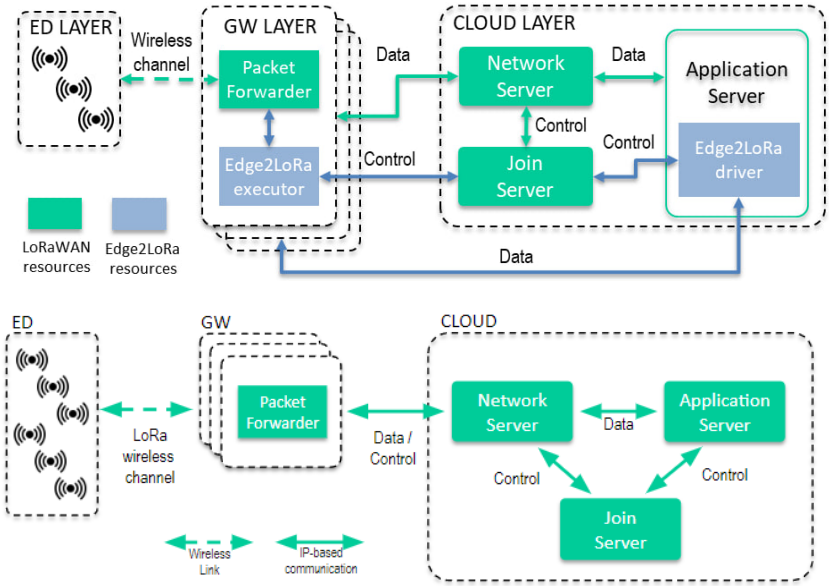


Figure 4.1. First figure shows the architecture of an Edge2LoRa Network, the figure below represents the architecture of a Standard LoRaWAN.

4.1.1 End Device Layer

End Device (ED): The end device initiates communication by sending uplink data packets, which typically contain sensor readings or status updates. The transmission uses the LoRa modulation technique, which is optimized for long-range and low-power operations.

4.1.2 Gateway Layer

Packet Forwarder: The Packet Forwarder program manages the transmission and reception of LoRa radio packets. It captures the data transmitted by end devices and forwards it to the network server using standard network protocols like IP/UDP. The Packet Forwarder also handles the downlink queue for messages to be sent to end devices. If the GW operates with the enhanced version (E2GW), it can process these frames locally before passing them on to the NS/AS. Conversely, frames from an E2ED received by a standard GW, or those from a standard ED received by an E2GW, are forwarded to the NS/AS as per the conventional LoRaWAN network standards.

E2L Executor: The Edge2LoRa executor module deployed at the E2GW is interconnected with the standard Packet Forwarder[1] through a proxy component implemented in Rust. The proxy intercepts traffic from the Packet Forwarder and redirects it to the Edge2LoRa executor module after it has been decrypted by the Edge2LoRa Key Agreement module executed within the E2GW. The communication over the RPC Protocol is implemented using the *tonic* create[2], which is a production-ready implementation of *gRPC*[3] for Rust.

4.1.3 Cloud Layer

Network Server (NS): Upon receiving the data from the gateway, the network server conducts various management tasks. It removes duplicate packets, schedules acknowledgments, and manages the adaptive data rate (ADR) to optimize the communication parameters for each end device.

Join Server (JS): In the device authentication phase, the end device's join request is passed from the gateway to the network server and then to the join server. After securely authenticating the device, the join server sends a join accept message back to the end device via the network server and gateway.

Application Server (AS): The network server processes the received packets and forwards the application payload to the application server. This server is responsible for making sense of the received data and leveraging it within the end-user applications, enabling actions and responses based on the data.

Edge2LoRa Driver (E2L AS) The Edge2LoRa AS is an extension of the AS offered by the Thing Stack. It is implemented using Python3 and it interfaces with Thing Stack using the MQTT integration that the latter offers. Using the MQTT integration the Edge2LoRa AS extension can receive the uplink LoRaWAN packets and schedule downlink ones. The MQTT interface is implemented with the Eclipse Paho library[4]. The communication between the AS and the E2GW exploit the RPC Protocol, implemented using the *gRPC Python3 SDK*[5].

[1] https://github.com/Lora-net
[2] https://docs.rs/tonic/latest/tonic/
[3] https://grpc.io/
[4] https://eclipse.dev/paho/
[5] https://grpc.github.io/grpc/python/

4.1.4 First Real Data Implementation

This subsection outlines the first practical implementation of the LoRaWAN Network for my experiments, detailing the specific hardware and software components utilized to simulate each layer of the LoRaWAN communication process.

End Device Layer Implementation

- **End Device (ED):** The Heltec CubeCell module, specifically the SX1262 model, was selected as the end device. Its long-range and low-power operational capabilities make it ideal for LoRa communication.

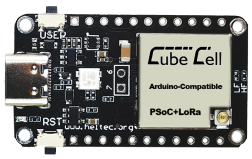
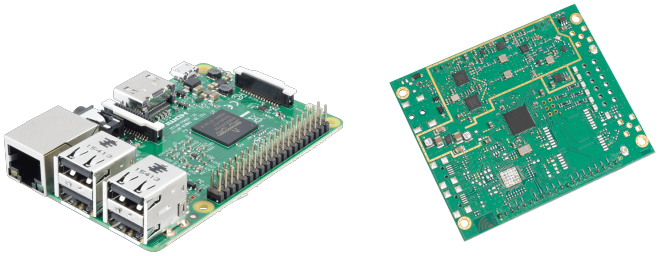


Figure 4.2. Heltec CubeCell A1 SX1262

  - **Software Development:** The CubeCell module was programmed using an Arduino-compatible environment provided by the *HelTecAutomation/CubeCell-Arduino* repository [7]. This framework enabled efficient development for the CubeCell to send/receive uplink/downlink data packets, which in my tests simulated sensor readings or status updates.

Gateway Layer Implementation

- **Gateway (GW) and Packet Forwarder:**



(a) (b)

Figure 4.3. (a) Raspberry Pi 3, (b) IC880A-SPI Module

  - **Hardware:** A Raspberry Pi 3, 1 GB RAM, equipped with an IC880A-SPI LoRaWAN concentrator module, was used as the gateway and the parts were assembled following this guide provided by The Things Stack [16]
  - **Software:** The Raspberry Pi ran a Linux-based OS(Raspberry Pi OS, Kernel version 6.1) and was installed with the packet forwarder program from the *Lora-net/packet_forwarder* GitHub repository [13]. This software managed the LoRa radio packet transmissions between the end device and the network server.

Cloud Layer Implementation

- **Network Server (NS), Join Server (JS), and Application Server (AS):** Managed using a MacBook Air 2020 with an M1 chip, leveraging its advanced processing capabilities with a running version of The Things Stack(TTS) v3.26.

This setup, integrating the Heltec CubeCell with software from the ‘CubeCell-Arduino‘ environment, the Raspberry Pi 3 with an IC880A-SPI Module, and the MacBook Air 2020, provided a comprehensive framework for my initial testing phase. This phase focused on transmitting real packets to The Things Stack, facilitating a deep understanding of the LoRaWAN communication process and the information flow at each communication step.

