SAPIENZA
UNIVERSITÀ DI ROMA

Adaptive Resource Management in the Edge-Cloud Continuum for IoT Systems with LoRaWAN Mobility

Facoltà di Ingegneria dell' Informazione, Informatica e Statistica
Engineering in Computer Science

Valerio Francione
ID number 2047712

Advisor
Prof. Ioannis Chatzigiannakis

Co-Advisor
Prof. Domenico Garlisi

Academic Year 2023/2024

Thesis defended on 23 May 2025
in front of a Board of Examiners composed by:

Prof. Alessandro De Luca (chairman)
Prof. Irene Amerini
Prof. Roberto Capobianco
Prof. Ioannis Chatzigiannakis
Prof. Francesco Leotta
Prof. Lorenzo Marconi
Prof. Giuseppe Oriolo

Adaptive Resource Management in the Edge-Cloud Continuum for IoT Systems with LoRaWAN Mobility
Sapienza University of Rome

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

Author's email: francione.2047712@studenti.uniroma1.it

Abstract

In the context of modern communication networks, efficiently managing message traffic across distributed gateways is a critical challenge, particularly in real-world scenarios with dynamic and uneven workloads. This thesis presents the design and implementation of a Reinforcement Learning (RL) algorithm aimed at optimizing message load balancing across multiple gateways. The proposed solution dynamically adapts to changing traffic patterns, learning optimal routing strategies that minimize bottlenecks and ensure more equitable resource utilization. The project involved a thorough analysis of the problem domain, selection and application of relevant technologies, and a comprehensive experimental evaluation. Results demonstrate that the RL-based approach improves workload distribution efficiency compared to traditional static methods, highlighting its potential for deployment in real-world systems. This work contributes to the broader field of intelligent network management by showcasing how RL techniques can be effectively applied to real-time, distributed environments.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) has become a game-changing technology that allows linked gadgets to communicate with the outside world by using sensors to gather data and actuators to carry out operations. Its applications cover a wide range of fields, such as industrial automation, transportation, healthcare, and smart cities. The creation of autonomous networks of devices with the ability to make decisions on their own is the ultimate goal of IoT systems. Developments in artificial intelligence (AI) and machine learning (ML), which enable gadgets to carry out intricate tasks with great accuracy, are progressively enhancing this autonomy. However, because of their limited processing power and storage, end devices are ill-equipped to meet the problems posed by these tasks, which frequently require significant computational resources and access to massive datasets.

Computational workloads can be moved away from end devices to overcome these difficulties. With end devices serving only as data collectors, the Cloud Computing paradigm first provided a solution by allowing the distant execution of complex processing activities [1]. However, because data are transmitted via the Internet, cloud computing has disadvantages such as higher latency and possible security threats. The Edge Computing paradigm has recently surfaced as a viable substitute. By processing data close to the source, edge computing [2] reduces latency, improves security, and guarantees scalability in Internet of Things applications.

Low Power Wide Area Network (LPWAN) technologies like LoRaWAN are essential for enabling IoT networks. Based on the LoRa protocol, LoRaWAN allows for long-distance communication while using very little energy. Although LoRaWAN offers effective gateway-based data forwarding, its centralized architecture poses problems with latency, scalability, and mobility management in dynamic situations. The architecture's primary job is packet forwarding, which restricts its capacity to carry out sophisticated network operations.

1.1 Inspiration and Extent

The `Edge2LoRa` framework was introduced by Milani et al. [14] to overcome the drawbacks of centralized LoRaWAN networks. By incorporating edge computing into LoRaWAN networks, this architecture reduces the dependence on centralized cloud servers and allows a distributed processing paradigm. Although `Edge2LoRa`

enhances processing efficiency and scalability, it does not have the best mechanisms to manage mobility scenarios and dynamic device-to-gateway allocations.

To avoid these restrictions, this thesis presents `Edge4LoRa`, an expanded architecture built upon `Edge2LoRa`. The use of a realistic dataset that mimics real-world mobility patterns and is drawn from the Roman Taxi Dataset which is the foundation upon which the balancing algorithms developed as a result of this work have been tested. The dataset is the foundation for assessing `Edge4LoRa`'s performance, especially in situations that require efficient load balancing and resource allocation.

The article "Resource Allocation Algorithm With Multi-Platform Intelligent Offloading in D2D-Enabled Vehicular Networks" [6] makes a significant contribution by discussing the use of a reinforcement learning algorithm to determine whether to conduct the computations on the cloud, on the gateway (MEC), or on the local device. I used a similar concept in my thesis, however I changed the method that chooses which gateway to utilize for the computations.

1.2 Objectives and Contributions

Through the resolution of significant mobility and scalability issues, this thesis aims to further the integration of edge computing into LoRaWAN networks. The primary goals are:

- Create load balancing algorithms so that there is a standard from which to work.
- Develop a reinforcement learning algorithm that can enhance the effectiveness of the conventional balancing technique.
- Adjusting the RL algorithm to achieve optimal results.

Important contributions consist of the following.

- Constructing a Reinforcement Learning system that could optimally manage an Internet of Things environment's resources.
- Analyzing how AI might enhance a system's overall performance in a practical setting.
- demonstrate the distinctions between conventional balancing systems and how they may be less effective than a contemporary AI system.

1.3 Structure of the Thesis

An introduction to IoT technologies is given at the beginning of the thesis, with particular attention to LoRaWAN, its drawbacks, and how Edge Computing could help with these issues. The `Edge2LoRa` and `Edge4LoRa` frameworks, which form the basis of this study, are then introduced. The development of the dataset is described in depth, as is an assessment of its properties both before and after the reshaping procedure. The visualization tools created to evaluate the efficacy of the suggested balancing algorithms in the `Edge4LoRa` architecture are also described in

the thesis. Following the presentation of experimental data that show the efficiency and scalability of the suggested framework, conclusions and recommendations for additional research are made.

Chapter 2

Background

The `Edge2LoRa` and `Edge4LoRa` frameworks are designed and implemented using a variety of technologies that facilitate effective data processing, network connectivity, and scalability. The main technologies used in this work are summarized in this chapter, along with their functions and contributions to the creation and assessment of the suggested systems.

The technologies used in this study were picked because they are pertinent to the problems with edge computing and Low Power Wide Area Networks (LPWANs). These consist of data processing frameworks, communication protocols, and simulation tools, all of which are essential to address particular facets of the issue area. The study guarantees a solid and trustworthy basis for experimentation and innovation by utilizing well-established instruments and frameworks.

2.1 Low-power Wide-area Network (LPWAN)

Standard networking transmission technologies and protocols are not the best option in the Internet of Things domain. In this situation, new protocols must be created to handle the limitations of IoT devices, such as their limited processing capacity and requirement to use the least amount of energy feasible. However, devices frequently do not need to exchange a lot of bytes during a transmission. A large amount of data that needs to be processed or stored is produced when data from numerous sensors is combined with periodic transmissions from the same device. Minimal power consumption Wide-area networks, or LPWANs, are cutting-edge technologies that meet IoT requirements while also utilizing the low throughput required. LoRaWAN, a network protocol that is popular in the context of the Internet of Things, is a participant in LPWANs.

2.2 LoRa

One of the most crucial network protocols in the LPWAN space is LoRaWAN. LoRa (**Lo**ng **Ra**nge) is a radio signal modulation technology that may be used to create mesh networks or in conjunction with the network protocol LoRaWAN. Since the Edge2Lora infrastructure is based on LoRaWAN, the mesh method is not included in this study. [10]

2.2.1 LoRa Description

As previously stated, LoRa is a unique physical layer protocol that was first patented by Cycleo in 2014 before being purchased by Semtech. Low power consumption and long-range communication are two of LoRa's primary features. The sub-gigahertz spectrum contains all of the frequency bands that are used, with the US and the European Community using 868 MHz and the US using 915 MHz, respectively. Under ideal circumstances (line of sight and free from radio frequency interference), the low-frequency band enables LoRa to transmit and receive signals at remarkable distances. In "Long-range communications in urban and rural environment" [3], an assessment of transmission capabilities has been conducted, taking into account three distinct scenarios: urban, suburban, and rural. This demonstrates that the transmission range in an urban setting decreases by about 2 km, whereas the LoRa modulation approach may reach rural areas over 5 km with still high connection quality. Additionally, this work demonstrates how items in the center and transceiver altitudes impact transmission.

2.2.2 Lora Modulation Technique

The frequency spectrum employed determines how far LoRa can communicate; it is well known that lower frequencies may travel farther due to their higher permeability of space, but the bit rate provided is poor. Another important element in the performances provided by LoRa is the Spreading Factor (SF), which is not solely dependent on the spectrum employed. LoRa produces a signal known as the Chirp Signal, which is characterized by a frequency modulation that begins at F_{min} and increases linearly to F_{max} (Up Chirp if goes in the other way is Down Chirp). LoRa disperses the energy over a greater frequency range to chirp the signal over a wider bandwidth. This improves the gateway's ability to receive signals while strengthening resilience to noise and interference.

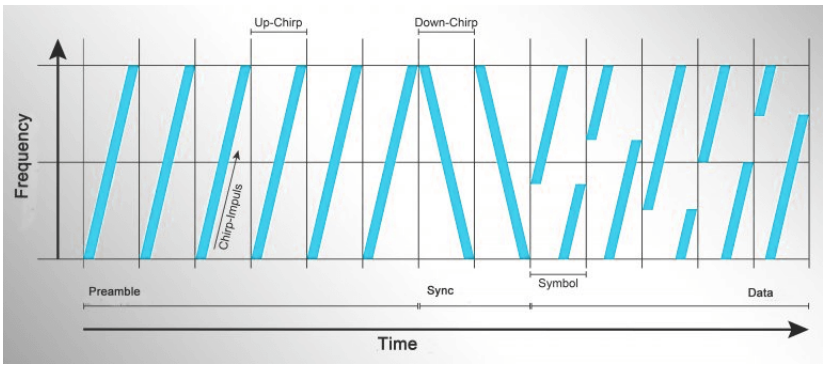


Figure 2.1. Lora Chirping and spreading Factor

LoRa's spreading factor values range from 7 to 12, and as it rises, the time slot for each tone grows and the resulting chirp signals cover a wider frequency band.

Preamble	Physical Header	Physical Header CRC	Payload	Payload CRC
6 - 65535 bits	1 Byte	1 - 2 Bytes	1 - 255 Bytes	4 Bytes

Table 2.1. Lora Physical Uplink Frame

2.2.3 Lora Frame Structure

There are two categories of Lora frames: uplink and downlink. To ensure more dependable and secure connection, devices send packets to the gateway using uplink frames. Instead, a packet transferred from the gateway to the device is known as downlink. An illustration of the uplink frame can be found in the table that follows.

- **Preamble** is used to synchronize communication between the two devices. It consists of a series of Up-Chirps signals followed by two Down-Chirps.
- **Physical Header** includes the fundamental data required for communication, such as the bandwidth, coding rate, spreading factor, and presence of CRC, which are not supplied in the case of a downlink frame.
- **Physical Header CRC** is the header's integrity check.
- **Payload** The physical payload is separated into various fields.
- **Payload CRC** is the payload's integrity check (only in Uplink frames)

The packet's explicit format is the one shown above; the implicit form eliminates the header and is appropriate for situations where the communication parameters are predetermined or known. Beacons transmit time-synchronizing data from gateways to end devices via LoRa radio packet implicit mode.

2.2.4 Physical Payload

The LoRaWAN packet is housed in the physical payload. The physical layout of LoRaWAN is as follows:

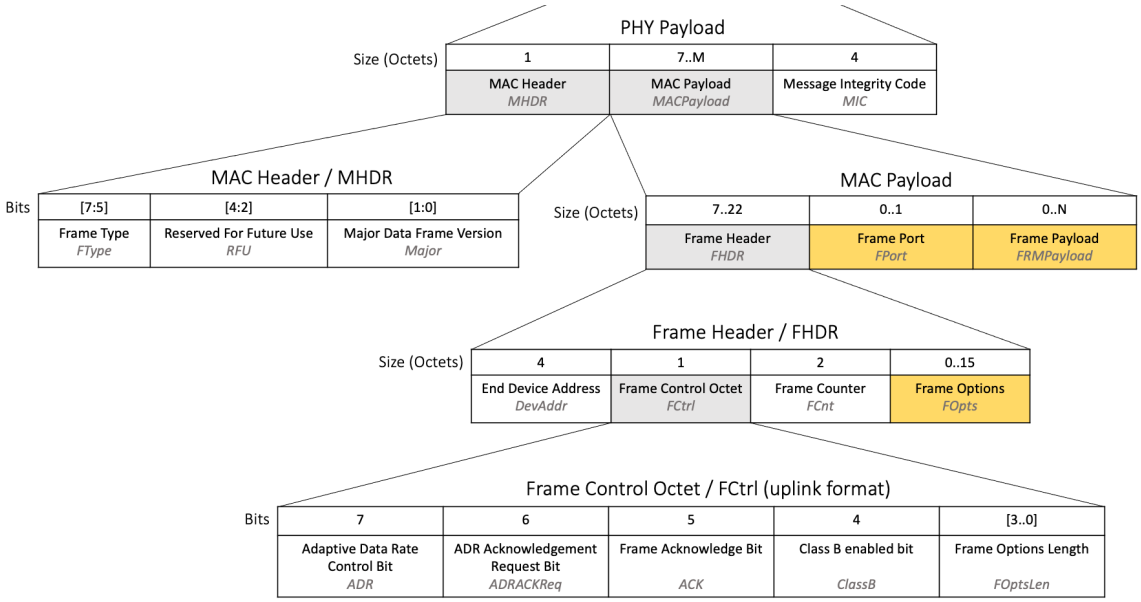


Figure 2.2. Physical LoraWan Payload

All of the control information is contained in the second block of the frame header, which is structured as follows:

- **Mac Header** is one byte long and includes the fields listed below:
  - **Frame Type** including a 3-bit code that indicates whether confirmed or unconfirmed uplink or downlink data is present, or that indicates the message nature, such as JOIN REQUEST/ACCEPT.
  - **Reserved for future use (RFU)** Three-bit fields are not used.
  - **Major Version** The primary version of the protocol is specified by two bits.
- **Mac Payload** includes all of the information related to LoRaWAN Transmission and Control.
  - **Frame Header**
    - **End Device Address** is made up of the 32-bit End Device address, which identifies the device in the current network but is not a unique address for it. This field is connected to the network composition of LoRaWAN. The network address is represented by seven bits of the DevAddr, while the device uses the remaining twenty-five bits. The device is given the address during a process known as activation, which will be examined in more detail later in this background section.

    - **Frame Control Octet**Each control option is represented by a sequence of bits, such as the Frame Acknowledgment bit, ADR Acknowledgment Request bit, Adaptive Data Rate (ADR) Control bit, Class B enabled bit, and Frame Options Length, which take up the remaining three bits.
    - **Frame Counter**
    - **Frame Options**
  - **Frame Port** (optional) determines the port of the application.
  - **Frame Payload** It has a variable length and includes the message data.

2.3 LoRaWAN

A popular network protocol in LPWANs, it is made up of more than just specifications for packet format and transmission parameters; it can be viewed as an entire network architecture made up of many actors and devices.

2.3.1 LoRaWAN architecture

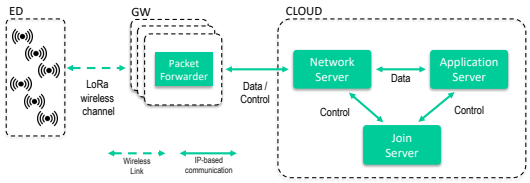


Figure 2.3. LoRaWAN Network Architecture

The definition of the infrastructure of a LoRaWAN network is shown in Figure 2.3. Using a wireless link with LoRa, end devices connect to gateways (also known as concentrators). The messages gathered by the gateways can then be transmitted to the cloud via an IP-based communication link (either Ethernet, WiFi, or a mobile cellular network connection), where the second LoRaWan Network component is set up with an Application Server, a Join Server, and a Network Server.

- **LoRaWAN End Devices (ED)** Usually small devices that run on batteries and have sensors. The gadget has sensors built inside it that are used to periodically gather environmental data. Since the data cannot be kept locally, the device broadcasts the data to every gateway it can reach using a LoRa transceiver.
- **LoRaWAN Gateways (GW)** Gateways constantly listen to the LoRa carrier in order to get ED messages. They have at least two interfaces to serve as a bridge between the Internet and the LoRaWAN network: a LoRa transceiver and an IP-based interface that links the gateway to the Internet via Ethernet, WiFi, or a 4G/5G connection. Here, energy conservation is not necessary because, in certain cases, gateways are linked to the grid so they can gather electricity.

- **LoRaWAN Network Server (NS)** is the organization in charge of overseeing network control data, such as the data rate and spreading factor. Unlike an Application Server or Join Server, the network server is not a physical part. Each of the three software components has a distinct function. The network server is the one who verifies that a message has not been duplicated in the AS because LoRaWAN communication is broadcast and it is common for numerous gateways to receive the same packet.
- **LoRaWAN Join Server (JS)**. To connect to LoRaWAN, devices must successfully complete an Over The Air Activation (OTAA) Phase. A session key is generated during activation to enable private and secure communication between the device and the Application Server.
- **LoRaWAN Application Server (AS)**. is the software component that has access to the messages' payload; all other components can only access control information, and the entire communication is encrypted.