4.2 Real Device Simulation

The `packetGenerator` function plays a crucial role in the LoRaWAN network simulation, focusing on the accurate creation and formatting of LoRaWAN packets. This function ensures a detailed and realistic emulation of packet structures and transmission dynamics, vital for the integrity of the simulation.

In contrast, the `simulateDevice` function is designed to replicate the behavior of individual LoRaWAN devices within the network. It encompasses the entire lifecycle of a device, from activation to deactivation, effectively mimicking the diverse and unpredictable nature of real-world device operations in a LoRaWAN network.

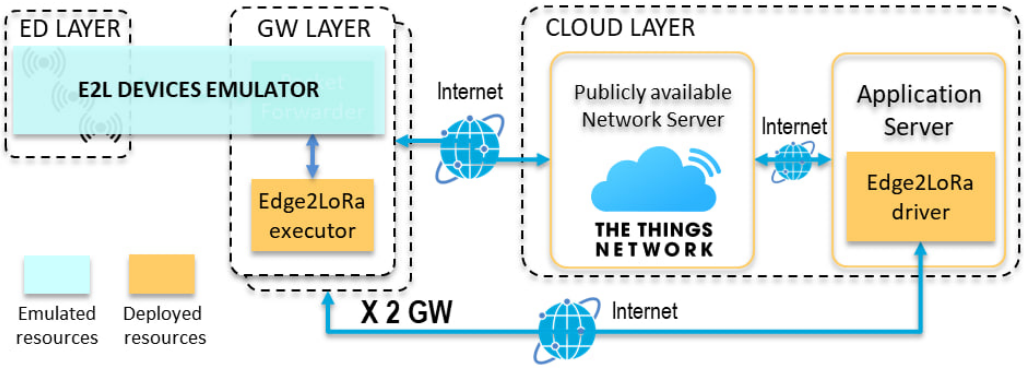


Figure 4.4. Architecture of the simulated environment

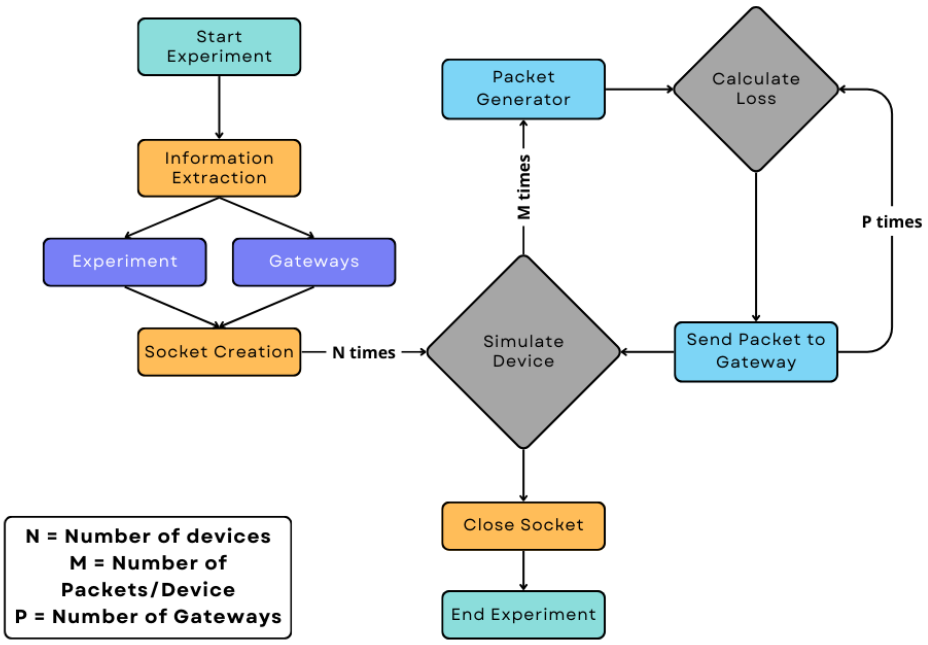


Figure 4.5. Flow Diagram of the Device Simulation Tool

4.2.1 Experiment Configuration

The `experiment.json` file contains the configuration for the simulation. It includes various parameters that define the behavior of the simulated devices and the network. Key parameters include:

- **ratio**: Defines the ratio of different types of devices in the simulation. The rule is "ratio = n => 1 standard device every n edge devices".
- **sleepTimer**: The time interval between packet transmissions for a device.
- **deviceTimer**: The delay before the start of each device's operation.
- **deviceNumber**: The total number of devices to be simulated.
- **minPacket** and **maxPacket**: Define the range for the number of packets each device will send.
- **frameLoss**: This can be used only if we wanna force a static frameLoss in the network, otherwise we can use the function described before
- **deviceList**: The path to the JSON file containing the list of devices and their configurations.

Each of these parameters can be easily modified to set up different experimental scenarios, enabling researchers to study various aspects of LoRaWAN network behavior under different conditions.

4.2.2 Gateway Configuration

The `gateways.json` file specifies the configuration for network gateways. This file includes information such as the host and port for each gateway, allowing the simulation to adapt to different network topologies and gateway configurations. An example configuration is:

- **host**: The hostname or IP address of the gateway.
- **port**: The port number on which the gateway is listening.

4.2.3 Objective

The main goal of my simulator is to facilitate comprehensive testing within a LoRaWAN ecosystem by simulating real-world device interactions with both standard and Edge2LoRa gateways, as well as cloud layers. This simulator aims to inject authentic LoRa packets into the network, enabling extensive validation of network configurations, device behaviors, and system integrations. While the inclusion of Message Integrity Code (MIC) validation serves to ensure the fidelity and security of the packets throughout the communication process, it is a supplementary feature that underscores the simulator's capability to maintain packet integrity. The primary objective remains to provide a holistic testing environment that mirrors actual operational conditions, thereby enabling a thorough evaluation of LoRaWAN networks and their components.

4.2.4 Language and Version

The simulation is developed in JavaScript, utilizing Node.js version 10. This choice is predicated on Node.js's non-blocking I/O model, which is crucial for simulating the asynchronous and concurrent operations characteristic of network communications. The use of version 10 ensures compatibility with the essential features required for the simulation, including but not limited to, asynchronous functions, network socket operations, and integration with necessary modules like `dgram` for UDP communication and the `lora-packet` library for precise packet encoding and decoding.

4.2.5 GitHub Repository

The entire codebase for the Real Device Simulation is hosted on GitHub[6]. This repository is meticulously organized to ensure ease of navigation and understanding. The repository is structured to clearly separate the core components of the simulation, such as packet generation (`packetGenerator.js`), device simulation logic within the main script, and configuration files for network and device parameters (`experiment.json` and `gateways.json`). This open-source resource invites collaboration, feedback, and contributions from the community, aiming to foster innovation and advancement in the simulation and testing of LoRaWAN networks.

4.2.6 Functionality of `packetGenerator`

The `packetGenerator` function is responsible for creating and formatting LoRaWAN packets, which are essential for the simulation of the network. The key components of this function are:

LoRa Packet Construction

Fields Setup

- **FPort**: Specifies the application port. Used 4 for edge devices and 2 for standard devices, indicating different types of devices in the simulation.
- **MType**: Set to "Unconfirmed Data Up", indicating an uplink message type that does not require acknowledgment from the server.
- **DevAddr**: Device address in hexadecimal, representing the source of the packet.
- **FCtrl**: Control field with flags like ADR (adaptive data rate), ACK (acknowledgment), etc., all set to false.
- **FCnt**: Frame counter which increments with each packet sent.
- **Payload**: Set to the current timestamp, simulating a data payload.

Packet Generation

The packet is generated with the `lora_packet.fromFields` method, incorporating the above fields and the keys `AppSKey` and `NwkSKey` which are enough since the activation of every simulated device has been done in ABP mode.

[6] https://github.com/Edge2LoRa/e2l-device-simulation

Base64 Encoding of Packet

The packet is then converted to a base64-encoded string using:

```
const payloadBase64 = constructedPacket.getPHYPayload().toString("base64");
```

This is a standard way to encode binary data for network transmission.

JSON Packet Structure

A JSON object is constructed with `rxpk` array, replicating the one used by the PF:

- **tmst**: Current timestamp in milliseconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).
- **chan**: Channel index (IF chain) on which the packet is received.
- **rfch**: RF chain index on which the packet is received, usually 0 or 1 in multi-chain gateways.
- **freq**: Frequency in MHz at which the packet was received.
- **stat**: Status of the packet (1 for CRC OK, -1 for CRC fail, 0 for no CRC).
- **modu**: Modulation technique used (e.g., 'LORA').
- **datr**: Data rate, represented as 'SFx BWxxx' for LoRa (e.g., 'SF7BW125').
- **codr**: ECC coding rate for LoRa packets (e.g., '4/5').
- **lsnr**: Signal-to-noise ratio in dB, a measure of signal quality.
- **rssi**: Received Signal Strength Indicator in dBm, a measure of signal power.
- **size**: Size of the payload data in bytes.
- **data**: The actual base64-encoded packet data.

Custom Header for Packet Forwarder

```
let headerPKTFWD = new Uint8Array([2, 45, 141, 0, 184, 39, 235, 255, 254, 230, 15, 44]);
```

A custom header (`headerPKTFWD`) is created, which includes:

- Protocol version (2 in this case).
- Two Random numbers for identification.
- Packet type identifier (0 for PKT_PUSH_DATA).
- LoRa gateway MAC address.

This header mimics the structure of a packet as handled by the packet forwarder.

Final Packet Assembly

Combines the custom header and the JSON packet into a single data structure and the function returns the packet encoded as bytes.

4.2.7 Functionality of `simulateDevice`

Initialization and Device Activation

The function starts by incrementing the `activeDevices` counter. This step is crucial as it simulates the activation of a LoRaWAN device in the network, impacting network conditions like channel occupancy and frame loss rate.

Randomized Start Time

The device begins its operation after a randomized sleep period. This randomness in starting times for each device is significant as it mimics the unsynchronized and independent behavior of real-world IoT devices in a LoRaWAN network.

Packet Transmission Loop

The core of the device simulation is the loop where packets are generated and sent (Packet Generation, Frame Loss Calculation, Conditional Packet Sending).

Packet Generation

The `packetGenerator` is called to create a new LoRa packet. The parameters passed (like `DevAddr`, `AppSKey`, `NwkSKey`, `FPort`, and `FCnt`) are vital as they ensure each packet accurately reflects the unique identity and state of the device. The `FCnt` (frame counter) is particularly noteworthy. It's a critical component in LoRaWAN's security and replay attack protection mechanism. Incrementing `FCnt` for each packet simulates this aspect.

Frame Loss Calculation

In the context of LoRaWAN (Long Range Wide Area Network), a critical challenge is ensuring efficient communication amidst network complexities such as congestion, interference, and channel capture. The paper "Capture Aware Sequential Waterfilling for LoRaWAN Adaptive Data Rate [6]" delves into these aspects, emphasizing the adaptive data rate mechanisms and the significant impact of channel capture on network performance. In this part we go through the analysis of frame loss, which serves as a key indicator of the network's capability to manage real-world challenges. This subsection elaborates on the methodology used in the paper for computing frame loss, with a focus on the underlying network conditions.

1. **Packet Generation Rate** (s): The rate at which packets are generated in the network, defined as:
$$s = \frac{1}{90}\,\text{pkt/s}$$
This represents the average rate at which each transmitting device generates packets.

2. **Normalized Offered Load** (G): This is a measure of the load offered to the network, normalized for the analysis. It is calculated as:
$$G = \frac{N \times s \times \text{ToA}}{M}$$
where N is the number of transmitting devices, ToA is the time interval required for transmitting a packet, and M is a normalizing constant. This

value represents the aggregated impact of all transmitting devices on the network load.

3. **Signal-to-Interference Ratio** (α): A crucial factor in wireless communications, representing the ratio of the signal power to the interference power, given by:
$$\alpha = 10^{\frac{\text{SIR}}{10 \times \eta}}$$
where SIR is the signal-to-interference ratio in dB and η is a constant. This ratio is pivotal in determining the likelihood of a packet being successfully received despite interference.

4. **Collision Probability** (Sc): The Collision Probability is a fundamental concept in understanding frame loss in LoRaWAN networks. In the realm of LoRa technology, the phenomenon of channel capture significantly influences network scalability. When packets collide, the probability of successful demodulation is impacted by the relative strengths of the colliding signals, a scenario frequently encountered in LoRaWAN environments. This dynamic is particularly pronounced in networks with multiple gateways, where the deployment enhances the probability of channel capture, thereby influencing collision dynamics.

   The formula for Collision Probability encapsulates the interplay between the normalized offered load (G) and the signal-to-interference ratio (α), which are pivotal in determining the behavior of LoRaWAN networks under various load conditions. The formula is expressed as:
$$Sc = \left(\frac{1}{2\alpha^2}\right)\left(1 - e^{-2G}\right) + G\left(1 - \frac{1}{\alpha^2}\right)e^{-2G}$$
   This formula reflects the complex relationship between packet collision likelihood, signal strength, network load, and interference. It is particularly relevant in assessing network behavior in scenarios with dense or congested network conditions, where individual and collective device behaviors crucially impact network performance.

5. **Data Extraction Rate** (DER_CC): This is the rate at which data is successfully extracted from the network, which is influenced by the collision probability. It is defined as:
$$\text{DER\_CC} = \frac{M \times \frac{Sc}{G}}{M}$$
   This rate is crucial in understanding the efficiency of the network in transmitting data under given conditions.

6. **Frame Loss** (loss): Finally, the frame loss, which is a measure of the percentage of frames lost due to collisions and other network issues, is calculated as:
$$\text{loss} = (1 - \text{DER\_CC}) \times 100\%$$
   This metric is essential for assessing the overall performance and reliability of the network.

Conditional Packet Sending

Building on the frame loss calculation, a probabilistic approach is adopted for packet transmission decisions (via the `sendPacketToAllGWs` function). This approach simulates real-world operational conditions where packet transmission success is not guaranteed and is influenced by factors such as interference, signal attenuation, and network congestion. This step reflects the practical realities of network operation in LoRaWAN environments, aligning with the study's focus on adaptive data rate mechanisms and channel capture impact.