2.3.2 LoRaWAN End Devices

The behavior and supporting functionalities of end devices are used to categorize them into three types.

- **Class A**: Every device can perform the functions covered by this class. For example, bi-directional communication is supported by scheduling two brief downlink windows following each uplink connection, which activates the transceiver to receive a packet from a gateway. This is the mode of operation that uses the least amount of power for an end device.
- **Class B**: In addition to Class A's random receiving slots, such devices provide additional receiving slots by adding scheduled receiving slots.
- **Class C**: Only when a transmission is required do devices that are constantly listening to the carrier stop receiving. This behavior makes it the class that uses the most energy.

2.3.3 End Device Activation

An outline of the LoRaWAN 1.1 activation process is provided in the section following. There are two techniques for activating devices: hard-coding the device's keys or using the ABP activation mechanism. When a device joins the LoRaWAN network, it must first be setup and enabled via an Over-The-Air activation phase. The following codes and identifiers are used in order to better comprehend the activation process, which makes use of multiple parameters and identifiers:

- **JoinEUI** is a 64-bit identification that is specific to the network and needs to be loaded into the device before it can be activated.
- **DevEUI** is the device's unique ID; it has the same format as JoinEUI and has to be inside the device during the activation phase.

- **NwkKey** and **AppKey** are AES-128 root keys that were assigned during fabrication and are unique to the finished device.

The following session keys are derived from the NwKey during the OTAA activation of an end device: NwkSIntKey, SNwkSIntKey, and NwkSEncKey. The AppSKey session key is derived using AppKey.

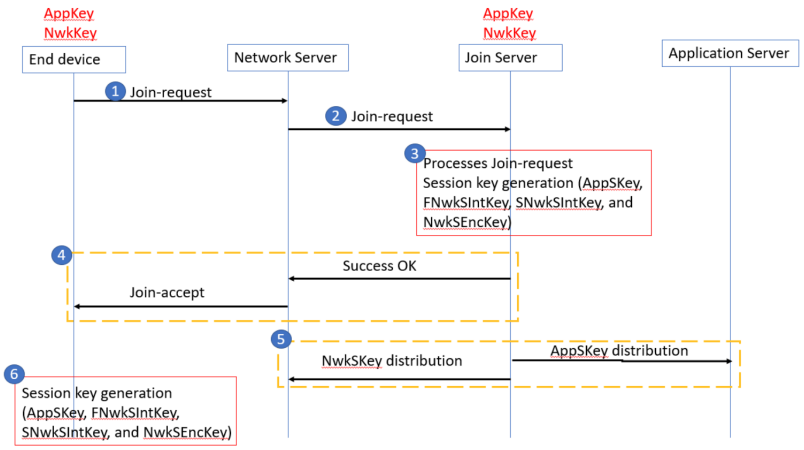


Figure 2.4. LoRaWAN 1.1 Activation procedure

The communication schema between the end device, network server, join server, and application server is shown in the above diagram. The following steps make up the activation process:

1. The message is always transmitted to the Network Server, and the End Device must first share with the Join Server via a Join Request using the JoinEUI.

2. The relevant Join Server receives the Join-request message from the Network Server. The components of the Join Request are JoinEUI, DevEUI, and a nonce.

3. After processing the Join Request, the Join Server creates all of the necessary session keys.

4. The following fields are included in the Join Accept message that is created by the network server: Join Nonce, which is used by the end device to produce all session keys; NetID, which is the network's identifier; Dev Address, which identifies the device in the network; and additional communication settings.

5. The three network session keys (FNwkSIntKey, SNwkSIntKey, and NwkSEncKey) are sent to the Network Server by the Join Server, whereas the AppSKey is sent to the Application server.

6. The Join-accept message is decrypted by the end-device using the AES encryption procedure. The final device generates session keys using AppKey, NwkKey, and JoinNonce.

Following completion of those procedures, the device is turned on and stores the following data: AppSKey, FNwkSIntKey, SNwkSIntKey, DevAddr, and NwkSEncKey.

Every other device in the network is identified by its DevAddr; the others are session keys. Every Session Key has a distinct function:

- **FNwkSIntKey** is employed to determine the MIC (partially) of each uplink data message by the end device in order to guarantee message integrity.
- **SNwkSIntKey** is employed by the end device to determine the message integrity metric (MIC) of all downlink data messages and the MIC (partially) of all uplink data messages.
- **NwkSEncKey** is employed to ensure message secrecy by encrypting and decrypting the contents of the uplink and downlink data messages using MAC instructions.
- **AppSKey** is employed to ensure message confidentiality by encrypting and decrypting the application data in the data messages by the Application Server and the end device.

2.4 Edge2Lora

This study starts with Edge2Lora, an architecture based on LoRaWAN that addresses some of the inefficiencies of LoRaWAN and allows edge processing. By improving scalability and decreasing processing latency, Edge2Lora expands the potential of LoRaWAN. Backward compatibility with legacy devices that do not support Edge2Lora is ensured by overlaying the Edge2Lora design on top of the current LoRaWAN architecture.

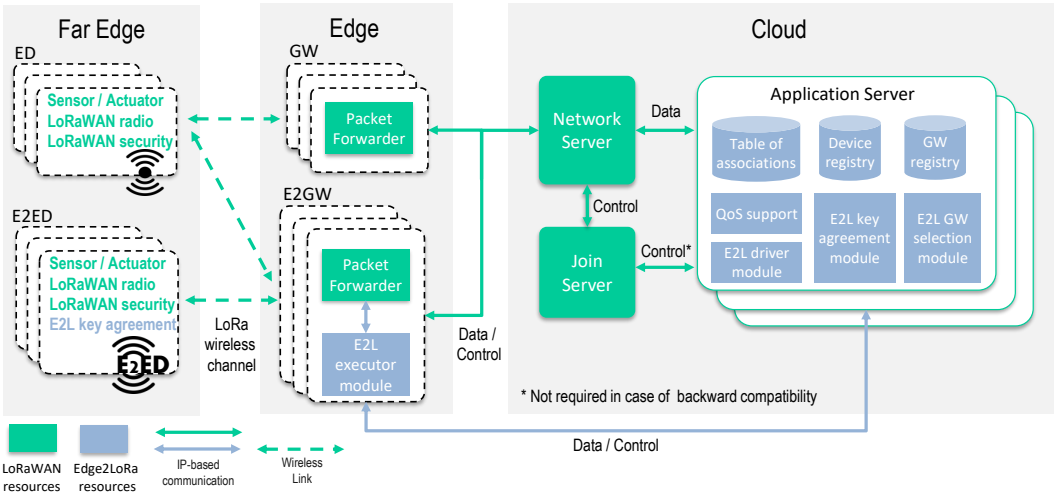


Figure 2.5. Edge2Lora architecture schema including LoRaWAN Legacy devices

According to the description of Edge2Lora given by Milani et al. [14], Figure 2.5 demonstrates that Legacy devices and gateways can function together in the same network with `Edge2Lora` devices and gateways. The gateways in `Edge2Lora`

are known as E2GWs, and they can also function similarly to a typical LoRaWAN gateway by forwarding packets to the application server.

2.4.1 Edge2Lora Architecture

The `Edge2LoRa` framework's performance and usefulness are made possible by a number of essential parts. These elements consist of the sensor data stream processing system, the group key establishment mechanism, and the device registry. These components work together to guarantee effective and safe network connection.

To indicate compatibility, the framework includes distinct device classes. The Gateways (GWs) intended to function inside the `Edge2Lora` framework are called **E2GW**, and the *End Devices* (EDs) compatible with the `Edge2Lora` framework are called **E2ED**.

An allocated **E2GW** receives and processes the sensor data streams that are transmitted by each **E2ED**. In order to ensure that the transmitted data streams are handled effectively and securely, the **E2GW** serves as the intermediate in charge of carrying out data processing activities. By distributing the computing load, this architecture improves the system's scalability and responsiveness.

The `Edge2Lora` framework's primary function is to control the existence of duplicate packets. This already occurs in LoRaWAN when a packet is sent to many gateways in duplicate, and the best one (based on signal quality) is selected. Avoiding redundant processing or storing of data from various gateways is crucial in `Edge2Lora`. The gateway no longer merely serves as a packet forwarder; it now actively participates in computation and is useless for repeatedly processing data across several gates.

Figure 2.5 shows the `Edge2Lora` framework's architecture. Based on their computational capability and location within the network, its components are divided into three types.

2.4.2 Far Edge Devices

Far Edge Devices are made to function with very little processing power. Small programs can operate on these devices without consuming a lot of processing power. *Heltec Cubecell* modules and *Arduino* boards are examples of appropriate hardware for far-edge devices. The main function of far-edge devices is to use their sensors to gather data and send it over LoRa. To enable deployment in remote or resource-constrained situations, these devices are usually battery-operated.