Sleep Interval Between Packets

After sending a packet, the device goes into a sleep state for a period defined by `sleepTimer`. This behavior is typical of many LoRaWAN devices, especially those operating on battery power, where energy efficiency is crucial.

Simulation Completion and Device Deactivation

Once the device has sent the designated number of packets (`nPackets`), it simulates the deactivation by decrementing the `activeDevices` counter. This final step is significant as it represents the device going offline or becoming inactive, which would affect network conditions and potentially the performance of other devices.

Configurability and Scalability

The use of JSON files for experiment configuration provides significant advantages:

- **Ease of Use**: Researchers can easily set up and modify experiment parameters without delving into the core simulation code.
- **Reproducibility**: Experiments can be reliably replicated by using the same JSON configuration files.
- **Scalability**: The simulator can handle a wide range of scenarios, from small-scale tests to large-scale network simulations.

Chapter 5

Performance Evaluation

5.1 Introduction to Performance Evaluation

This section offers an in-depth analysis of diverse performance aspects using a real-world implementation of Edge2LoRa. We embark on an exploration of the deployment of the Edge2LoRa architecture [5], incorporating all the elements as depicted in Fig. 4.1. The objective of those tests is twofold: firstly, to elucidate the operational dynamics of Edge2LoRa within an extensive LoRaWAN framework, and secondly, to spotlight the performance improvements achieved by Edge2LoRa in various traffic scenarios, while also considering its impact on conventional LoRaWAN systems.

The Edge2LoRa injection module, fundamental for our research, is specifically designed to closely emulate the network environment of a large-scale LoRaWAN system. This sophisticated tool serves as a comprehensive emulator of LoRaWAN terminals, enabling the creation of detailed testing environments that encompass thousands of terminals. These environments are meticulously controlled, providing a fertile ground for conducting extensive and realistic experiments. The module's ability to simulate diverse network scenarios allows us to rigorously test and validate the performance of the Edge2LoRa system under various conditions, ranging from low to high traffic loads.

Furthermore, this module not only replicates the behavior of individual LoRaWAN terminals but also models their interactions within a network, offering invaluable insights into the system's scalability and reliability. By providing a platform for such in-depth analysis, the Edge2LoRa injection module plays a pivotal role in advancing our understanding of the system's capabilities.

For researchers and practitioners interested in replicating or extending our work, the complete suite of developed modules, along with exhaustive documentation detailing the configurations used for our experiments, is made available in the public domain[1]. This open-source repository is a testament to our commitment to transparency and collaboration in the research community, encouraging further innovation and exploration in the field of LoRaWAN technologies.

In the following sections, we will delve into the specifics of the Edge2LoRa system's architecture, its integration within the broader LoRaWAN environment, and the significant enhancements it brings to network performance and efficiency.

[1] https://github.com/Edge2LoRa

5.2 Description and Overview of System Architecture

The Edge2LoRa injection module, my implemented simulator, is meticulously designed to emulate terminals adhering to LoRaWAN specifications. This modular tool is capable of modeling various types of LoRaWAN EDs, each equipped to transmit diverse payloads as specified by scripts. The emulated terminals dispatch their PhyPayload frames through UDP to Gateways (GWs). These frames are then encapsulated into Semtech UDP frame Forwarder (GWMP) messages, directed either to the designated Network Server (NS) or the E2GW executor, based on the logic programmed into the Edge2LoRa driver.

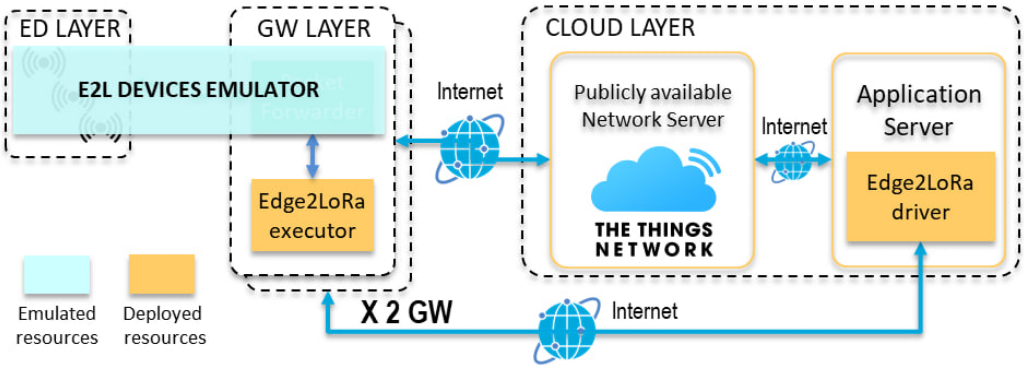


Figure 5.1. Architecture of the simulated environment.

As depicted in Figure 5.1, the architecture has evolved, with the 'Device Simulator' now assuming dual roles, acting as both an End Device (ED) and a Packet Forwarder (PF). This innovative configuration enhances the efficiency of communication between these two elements, streamlining the network's operation within the simulated environment.

Additionally, the Edge2LoRa injection module integrates a sophisticated frame loss model to replicate authentic LoRaWAN network conditions. This model accounts for cell interference and the presence of multiple Gateways (GWs), drawing on methodologies outlined in the referenced paper [6], particularly formula (13).

5.3 Replication of Large-Scale IoT Network Scenarios

The complete list of parameters used for the simulation are provided in Table 5.1.

In this scenario, we simulate an environment with 3000 EDs, each transmitting within a 1 km radius circular cell at a source rate of one frame every three seconds. Each frame is configured to be 24 bytes in size. These EDs are uniformly distributed, and two GWs are strategically placed, equidistant at 0.15 km from the cell's center. This setup results in an ED density of approximately 950 per square kilometer, generating a total of 1000 frames per second.

Given that the software aspects of the simulator were extensively detailed in the previous chapter, our focus here is on the practical implementation of these scenarios and the subsequent performance evaluation of the Edge2LoRa system under these conditions.

In the configured simulated environment for our Edge2LoRa system, End Devices (EDs) are set up in two distinct operational modes: the standard mode for regular

Parameter	Value
Deployment area	circular area of 1km radius
Number of Gateways	2
Gateways deployment	equally spaced at 0.15 km from the center
Number of Devices	3000
Device deployment	uniform
Device activation pattern	progressively activated with an interval of 0.1 sec
Device activation method	ABP
Transmission Data Rate	$DR = 5$
Spreading Factor	$SF = 7$
Frames transmitted	500
Frame transmission rate	1 frame every 3 seconds
Frame size	24 bytes
Frame delivery ratio	formula (3) in [6]
Aggregation window size	30 seconds
Experiment duration	30 minutes

Table 5.1. LoRaWAN scenario simulation parameters for the realistic large-scale scenario.

EDs and an enhanced mode for E2ED. This bifurcated configuration is critical in our assessment, allowing us to conduct a comprehensive evaluation of how the system handles varying device types and responds to diverse network requirements.

The deployment of two E2GW units marks a significant advancement in our network architecture. These units are not mere pass-through gateways; they are equipped with advanced capabilities to execute multiple data aggregation functions. This functionality is crucial in boosting the efficiency of data processing within the network. Each E2GW unit is intricately linked to the Edge2LoRa driver, which plays a pivotal role in intelligently managing network traffic, with a particular focus on efficiently routing data from the E2ED terminals. This process is governed by a set of predefined logic within the Edge2LoRa driver, tailored to optimize network performance and enhance resource utilization.

As part of our study, we have implemented a complex sensor data stream processing task. This task is characterized by the deployment of a window-based many-to-one operator on the E2GW. This setup is key to demonstrating the system's adeptness at managing intricate data operations, further asserting the Edge2LoRa system's capacity for efficient data processing.

5.4 Methodology for Performance Testing

This section outlines the systematic approach and specific criteria we employed for the performance testing of the Edge2LoRa system. Our methodology is designed to ensure a thorough evaluation of the system's efficiency and capabilities.

5.4.1 Testing Criteria and Evaluation Metrics

Our testing methodology is anchored on the following key performance indicators:

1. **Total Frame Count:** This metric assesses the total number of frames transmitted over the IP network, reflecting the network's overall load and

traffic patterns.

2. **Frame Delivery by E2GW:** We measure the efficiency of each E2GW in delivering frames, providing insights into the effectiveness of the network's data distribution and processing.

3. **Computational Resource Utilization:** The system's demand on hardware resources is evaluated by analyzing the CPU and memory usage of the host machines.

These metrics were selected to provide a comprehensive view of the Edge2LoRa system's performance, focusing on aspects crucial for its practical implementation in real-world scenarios.

5.4.2 Testing Environment Setup

The testing environment has been meticulously configured to mirror a real-world LoRaWAN network setting. This involves:

- Deployment of both standard and enhanced EDs.
- Integration of E2GW units with the Edge2LoRa driver.
- Configuration of network parameters to align with specific testing scenarios.

5.5 Laboratory Setup and Deployment

The practical deployment for testing the Edge2LoRa system was conducted in the University of Palermo's laboratory, encompassing the following hardware setup:

- **Host Machines:** Three Intel NUCs, each with an Intel i5-6260U CPU and 8GB of RAM, running Ubuntu 22.04.3 LTS. These machines are allocated for the Edge2LoRa injection module, the Network Server (NS), and the Application Server (AS) with the Edge2LoRa driver.
- **Gateways:** Two Raspberry Pi 4 Model B units, powered by Broadcom BCM2711 SoCs and 4GB of RAM, operating on Debian GNU/Linux 11, serve as the GWs.

This diverse and robust hardware configuration is instrumental in creating a realistic and varied testing environment, mimicking real-world operational conditions as closely as possible.

The data collected in the experiments below is part of our analysis for the different scenarios in the following sections.

With the details given in 5.1, 7 different experiments has been done with 3000 devices, each one being activated every 100ms, sending an incremental number of packets every 3 seconds in a maximum of 500 packets for each device. The table summarizes the different attributes involved in the experiments

Table 5.2. Summary of Experiments

Experiment	Aggregation	Handover	Bandwidth
1. Standard	Standard	False	Unlimited
2. Edge	10	False	Unlimited
3. 50% Std 50% Edge	10	False	Unlimited
4. Edge	1	False	Unlimited
5. Edge	10	True	Unlimited
6. Standard	Standard	False	30 kbps
7. Edge	10	False	30 kbps

5.6 Data Collection and Analysis

This section presents a detailed analysis of the data collected from our tests, in order to have a comparison between the Standard and the Edge2LoRa-enabled environment.

5.6.1 Standard Scenario Analysis

Fig. 5.2 depicts the scenario where the Edge2LoRa system is disabled, showcasing typical LoRaWAN behavior. End Devices (EDs) are sequentially activated at 0.1-second intervals, leading to a 5-minute transition phase until all EDs are operational. Upon activation, each ED transmits 100 frames and then ceases activity. Fig. 5.2a displays the frame count received by each E2GW. With all EDs functioning, the system produces a total of 1000 frames per second. The applied FDR model indicates a 31% Frame Delivery Rate (FDR) at each E2GW, translating to roughly 300 frames per second. This reflects the extent of transmission failures in the wireless medium, primarily due to interference. Such visualizations are crucial to comprehend the load spread across various E2GW in the network.

Included are overlapping frames from two distinct frame groups, identified as duplicates, and frames exclusively received by a single E2GW. These frames, once they reach the AS (labeled as the sink in the diagram), are depicted through the combined probability of frames received at each E2GW, denoted as $P(GW_1)$ and $P(GW_2)$. The cumulative probability, calculated as $P(GW_1 \cup GW_2) = P(GW_1) + P(GW_2) - (P(GW_1) \cap P(GW_2))$, takes into account the intersect probability representing the duplicated frames. This formula calculates the likelihood of frames received at both gateways, ensuring no repeated counting of the overlapping frames in the system. Consequently, the AS receives about 520 frames per second, equivalent to 50% of the total frames sent by the simulated EDs.

In the same test, Fig. 5.2b and Fig. 5.2c illustrate the CPU and memory consumption of the host computers housing the E2GW and the Edge2LoRa driver. The host machine containing the AS shows lower CPU usage, averaging 1.8% compared to 3.2%, attributed to its superior hardware capabilities. On the other hand, this machine exhibits higher memory usage due to its operation on the Ubuntu operating system, unlike the Raspbian system used on the two hosts for the E2GW. Ubuntu's greater demand for resources leads to an increased memory consumption of 1GB, in contrast to the 0.4GB noted on the hosts with the E2GW.