2.4.3 Edge Devices

Edge Devices are typically linked to a dependable energy source, like the electrical grid, and are more potent than far edge devices. Edge devices are capable of handling more computationally demanding tasks than far edge devices, which are frequently battery-operated. A *Raspberry Pi* with a LoRa antenna is a good example of hardware for edge devices since it can receive LoRaWAN packets from distant edge devices. By connecting the distant edge devices to higher-level processing systems, edge devices serve as gateways.

2.4.4 Cloud

The *Cloud* is the last class of devices in the `Edge2LoRa` framework. This layer is in charge of overseeing centralized duties for the network as a whole and running computationally demanding software components. The device-gateway association module and the key management system are two important functions carried out at this level. The framework's backbone, the cloud, makes it possible for the network's devices to be operated and managed effectively.

2.5 Edge4Lora

`Edge4LoRa` is a derived architecture that expands on the initial implementation, starting with `Edge2LoRa`. To improve gateway-level data processing capabilities, it integrates a number of other modules. While adding a `Map/Reduce Engine` for data processing and a new interface to control packet redirection and facilitate load balancing between gateways, the `Edge4LoRa` framework retains all the elements of `Edge2LoRa`.

2.5.1 Components in Edge4LoRa

`Edge4LoRa` adds three new elements to the gateway layer:

- **Message Broker:** ensures reliable message delivery by facilitating the exchange of frames utilizing *MQTT* as the communication protocol.
- **Apache Spark Engine:** allows sophisticated calculations to be performed right at the gateway by offering a reliable environment to process large amounts of data.
- **Edge4LoRa Parser:** a specialized message parser made to effectively manage incoming data streams, guaranteeing smooth sensor data integration and preparation.

By lowering reliance on cloud resources and improving overall system performance, these changes greatly increase the framework's capacity to handle, analyze, and manage data at the edge.

The entire `Edge4Lora` architecture is depicted in Figure 2.6. It is arranged in three layers, similar to `Edge2Lora`: Cloud Layer, Edge Layer and Far Edge Layer. The arrangement and positioning of the previously described elements within the overall architecture are depicted in the picture.

Gateways now come with a new set of parts:

- The **Message Parser** is in charge of extracting the information from the packet by opening its contents. Gateway additionally comprises:
  - A **SPARK Module**, able to process the data using a series of map-reduce procedures.
  - A **Local Broker**, this is a MQTT server that makes it easier for gateways to share data and control information.

– A **Edge4Lora Parser**, It is responsible for unpacking the data that is contained in the LoRaWAN frame.

Three potential outcomes of packet delivery via LoRaWAN are also depicted in the figure:

1. **Processing on Receiver:** In this instance, the gateway in charge of processing the data in the transmitted message receives the packet. The `Packet Forwarder` receives the packet first, followed by the `Edge4Lora Parser` unpacking it and publishing it on the assigned MQTT topic. The data is processed directly if the subject matches the receiving Gateway, and the outcome is published on a different topic that the `Edge4Lora Sink` can access.

2. **Processing on Another Gateway:** The Gateway that receives a packet acts similarly to the preceding scenario if it is not in charge of processing the data included in that frame. In this case, however, the message's MQTT topic is linked to another network gateway rather than the receiving gateway.

3. **Legacy Device:** The data from the device will not be published on the processing topic if a delivered message comes from a legacy device that does not support `Edge4Lora` or `Edge2Lora`. It will only be sent straight to the application server instead.

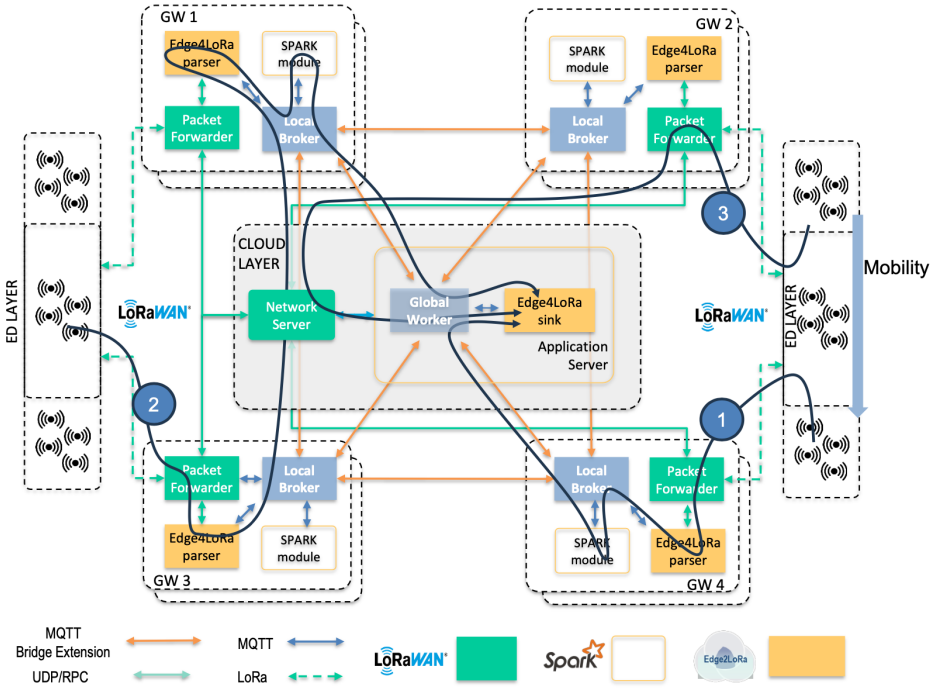


Figure 2.6. Edge4Lora architecture

2.6 The Packet Forwarder: A cornerstone in LoRaWAN Gateway Operation

Every LoRaWAN gateway's packet forwarder, a core software application running on the gateway's host system, usually an embedded computer, is what makes it work. For controlling the two-way flow of radio frequency (RF) packets, the packet forwarder is the main engine. Its primary functions are to process radio frequency packets (RF) that end devices send to the gateway's radio concentrator and forward them to a network server, typically through an IP and UDP link; and, on the other hand, to receive commands and data packets from the network server and plan their return to the end devices as RF packets.

In both downlink (server-to-device) and uplink (device-to-server) communication pathways, the packet forwarder plays a crucial role. When an end device transmits data for uplink traffic, the gateway radio concentrator records the RF packet. Before encapsulating these data, usually in a JSON format, for transmission to the Network Server, the packet forwarder retrieves this raw packet and adds necessary metadata, such as the Received Signal Strength Indicator (RSSI), Signal-to-Noise Ratio (SNR), exact timestamp of arrival, and the particular radio channel used. The Network Server depends on this metadata to carry out essential tasks like localizing devices, evaluating link quality for Adaptive Data Rate (ADR) modifications, and de-duplicating packets received by multiple gateways.

The Network Server sends data payloads and instructions to the packet forwarder for downlink communication. These payloads may consist of network administration commands, such as possible gateway configuration updates, or application-specific data headed to an end device. The packet forwarder then schedules these packets to be transmitted over the air through the radio concentrator. The emission of network-wide beacon signals that are synced with GPS is also supported by certain packet forwarder implementations. Class B LoRaWAN operations, which allow scheduled downlink communication slots and hence enable low-latency commands to end devices, are made possible by these beacons.

In terms of functionality, the packet forwarder serves as an abstraction layer, separating the IP-based Network Server from the complexities of the actual radio hardware, or concentrator. While the Network Server manages higher-level LoRaWAN protocol logic, security, and application data routing over conventional IP networks, the concentrator is focused on RF signal processing and LoRa modulation/demodulation. By translating between these domains, the packet forwarder fills this gap. As long as a common packet forwarder protocol is followed, this abstraction enables the Network Server to communicate with a variety of gateway hardware without having to comprehend the subtle differences between each radio chipset. But because of its crucial function, the packet forwarder software itself also becomes a crucial link in the data communication chain, directly affecting its stability, performance, and proper implementation.

The packet forwarder is more than just a passive conduit, even while the Network Server, which controls device authentication, message outing, ADR techniques, and security key distribution, contains a large portion of the intelligence of the LoRaWAN network. It carries out vital local tasks that require a certain level of

"intelligence" and exact time. The "Just-in-Time" (JiT) scheduling technique used by several packet forwarders for downlink messages is a prominent example. The packet forwarder keeps a local queue of pending downlink packets as part of JiT scheduling. Based on the packet's timestamp, it then programs a packet into the concentrator's transmission buffer just prior to its allotted airtime. This maximizes downlink capacity and makes the best use of the concentrator's constrained transmit buffer. Accurate time synchronization between the gateway's host system, the radio concentrator, and the Network Server is necessary for effective JiT scheduling; this frequently calls for a special timer synchronization thread inside the packet forwarder. Network performance can be severely harmed by errors or inefficiencies in these local processing operations, especially for applications that depend on timely downlink communication or on Class B devices operating as intended.

2.6.1 Operational Dynamics and Protocols of the Packet Forwarder

The host system of the gateway, which is usually an embedded computing platform (such as one running Linux), is where the packet forwarder runs. It connects directly, typically through a Serial Peripheral Interface (SPI) link, to the LoRa radio concentrator module or modules inside the gateway, such as the Semtech SX1301 or the more recent SX1302/SX1303 chipsets. Through this interface, the packet forwarder can enqueue packets for transmission, retrieve received RF packets, and configure the radio hardware. A backhaul network connection, which can be wired Ethernet, Wi-Fi, or a cellular (e.g., 3G/4G/5G) link, is used to connect the gateway and, consequently, the packet forwarder to the Network Server. The standard transport layer protocol, UDP, is used for most IP-based communication over this backhaul.

The Semtech UDP Gateway Messaging Protocol (GWMP) has long been the de facto standard for communication between a large number of packet forwarders and network servers. This protocol, which is described in a document commonly called PROTOCOL.TXT, specifies a number of message formats that make it easier to communicate gateway status information and LoRaWAN data. Important GWMP message types consist of:

- **PUSH_DATA (Gateway to Server):** used to transmit the uplink radio frequency packets that the gateway has received, together with the metadata that goes with them (e.g., time, tmst, freq, modu, datr, codr, rssi, lsnr, size, data). Within a "stat" object, this message can also include gateway status information. Usually, the payload with this data is organized in JSON format.
- **PUSH_ACK (Server to Gateway):** In response to a PUSH_DATA message, the server sends an acknowledgment.
- **PULL_DATA (Gateway to Server):** a message sent to the server on a regular basis by the gateway. Its major objectives are to keep the session active in order to maintain the UDP communication path past firewalls or Network Address Translators (NATs) and to poll the server for any pending downlink messages.

- **PULL_ACK (Server to Gateway):** An acknowledgment from the server for a PULL_DATA message.
- **PULL_RESP (Server to Gateway):** Utilized by the server to transmit a downlink radio frequency packet to the gateway, which usually consists of a JSON object called "txpk" and contains the payload and transmission parameters (such as imme, tmst, freq, powe, modu, datr, codr, size, and data).
- **TX_ACK (Gateway to Server):** gives the server information about the status of a PULL_RESP order from the gateway, such as whether the downlink packet was successfully scheduled for transmission or if an error occurred (TOO_LATE or COLLISION_PACKET, for example).

According to the GWMP's specification, its design indicates that it was first intended to function on "reliable networks." Its acknowledgment messages (PUSH_ACK, PULL_ACK) are designed more for network quality evaluation than for implementing dependable delivery through retransmissions, which is a feature of UDP's connectionless nature. It also noticeably lacks built-in authentication mechanisms for the gateway or server. This implies that during the protocol's initial development, ease of use and quick deployment were given precedence above the resilience needed for large-scale, production-grade networks. These intrinsic constraints, especially the unreliability of UDP and the lack of robust security measures, became major operational pain problems as LoRaWAN deployments grew and spread into more important application domains.

Alternative and improved packet forwarder implementations were directly influenced by this evolutionary pressure. Many commercial gateways still employ the original Semtech UDP Packet Forwarder, which is frequently pre-compiled. Later on, though, Semtech unveiled LoRa Basics Station, a next-generation packet forwarder that was created to fix a number of the issues with its predecessor. LoRa Basics Station has a number of important benefits, including the ability to communicate with the Network Server via secure WebSockets (which are based on TCP and can be secured with TLS) rather than raw UDP, the ability to use the Configuration and Update Server (CUPS) protocol for centralized updates and configuration management, the ability to manage channel plans centrally, and the removal of reliance on exact local timekeeping at the gateway. Other noteworthy implementations include the ChirpStack Gateway Bridge, which can abstract the underlying packet forwarder protocol (such as Semtech UDP) and translate it to Protobuf or JSON messages over MQTT, frequently leveraging TCP for more dependable transport, and the Things Network (TTN) Packet Forwarder, which is written in Go and uses the Gateway Connector protocol for authenticated, dependable, and encrypted communication.

An additional operational difficulty is the dependence of conventional UDP-based packet forwarders on local configuration files (such as global_conf.json and local_conf.json). Every gateway needs to have its parameters, including the Gateway EUI, Network Server address, frequency plan information, and communication ports, precisely set up. If not done carefully, managing these configurations across a potentially sizable fleet of gateways might result in errors and a major operational overhead. By centralizing and automating configuration delivery, CUPS with LoRa Basics Station seeks to lessen this load while subtly drawing attention to the issues

that already existed. Misconfigurations can affect network stability and the general environment in which a load balancing system would operate by causing connectivity problems or incorrect gateway functionality.

2.6.2 Packet Forwarder Challenges: Impact on Network Performance and Reliability

Although LoRaWAN technology has advanced, packet forwarders still suffer a number of difficulties that can have a big impact on the overall performance and dependability of the network, especially those that use the outdated Semtech UDP GWMP. These difficulties are caused by scaling problems, hardware restrictions, operational complexity, and protocol limitations.

One major cause for concern is the intrinsic properties of UDP, the transport protocol that forms the basis of the conventional GWMP. Because UDP is connectionless, it cannot guarantee packet delivery, order, or avoidance of duplication. This instability can directly impact data integrity and the responsiveness of IoT applications in the context of LoRaWAN by taking the form of missed downlink commands or lost uplink sensor data. The unreliability of the gateway-to-server link adds another possible point of failure, even though LoRaWAN itself contains MAC layer methods for confirmed messages. Furthermore, the original Semtech UDP GWMP lacks solid, built-in security procedures like encryption for the data in transit or strong authentication of gateways. The entire network segment that the gateway serves may be compromised by this flaw, which leaves the communication channel between the gateway and the Network Server vulnerable to security threats such metadata eavesdropping, message manipulation, and even gateway impersonation.

Scalability is still another major obstacle. The sheer volume of uplink traffic can overwhelm individual packet forwarders' backhaul connections and strain their processing capabilities as LoRaWAN networks expand to accommodate an ever-increasing density of end devices. To make matters worse, LoRaWAN uses unlicensed ISM bands, which are frequently congested and vulnerable to interference from other wireless technologies and even other LoRaWAN installations. If packet forwarders try to filter or report these distorted messages, the processing load on them increases due to the corrupted packets caused by such interference, which ultimately lowers the effective throughput. Despite being straightforward, LoRaWAN's Aloha-based medium access technique may increase the likelihood of packet collisions in crowded areas, requiring end devices to retransmit, further taxing the gateways and the shared radio spectrum.

Resource limitations are frequently imposed by the physical hardware of LoRaWAN gateways, which house packet forwarders. These embedded systems usually contain persistent storage, RAM, and a CPU with limited computing capacity. These limited resources can be quickly depleted by heavy traffic loads, the computational demands of tasks like JiT scheduling for downlinks, sophisticated filtering rules, or the presence of other software services operating on the gateway (like remote management agents, MQTT bridges, or local data processing). Network dependability and data delivery are seriously jeopardized by resource fatigue, which can show itself in a number of ways that are harmful to network operation, such as missed packets, increased packet processing latency, and unstable or crashed packet

forwarder software.

These difficulties may have a domino effect that makes network problems worse. For example, packet loss between the gateway and the network server could result from UDP's instability. Devices may retransmit their data if end devices do not receive acknowledgments for confirmed uplinks (due to backhaul loss or other problems). In already crowded networks, these retransmissions enhance the likelihood of collisions by increasing channel use. The packet forwarder and the gateway's host system are subjected to a higher processing load when the total number of original packets, retransmissions, and collision-induced retransmissions arrive at the gateway. Further packet drops at the gateway itself, forwarder instability, or higher latency could result from the gateway's inability to manage this additional load. This would create a negative feedback loop that would impair network performance as a whole.

The intricacy is further increased by management and operational difficulties. It can be difficult and prone to mistakes to configure and manage a large fleet of gateways, particularly those that use UDP-based forwarders that require separate file-based setups. The distributed nature of IoT deployments and the numerous interacting components, ranging from end devices to application servers, can make troubleshooting intermittent errors, performance bottlenecks, and connectivity issues more challenging. Another crucial operational issue that affects network security and stability is making sure that many, frequently geographically scattered gateways receive timely firmware updates and security patches.