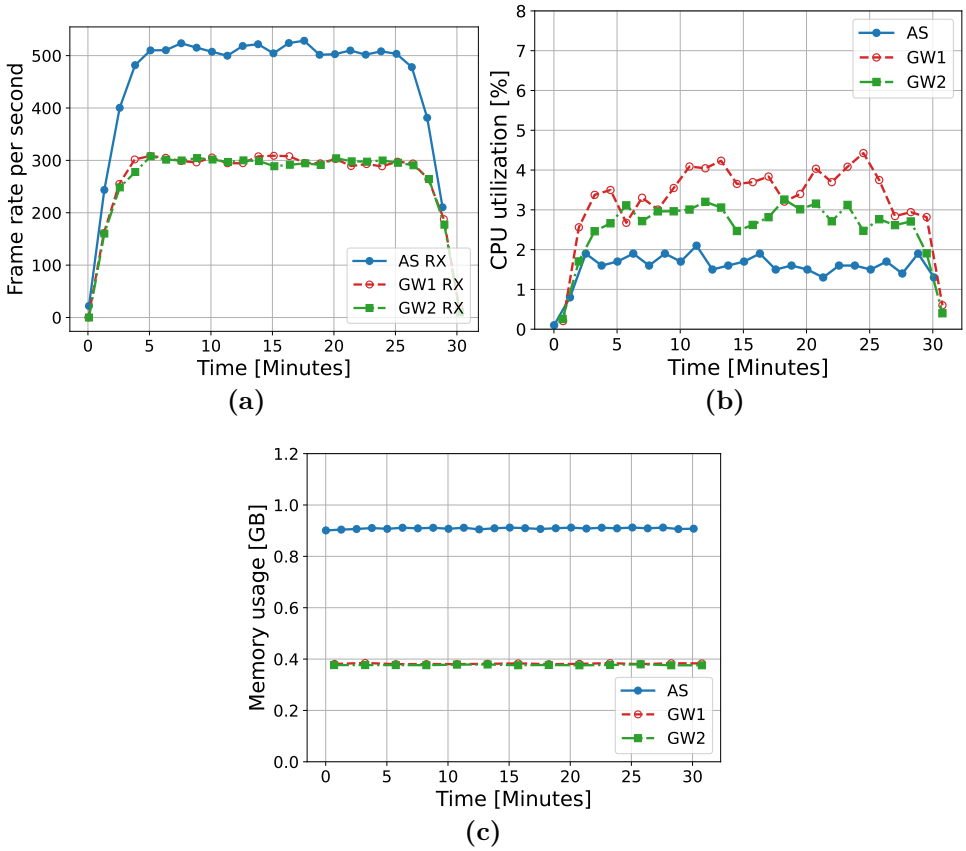


Figure 5.2. Standard scenario performance metrics for E2GW and AS. (a) illustrates the frame rate received per second by each E2GW and by the AS. (b) and (c) detail the CPU and memory usage of the host machines accommodating the E2GW and the AS, providing insight into the system resource utilization in the legacy scenario.

5.6.2 Edge Scenario Analysis

Fig. 5.3 illustrates the performance metrics of the same scenario with the activation of the Edge2LoRa system. Here, the Edge2LoRa driver is set to distribute equal data processing responsibilities between each E2GW. In Fig. 5.3a, the count of frames received by each E2GW is depicted, mirroring the outcomes seen in the legacy scenario. Nevertheless, there's a notable decrease in the number of frames arriving at the sink, a result of the data consolidation function implemented by the driver within the E2GW module executor. For this experiment, the driver's configuration includes a mean function on a 10-frame window in the executors. Consequently, this aggregates 10 incoming frames for each E2ED, producing a single frame encapsulating the average of the values contained in the original frames' payloads.

When juxtaposed with the legacy scenario, the frames reaching the sink are considerably fewer - about 10% of the legacy figure (52 versus 520 on average). This significant drop exemplifies the effectiveness of the aggregation function in saving network resources and minimizing data transmission volumes while retaining crucial data.

Concurrently, Fig. 5.3b and Fig. 5.3c reveal the CPU and memory demands of the hosts for the GWs and the AS, inclusive of the Edge2LoRa driver. In terms of

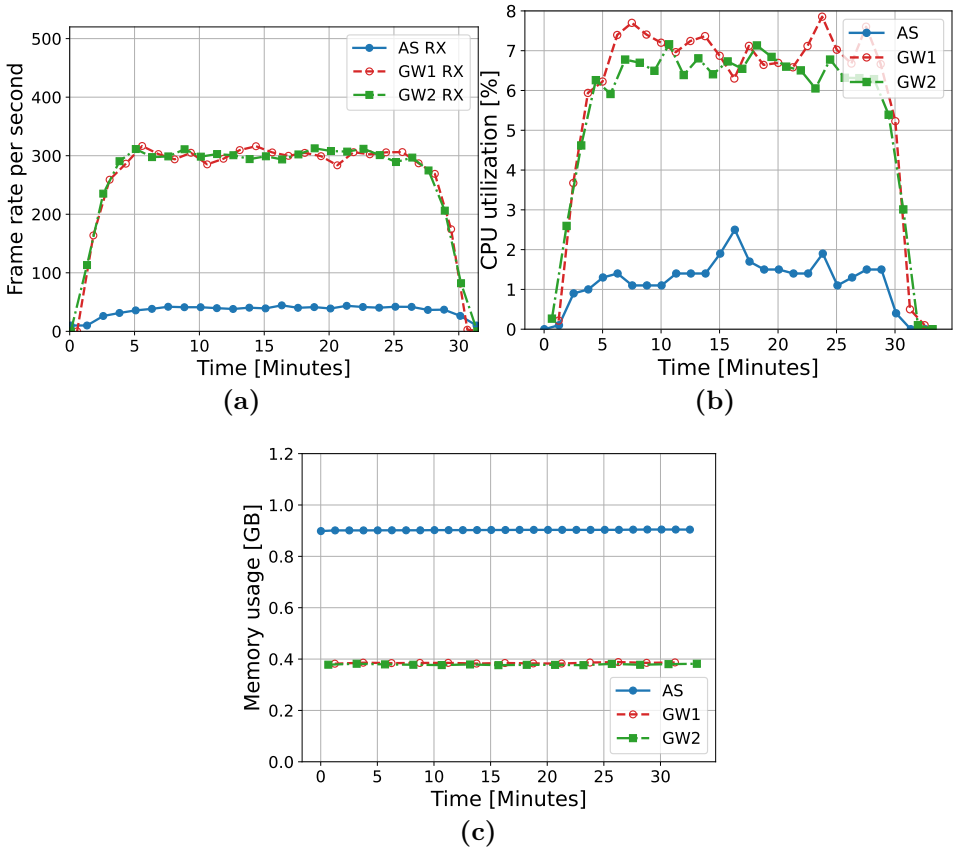


Figure 5.3. Edge scenario performance metrics for E2GW and AS. (a) illustrates the frame rate received per second by each E2GW and by the AS. (b) and (c) detail the CPU and memory usage of the host machines accommodating the E2GW and the AS, providing insight into the system resource utilization in the standard scenario.

CPU utilization, a decrease is seen in the AS's host (averaging 1.2%) and an increase at the E2GW (averaging 6.8%) compared to the legacy setup. This shift is attributed to the lesser frame processing at the AS and the additional computational load from the aggregation function at the E2GW. Additionally, the Edge2LoRa scenario involves the encryption and decryption of each frame, yet this step has a negligible impact.

This transfer of computational tasks from cloud to edge does not lead to a significant surge in CPU consumption. The same holds true for memory utilization, as evidenced in Fig. 5.3c, where memory usage shows no substantial deviation from the legacy scenario.

These insights confirm that the Edge2LoRa system effectively streamlines data aggregation at the edge, lessening the data burden on the AS without markedly escalating resource usage. This finding is vital, underscoring the ability of edge computing to optimize network resource management without overburdening system resources. However, frame statistics at the NS are omitted, as they hold no relevance to this evaluation approach.