Some of these UDP-related dependability and security difficulties are directly addressed by the creation of packet forwarder protocols, such as LoRa Basics Station, which uses secure WebSockets and centralized administration capabilities. However, the issues of limited gateway hardware resources and the best way to distribute traffic in a multi-gateway deployment are not automatically resolved by an enhanced and dependable communication link between the gateway and the Network Server. Even with a strong connection to the Network Server, a gateway may still be overloaded with RF traffic if it serves an excessively high number of devices or if it is the only active gateway in a crowded region. Network stability can also be jeopardized by security flaws that continue to exist in packet forwarders or the underlying gateway systems; a compromised gateway could interfere with traffic flow or provide a network management system with erroneous status information, undermining attempts to optimize the network.

Together, these complex issues—which include protocol restrictions, scalability difficulties, gateway resource limitations, security flaws, and administrative overhead—highlight the urgent need for more sophisticated network management techniques in LoRaWAN settings. These tactics are crucial for both addressing current issues and guaranteeing that the network can scalably and dependably accommodate the increasing needs of various IoT applications.

Chapter 3

Load Balancing algorithms

This chapter is devoted to a thorough investigation and methodical classification of the various load balancing methods that have been created and implemented over my career. Giving a basic grasp of the concepts, workings, advantages, and disadvantages of different algorithmic approaches is the aim.

3.1 Simple algorithm

The 'Static Gateway Assignment' algorithm, which is the straightforward algorithm that was assessed in this study, follows a simplistic architecture. Its primary method is the fixed one-time assignment of a particular gateway to every end-user device on the network. Regardless of future modifications to network traffic patterns or gateway load levels, this initial mapping is unchangeable for the duration of the system's operation. Neither load-aware modifications nor dynamic reassignment are supported.

The inherent drawbacks of this static method were made abundantly clear throughout the simulation phase. Its primary flaw was found to be the inability to reallocate devices from a gateway that was overloaded to one that had capacity available. Simulations showed that a gateway would unavoidably and quickly approach resource saturation once it was assigned a set of active devices, even under conditions of modestly fluctuating load. The algorithm's practical utility would be severely limited, and further development of this particular, unmodified approach would be unproductive for realistic deployment scenarios. This saturation, which would appear as exhausted processing capabilities, memory, or bandwidth, would occur in a relatively short timeframe.

Therefore, a methodological compromise was established to allow additional investigation of this algorithm, particularly for comparison as a baseline or to explore other behavioral characteristics in an unrestricted setting. The crucial restrictions of finite gateway resources and the consequent saturation effects were purposefully left out of the model for the simulation runs that were specifically related to this 'Static Gateway Assignment' algorithm. This idealization was required to examine its basic assignment behavior in isolation and to avoid premature simulation collapse owing to resource exhaustion, even if it is not representative of real-world operating restrictions.

3.2 Random algorithm

Within a dynamic system, one of the simplest methods for controlling device-gateway connections is the 'Random Gateway Reassignment' algorithm. Its operational simplicity and dependence on stochastic processes as opposed to deterministic or load-aware logic are its distinguishing features. Specifically, in our simulations, the algorithm is activated when a device experiences a 'movement' event, which is a phrase that encompasses any change in the device's position or status that is considered substantial enough to warrant a possible gateway re-evaluation.

When such a movement event occurs, the algorithm carries out a simple random selection procedure. It makes a probabilistic decision about whether the device should continue to connect to the gateway it is currently linked with or start a handover to a new one. A new gateway is then chosen at random from a candidate pool if the random selection supports a change. Other gateways that have been determined to be "in proximity" to the device's new context or location make up this pool. On the other hand, the device remains served by its previously allocated gateway if the initial random decision requires no change or if the random selection procedure for a new gateway actually re-selects the current one.

3.3 Greedy algorithm

When pairing an end-device with a network gateway, the 'Greedy Multi-Criteria Gateway Selection' algorithm—later known as the Greedy Algorithm—works on the basis of making an instantaneous, locally optimal decision. Usually, this technique is used during recurring system re-evaluation stages or following a device movement event. Its primary purpose is to carefully evaluate and rank accessible gateways according to a hierarchical set of standards to determine which is most suitable at that particular moment. When the Greedy Algorithm is activated, it first creates an exhaustive list of all discoverable or pertinent network gateways that the end device can access. After that, this list goes through an intricate, multi-level sorting procedure. 'Proximity' to the end-device is the main sorting key, used to reduce latency and guarantee effective communication channels. Topologically or geographically closer gateways are given preference when ranking them. If several gateways show similar closeness, 'connection quality,' an additional ranking criterion, is used. Lastly, a tertiary sort based on "available resources" is carried out for gateways that remain tied after taking proximity and connection quality into account. This entails determining or querying the gateways' current spare capacity, including CPU load, and favoring less crowded gateways that can provide better performance under demand. The selection stage of the procedure is carried out after this hierarchically sorted list has been created. It identifies the gateway positioned at the top of this ranked list—the one deemed optimal according to the prioritized criteria. Nonetheless, an important and unique rule is used: the algorithm specifically does not include the gateway that the end-device is already assigned to. As a result, the first gateway that is not the current gateway in the sorted list is chosen. In order to minimize latency and guarantee that request processing for end devices takes place as close to them as possible, the sorting algorithm prioritizes

proximity. Instead of letting the device stick with its current gateway, even if it's still highly ranked but no longer the best option, the exclusion of the currently assigned gateway has a specific function: it forces a switch to a new optimal gateway, if one exists, facilitating load distribution or adaptation to changes.

3.4 Reinforcement Learning algorithm

One of the 21st century's most revolutionary technologies, artificial intelligence (AI) is changing industries, pushing the limits of automation, and changing how humans interact, live, and work. AI is becoming more and more ingrained in our daily lives, from recommendation engines and virtual assistants to self-driving cars and clever medical diagnostics. It is an effective tool for resolving challenging, real-world issues because of its ability to learn from data, adjust to new knowledge, and make judgments with little assistance from humans.

Increased processing power, the availability of vast amounts of data, and developments in machine learning algorithms have all contributed to the notable acceleration of artificial intelligence in recent years. These advancements have created new opportunities in a variety of industries, including as communications, manufacturing, transportation, and finance. AI methods like Reinforcement Learning (RL) in particular have demonstrated a lot of promise in allowing systems to interact with dynamic environments and learn optimal behaviors on their own.

AI's expanding impact highlights its function as a key technology for creating intelligent, adaptable systems. One such application—an RL-based algorithm intended to control and distribute message burdens among gateways in a real-world communication network scenario—is examined in this section.

3.4.1 State representation

From the agent's point of view, the state in Reinforcement Learning refers to the current circumstance or picture of the surroundings. It includes all of the pertinent data the algorithm requires in order to make decisions at any given time.

```python
# Function to create the state tuple
def get_state(gw_powers, near_gw, operation_cost, current_assigned, moved_device):

    viable_nearby_gw = [gw for gw in near_gw if gw_powers[gw] >= operation_cost]

    resource_availability = min(3, len(viable_nearby_gw))

    best_tech = 0
    for gw in viable_nearby_gw:
        tech_score = {"Ethernet": 3, "Wifi": 2, "GSM": 1}.get(gateway_tech[gw], 0)
        best_tech = max(best_tech, tech_score)

    current_state = 0
    if current_assigned != -1:
        if current_assigned not in near_gw:
            current_state = 1
        elif current_assigned not in viable_nearby_gw:
            current_state = 2
        else:
            current_state = 3

    device_moved = int(moved_device)

    return (resource_availability, best_tech, current_state, device_moved)
```

Figure 3.1. Definition function for the state

The **"get_state()"** function defines the state, as seen in figure 3.1, which is represented as a tuple with four elements:

- **Resource availability:** symbolized by the smallest number between three and the total number of nearby gateways that, depending on power availability, can enable the activity.
- **Best technology score:** a score based on the gateway technology (GSM, WiFi, or Ethernet), with corresponding values awarding or penalizing the option dependent on the connection quality.
- **Current state:** a classification of the assignment's present status that indicates whether the currently linked gateway is still in use, no longer accessible, or one of the more reasonably priced options.
- **Device moved:** a binary signal indicating if the end device's location has changed, a circumstance that could influence the choice of gateway.

3.4.2 Action Definition

The action in reinforcement learning refers to a choice or action that the agent can make in reaction to the environment as it is at that moment. It has a direct impact on how the environment changes and decides the agent's feedback (reward). An action in the context of this algorithm usually translates into the choice of a gateway to which an incoming message should be forwarded. In order to maximize

system performance by reducing congestion and preventing individual node overload, the agent gradually learns which actions result in a more evenly distributed burden across the gateways.

```
# Function to get the possible actions for a given state
def get_possible_actions(gw_powers, operation_cost, near_gw, current_assigned):

    actions = []

    viable_nearby_gw = [gw for gw in near_gw if gw_powers[gw] >= operation_cost]

    if current_assigned != -1 and gw_powers[current_assigned] >= operation_cost:
        actions.append(("MAINTAIN", current_assigned))

    for gw in viable_nearby_gw:
        if gw != current_assigned:
            actions.append(("NEAR_GW", gw))

    if not viable_nearby_gw:
        all_viable_gw = [i for i in range(number_of_gateways) if gw_powers[i] >= operation_cost]
        for gw in all_viable_gw:
            actions.append(("FALLBACK", gw))

    return actions if actions else [("NONE", -1)]
```

Figure 3.2. Function that specifies the course of action the RL may take

The **"get_possible_actions()"** method determines the potential courses of action by weighing the cost of the operation against the energy availability of neighboring gateways, or "near gateways." The algorithm additionally offers a fallback mode that expands the selection to all gateways capable of supporting the operation in the event of an undesirable state.

3.4.3 Decision Mechanism

The process by which an agent chooses the best course of action given the current situation in the environment and its prior knowledge is referred to as decision-making in a reinforcement learning framework. It is the primary method by which the agent engages with the surroundings in order to accomplish its objective. This approach strikes a compromise between exploitation (selecting behaviors that are known to produce large rewards) and exploration (trying new actions to learn their effects). Making decisions is essential to this project since it determines how communications are dispersed among gateways. These choices are crucial to the algorithm's performance since they have a direct impact on the system's capacity to maintain a balanced workload and adjust to changing network conditions.

```python
# Function to select the action using a hybrid approach
def hybrid_action_selection(state, possible_actions, near_gw, current_assigned):

    if not possible_actions or possible_actions == [("NONE", -1)]:
        return -1

    if np.random.random() < epsilon:
        chosen_action = np.random.choice(len(possible_actions))
        return possible_actions[chosen_action][1]

    scores = []
    for action_type, gateway_id in possible_actions:
        greedy_score = 0

        if gateway_id in near_gw:
            greedy_score += 5

        if gateway_tech[gateway_id] == "Ethernet":
            greedy_score += 4
        elif gateway_tech[gateway_id] == "Wifi":
            greedy_score += 2

        if gateway_id == current_assigned:
            greedy_score += 3

        power_ratio = gateway_powers[gateway_id] / GWResource
        greedy_score += power_ratio * 3

        q_value = q_table[state][gateway_id]

        progress = min(1.0, num_messages / 50000)
        q_weight = 0.3 + 0.6 * progress
        greedy_weight = 1.0 - q_weight

        normalized_q = (q_value + 20) / 40
        combined_score = q_weight * normalized_q + greedy_weight * (greedy_score / 10)
        scores.append((gateway_id, combined_score))

    best_gateway = max(scores, key=lambda x: x[1])[0]
    return best_gateway
```

Figure 3.3. Function that specifies how the algorithm make the decision

Based on the current state and a list of potential actions, the **"hybrid_action_selection()"** function is in charge of choosing the best action, or the gateway to which a message should be allocated. This function employs a hybrid decision-making approach that strikes a balance between applying prior information and learning new behaviors by combining exploration and exploitation.

This is how the function works:

1. **Empty Action Check:** The function returns -1 to show that no decision can be taken if there are no actions available or merely a placeholder action (**("NONE", -1)**).

2. **Exploration:** The agent explores by choosing a random action from the list of possible actions, with a probability determined by epsilon (the ϵ-greedy strategy). Instead of depending only on prior experience, this motivates the agent to find potentially superior answers.

3. **Exploitation with Heuristic Enhancement:** In the absence of exploration, the agent assesses every potential course of action using a mix of:
    - **Heuristic (Greedy) Score:** Several domain-specific variables are used to estimate the immediate desirability of an action:
        - Preference for gateways that are near the end device (+5 points).
        - Ethernet connections are preferred over Wi-Fi (+4 vs. +2 points).
        - Bonus points (+3) if the message already has a gateway allocated to it.
        - a weighted contribution determined by the gateway's capacity in relation to its remaining power.
    - **Q-value Score:** The agent represents the long-term expected reward using the learnt Q-value for the specified state and gateway.
4. **Score Combination:** Each action's final score is a weighted sum of the normalized greedy score and the normalized Q-value:
    - The algorithm can switch from rule-based judgments to learned behaviors when the weight swings more heavily toward the learnt Q-values as more messages (up to 50,000) are analyzed.
5. **Best Action Selection:** The best course of action is determined by choosing the gateway with the highest total score.

Early in training, the agent can use predetermined heuristics to make well-informed judgments thanks to this hybrid strategy, which gradually moves toward completely learnt tactics as more experience is gained. It offers a strong balance between autonomous learning and domain knowledge, which is particularly useful in real-world settings where the initial data may be chaotic or sparse.

3.4.4 Q-Table Update

A key component of value-based Reinforcement Learning algorithms like Q-learning is the Q-table update. It symbolizes the process by which the agent updates its predictions of the anticipated future rewards connected to particular acts in particular states in order to learn from its experiences. The Q-table is a data structure that determines the long-term value of selecting a specific action from a given state by mapping state-action pairs to their corresponding Q-values. Using the general rule shown in the picture 3.4, the agent changes the Q-value for that state-action combination each time it performs an action and observes the reward and subsequent state.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right]$$

Figure 3.4. General rule for Q-value update

Where:

- s is the current state,
- a is the action taken,
- r is the reward received,
- s' is the next state,
- α is the learning rate,
- γ is the discount factor,
- $\max_{a'}$ Q(s', a') is the estimated optimal future value from the next state.

Figure 3.5 shows the function that updates the table with every reinforcement learning call; it was created with inspiration from the general rule.

```
# Function to update the Q-table
def update(state, action, reward, next_state, next_possible_actions):

    if next_possible_actions and next_possible_actions != [("NONE", -1)]:
        next_action_values = [(gateway_id, q_table[next_state][gateway_id])
                              for action_type, gateway_id in next_possible_actions]
        next_best_action = max(next_action_values, key=lambda x: x[1])[0]
        next_q = q_table[next_state][next_best_action]
    else:
        next_q = 0

    current_q = q_table[state][action]
    td_error = reward + gamma * next_q - current_q

    visit_count = state_visits.get(state, {}).get(action, 0) + 1
    state_visits.setdefault(state, {})[action] = visit_count

    adaptive_alpha = max(0.05, alpha / (1 + 0.1 * visit_count))

    q_table[state][action] = current_q + adaptive_alpha * td_error
```

Figure 3.5. Function that update the Q-table

Chapter 4

Dataset Overview

4.1 Description

The analysis and construction part of the dataset was conducted by Lorenzo Frangella during his thesis.[11] The principal aim of this study is to conduct a comprehensive analysis and comprehend the behavior of the `Edge2Lora` system in a dynamic, large-scale setting. In this setting, a large number of devices travel throughout a predetermined geographic area, changing the gateways that can receive their LoRaWAN signals as they move.

The lack of an intelligent element for controlling device-to-gateway assignments is a significant drawback of the current `Edge2Lora` solution. This means that the connection is cut off and the onus of processing the device's packets is transferred to the next available gateway when a gateway becomes unavailable, whether because of range restrictions or other reasons. Although this strategy works, it does not maximize system efficiency or dependability in situations involving dense gateway deployment or high device mobility.

Particularly when it comes to data-driven evaluations of gateway assignments in mobile situations, there is a dearth of literature on network management and optimization in LoRaWAN systems. The uniqueness and importance of this work are further highlighted by this gap in the literature.

The project includes the development of a unique dataset that simulates a real-world network situation in order to address this. To create a solid basis for testing and analysis, this entails utilizing location-based data and supplementing it with other datasets. The study intends to offer important insights into `Edge2Lora`'s performance under various scenarios by creating a dataset specifically designed to meet the needs of this research, providing a route for optimizing device-to-gateway interactions in LoRaWAN networks.

4.2 Starting Data

A dataset has to be used as a starting point in order to assess and test the algorithm. To give a clear and accurate depiction of the world, this dataset had to have a collection of devices that moved according to a certain and significant pattern rather than at random. After examining a number of possibilities, it was determined

that using data that showed how taxis moved throughout a city was a viable choice because it provided intriguing and distinct movement patterns.

4.2.1 Roman Taxi Data

The best option for constructing the scenario needed for this work turned out to be the Roman Taxi dataset. This dataset made it possible to create a realistic and thorough depiction of device mobility thanks to positional data that was gathered at regular intervals and an average GPS accuracy of 20 meters. There are 320 taxis in the dataset, and the positional data is based on working hours (about 8 hours a day) rather than round-the-clock surveillance.

The suggested study is based on this dataset, which offers the required mobility patterns to simulate a real-world network situation. The study successfully fills the gap caused by the absence of comparable datasets in the literature by utilizing this comprehensive data.