5.6.3 Mixed Scenario Analysis

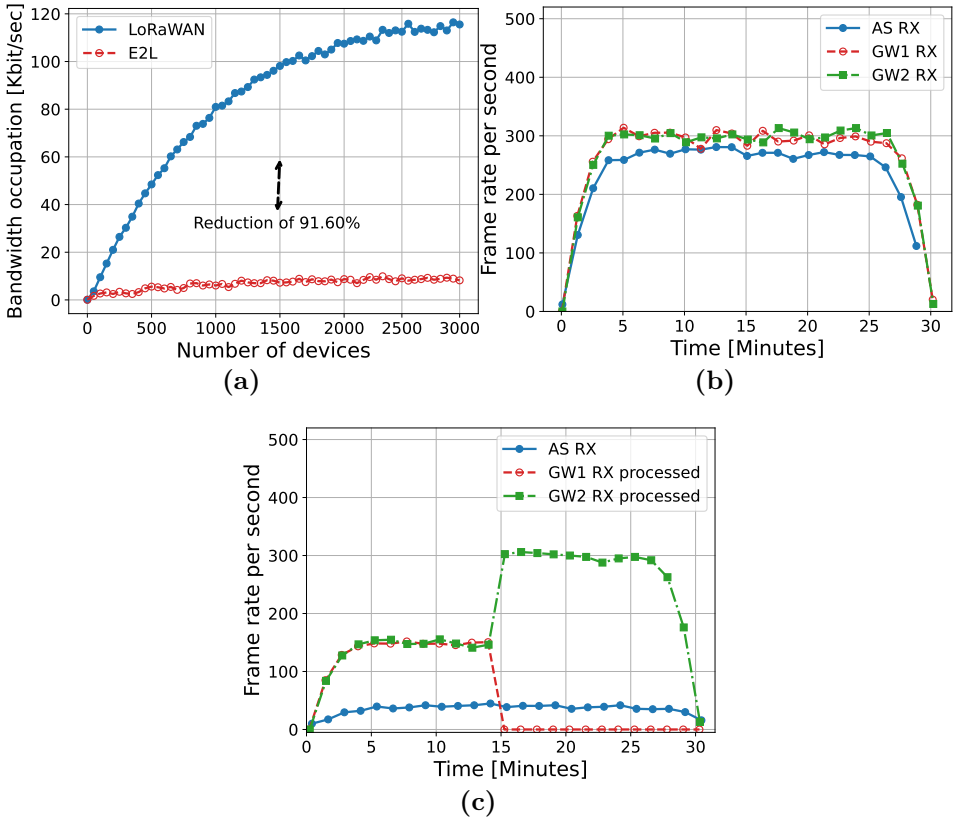


Figure 5.4. Assessment of the benefits for the proposed approach. (a) highlights the bandwidth utilization for both the LoRaWAN standard and Edge2LoRa with a 90% aggregation factor. (b) demonstrates the backwards compatibility of the proposed approach. (c) extends the Edge2LoRa scenario, simulating a failure of E2GW 1 after 15 minutes of activity to showcase the resilience of our proposed method.

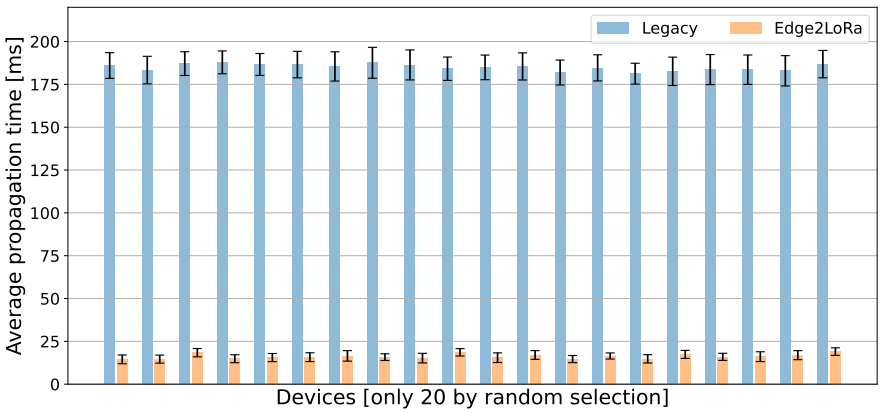
To further showcase the advantages of the Edge2LoRa method, we concentrated on the initial 5-minute interval from the preceding tests and calculated the precise bandwidth consumption in both cases. The scenarios involving the LoRaWAN standard and Edge2LoRa with 90% data compression are represented in Fig. 5.4a, showing the bandwidth use. The graph reveals a marked increase in bandwidth usage in the LoRaWAN setup as the count of operational EDs rises from 1 to 3000. From this data, we infer an average bandwidth saving of 91.60% when the active E2ED tally reaches 1500. This underscores the efficiency of the Edge2LoRa system in considerably lowering network bandwidth needs, especially in large or dense deployments with multiple active EDs, where GWs rely on various access technologies for internet connectivity (like 3G/4G networks with limited bandwidth). This is also relevant in scenarios where multiple GWs share a single backhaul via a centralized NS.

For assessing backward compatibility, we executed a test in a setting with 3000 EDs, split equally between traditional and E2ED models. Fig. 5.4b portrays the frame count received at each GW. The outcomes resemble those of previous experiments concerning the frames received at each GW. However, the frame count

at the AS or sink is intermediate between the figures seen in pure legacy and Edge2LoRa setups. In this test, only half of the EDs are set as E2ED, with the aggregation function applied solely to these. The E2ED produce 500 frames per second, reduced to 50 post-aggregation. The other 500 frames per second, from legacy EDs, are transmitted to the AS after NS processing, noting that some frames are lost due to interference. This evaluation demonstrates the system's capacity to simultaneously manage traditional EDs and E2ED, obviating the need for immediate ED replacement during a transition phase. This feature enhances the system's versatility and practicality for real-world applications. Lastly, the tests verify that Edge2LoRa enables the coexistence of legacy and E2L components, maintaining backward compatibility with the existing LoRaWAN standard.

Fig. 5.4c continues the Edge2LoRa scenario from 5.3, but simulates a failure of E2GW 1 after 15 minutes of operation. The figure highlights the resilience of the proposed method. Specifically, the Edge2LoRa executor module identifies E2GW 1's failure, and reassigns all E2ED processing to E2GW 2. The graph shows an immediate drop to zero in processed frames at E2GW 1 at the 15-minute mark. Nevertheless, the frame count received at the sink experiences minimal impact as all E2ED are now managed by E2GW 2.

The proposed system also significantly affects the latency of data reaching the AS. As depicted in Fig. 5.1, the direct link between the GW layer and the AS circumvents the need for data traversing the NS, thereby reducing overall latency, including the time spent in NS processing.



(a)

Figure 5.5. Average propagation time for 20/3000 random devices.

For our conclusive performance assessment, we selected a subset of 20 devices from the total 3000 and charted the average latency comparison between the traditional and Edge2LoRa setups (with a 1-frame aggregation interval). As illustrated in Fig. 5.5, the mean delay for each device stands at around 180 ms, represented in a bar chart with the standard deviation indicated by error bars. It's noteworthy that these figures encompass the entire transmission route from the GW to the AS, passing through the NS, inclusive of processing duration. In contrast, the same 20 devices, when set up as E2ED, are depicted in Fig. 5.5 showcasing latency measurements in the Edge2LoRa configuration, recording an average of 16ms. A comparison of both scenarios reveals a substantial 76% decrease in latency when data is transmitted directly from the GW to the AS, bypassing the NS.

5.7 Cloud-based Testing and Modularity

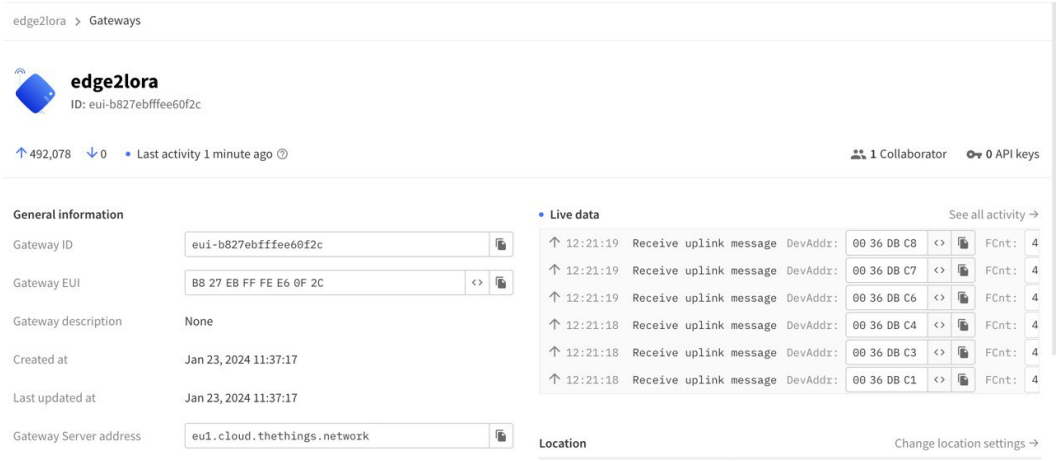
Following comprehensive evaluations within the controlled confines of the University of Palermo's laboratory, it became essential to verify the modularity and adaptability of our device simulator in broader contexts. To this end, we expanded our testing to The Things Network (TTN) [17], a renowned cloud-based IoT platform. This move was instrumental in demonstrating our system's ability to extend beyond laboratory boundaries and operate effectively in a cloud environment.

A unique feature of our device simulator is its ability to emulate not just the device functionalities but also the packet forwarding mechanism typical of LoRaWAN gateways, which traditionally act merely as conduits for data packets without incorporating any processing intelligence. For our cloud-based testing, we adapted the simulator's configuration to interface with TTN. By simply modifying the IP address and port details in the `gateways.json` file to align with TTN's specifications, we facilitated the seamless transmission of packets to the cloud platform.

This adjustment allowed us to replicate the exact test conditions and procedures previously conducted in our laboratory environment, yielding consistent results that reinforced the reliability and robustness of our system. Although the detailed figures from these tests are not reiterated here, they are meticulously documented in our database, readily accessible for review.

The successful deployment and testing on TTN underscore the system's remarkable adaptability and confirm its functionality across diverse settings, be it in standard LoRaWAN or Edge2LoRa configurations. This versatility is crucial, ensuring that our solution is capable of operating efficiently in any scenario, thereby broadening its applicability and appeal in the realm of IoT technologies.

(a)



(b)

Figure 5.6. (a) gateways.json fields can be also used for sending packets to TTN instead of a gateway,(b) is a portion of the first experiment from 5.2 conducted on TTN [17].

Chapter 6

Conclusions

This thesis has presented the development and evaluation of a LoRaWAN Packet Simulator, aimed at enhancing the understanding and optimization of LoRaWAN networks within IoT applications. The primary goal was to create a tool that allows for realistic simulation of network behaviors, addressing the challenges of scalability and performance in large-scale deployments.

The simulator has proven to be a significant asset in the field, offering insights into network dynamics and facilitating the exploration of various deployment strategies. Through comprehensive testing, it has demonstrated its ability to accurately model network conditions and interactions, thereby contributing to more effective network design and management.

Looking forward, there is substantial potential for further enhancing the simulator. Notably, the simulator can be adapted to support downlink communication, in addition to its existing uplink capabilities. This addition would enable a more comprehensive simulation of LoRaWAN network traffic. Additionally, the simulator's capability to work with JSON datasets containing real-world data, already implemented but not covered in this paper, presents a significant opportunity for future research. These features, accessible through the simulator's GitHub[1] repository, offer exciting prospects for more advanced and realistic network simulations. In summary, this work has successfully achieved its objectives, laying a foundation for continued advancements in LoRaWAN technology and its application in the burgeoning field of IoT.

[1] https://github.com/Edge2LoRa/e2l-device-simulation

Bibliography

[1] Chirpstack gateway bridge. https://www.chirpstack.io/gateway-bridge/. Accessed: 2024-03-01.

[2] Lorawan architecture. https://www.thethingsnetwork.org/docs/lorawan/architecture/.

[3] Pbench. https://distributed-system-analysis.github.io/pbench/index.html.

[4] LoRa Alliance. Lorawan l2 1.0.4 specification. https://lora-alliance.org/wp-content/uploads/2020/11/LoRaWAN-1.0.4-Specification-Package_0.zip, 2020.

[5] Domenico Garlisi, Pietro Gallo, Fabrizio Giuliano, and Ilenia Tinnirello. Enabling edge processing on lorawan architecture. *IEEE Access*, 10:58443–58457, 2022.

[6] Domenico Garlisi, Ilenia Tinnirello, Giuseppe Bianchi, and Francesca Cuomo. Capture aware sequential waterfilling for lorawan adaptive data rate. *IEEE Transactions on Wireless Communications*, 20(3), 2021.

[7] HelTecAutomation. Cubecell-arduino. https://github.com/HelTecAutomation/CubeCell-Arduino. Accessed: 2024-01-08.

[8] Jose D. Hernandez-Betancur. Data orchestration using prefect and pyspark. https://medium.com/globalwork-data-driven-world/data-orchestration-using-prefect-and-pyspark-7321559864f7, 2023.

[9] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. Energy aware edge computing: A survey. *Computer Communications*, 151:556–580, February 2020.

[10] Mohammed Jouhari, Nasir Saeed, Mohamed-Slim Alouini, and El Mehdi Amhoud. A Survey on Scalable LoRaWAN for Massive IoT: Recent Advances, Potentials, and Challenges. *IEEE Communications Surveys & Tutorials*, 25(3):1841–1876, 2023. arXiv:2202.11082 [cs, eess].

[11] Anthony Kirby. lora-packet. https://github.com/anthonykirby/lora-packet.

[12] Sachin Kumar, Prayag Tiwari, and Mikhail Zymbler. Internet of Things is a revolutionary approach for future technology enhancement: a review. *Journal of Big Data*, 6(1):111, December 2019.

[13] Lora-net. Packet forwarder. https://github.com/Lora-net/packet_forwarder/tree/master. Accessed: 2024-01-08.

[14] Jaco M. Marais, Reza Malekian, and Adnan M. Abu-Mahfouz. LoRa and LoRaWAN testbeds: A review. In *2017 IEEE AFRICON*, pages 1496–1501, September 2017. ISSN: 2153-0033.

[15] Stefano Milani, Domenico Garlisi, Matteo Di Fraia, Patrizio Pisani, and Ioannis Chatzigiannakis. Enabling Edge processing on LoRaWAN architecture. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–3, Madrid Spain, October 2023. ACM.

[16] The Things Industries. Raspberry pi gateway models. https://www.thethingsindustries.com/docs/gateways/models/raspberry-pi/, 2023.

[17] The Things Network. The things network. https://www.thethingsnetwork.org/, 2024. Accessed: 2024-01-23.

[18] Asif M. Yousuf, Edward M. Rochester, and Majid Ghaderi. A low-cost LoRaWAN testbed for IoT: Implementation and measurements. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 361–366, February 2018.