4.3 Dataset Description

Bracciale et al.'s dataset [8] offers comprehensive positioning data for Rome's taxi industry. Because it records real-world mobility patterns with a high degree of granularity, it is a crucial resource for this research. A thorough temporal resolution of movements is ensured by the dataset's records of taxi positions, which are gathered at intervals average between 7 and 15 seconds.

4.3.1 Dataset Overview

There are 320 cabs in all in the dataset. However, the location data were not constantly collected for a whole day. Rather, it seems that data gathering was restricted to the hours when drivers were really running their cars, which, according to analysis, amounts to about eight hours per day per taxi.

One month of continuous data is provided by the collecting period, which runs from February 1, 2014, to March 1, 2014. Positional data was recorded using GPS technology, which provides an accuracy of about 20 meters. This degree of accuracy is adequate for researching urban mobility trends and how they affect network behavior.

It is crucial to remember that the collection rate is not stated clearly in the dataset description. The dataset is reflective of typical urban traffic patterns rather than continuous tracking, though, as it can be deduced from the data analysis that location recordings were only kept during times when driving was active.

4.3.2 Data Format

The dataset's 5-minute time intervals are called *snapshot*. 288 non-overlapping snapshot files, each reflecting a 24-hour day split into 5-minute halves, make up the reshaped dataset.

Only active taxis—those that registered at least one location at that time—are included in the corresponding snapshot file for each snapshot. Each active taxi

is linked to a single pair of coordinates, selected at random from the collection of places the taxi logged during that particular interval, in order to simplify the representation.

The dataset is guaranteed to be manageable and reflective of a genuine, large-scale deployment situation thanks to our snapshot-based methodology. It offers a useful foundation for assessing `Edge2Lora`'s capacity to manage dynamic, dense networks efficiently.

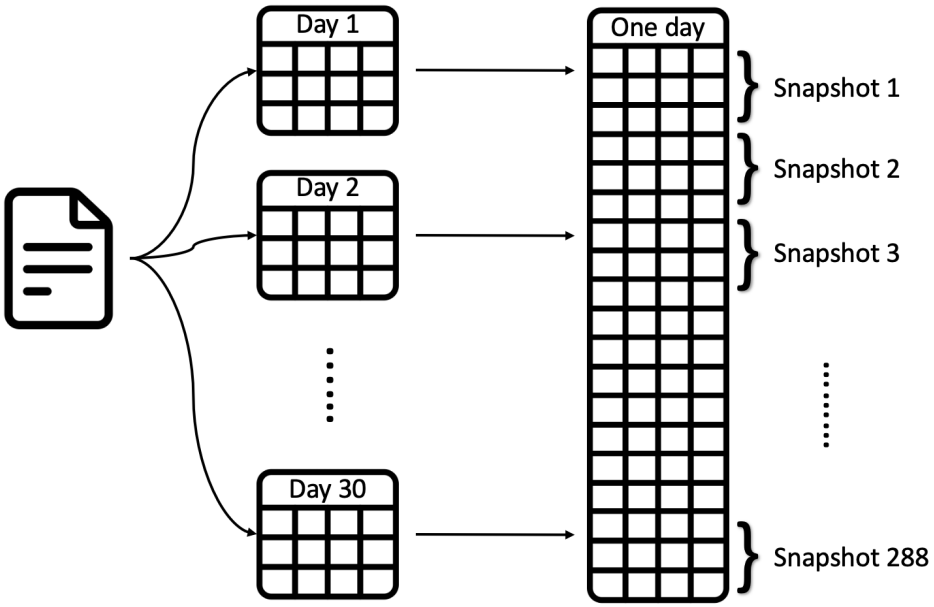


Figure 4.1. Dataset Schema

Taxis are not constantly in operation and do not run every day of the week since drivers are not always operating their cars. Taxis are arranged into eight-hour shifts to guarantee reliable service. Although the three hours are not equally divided among drivers, these schedules are intended to ensure that a taxi service is available at all times of the day. Furthermore, it is vital to extract information on the number of taxis operating at each hour of the day because the number of active drivers can change.

The number of active taxis throughout each hour may be easily extracted from the dataset, which has previously been split into various snapshots, in order to address this. The number of taxis that are available and operating during each given time period can be determined by examining the timestamps linked to each taxi's location. A better grasp of the fleet's operational dynamics and the fluctuations in taxi availability throughout the day is made possible by this information.

The average speed at which the taxis move during the day is another significant feature of the dataset. The pace at which taxis travel can reveal important information about how networks behave. The traveling speed, for instance, can be

used to estimate how frequently a cab or other device shifts its geographic location or passes between coverage zones. Since faster-moving devices could need more frequent gateway reassignments to ensure constant connectivity, this information can be crucial in determining how many gateways should be installed in the network.

These two crucial pieces of information—the average moving speed and the number of taxis operating per hour—can assist enhance the `Edge2Lora` system and are crucial for comprehending the dynamics of the entire network. Better network load forecasts and informed choices about gateway placement, capacity planning, and changeover techniques are made possible by the analysis of these variables. These observations are depicted in the following figure.

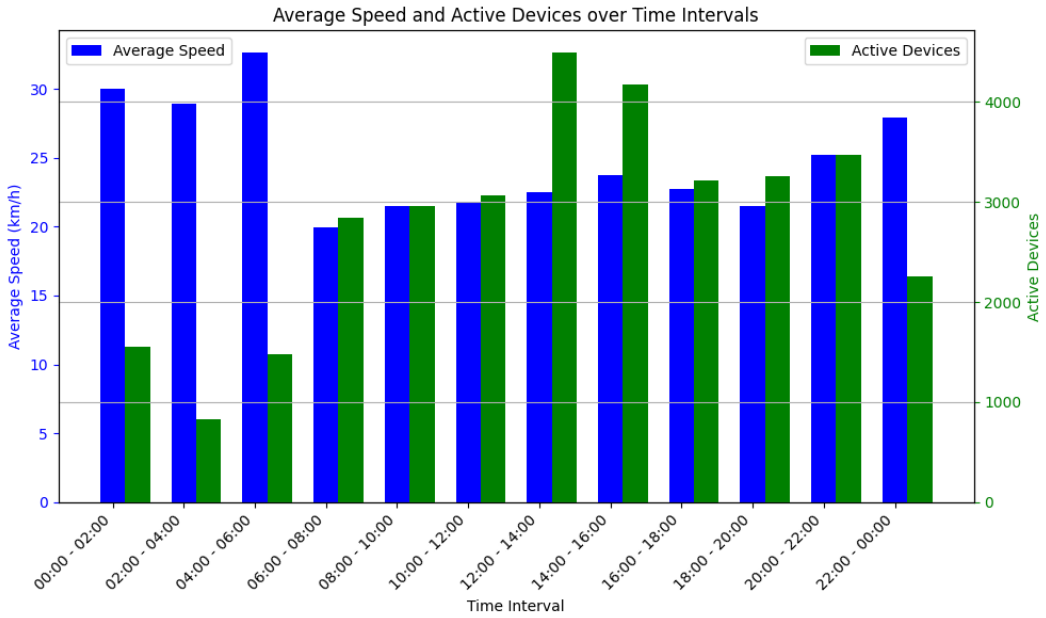


Figure 4.2. Active Taxis and Speed

The figure 4.2 illustrates something that was previously imaginable: the number of active taxis fluctuates throughout the day, peaking between 12:00 and 14:00 and falling at night. The average traveling speed exhibits the reverse pattern, peaking at night and decreasing throughout the day.

The number of active taxis rose from a maximum of 320 (assuming all taxis are operating simultaneously) to a mean value of active devices between 2000 and 3000 as a result of compressing the dataset in a single day. This is significantly more intriguing than it was previously for conducting tests in a large scenario.

4.4 Gateway Deployment

4.4.1 Gateway Deployment Strategy

The geographic layout of the city, the density of taxi activity, and the configurations offered by an established LoRaWAN network provider were some of the considerations that influenced the gateway placement. Three distinct deployment

scenarios were created, each with a different number of gateways, to replicate different network circumstances and scales:

- **Small Deployment (2 Gateways):**
  - centered on covering Rome's main central regions.
  - Perfect for researching low-density gateway setups and how they affect handover performance and coverage.
- **Medium Deployment (20 Gateways):**
  - Expanded coverage to include the central area of Rome.
  - Provides details on the system's scalability in relatively dense deployments.
- **Large Deployment (50 Gateways):**
  - Seeks to provide thorough coverage over the whole city.
  - Especially useful for evaluating how well `Edge2Lora` manages device connections over a dense and widely dispersed network.

4.4.2 Gateway Placement Methodology

The provider's current network was not precisely replicated in the gateway deployment. Rather, it adopted the key locations and density that the provider determined to be ideal, modifying these configurations to fit the taxi dataset's mobility patterns.

Important factors to take into account when placing gates were:

- **Taxi Density Distribution:** The dataset analysis showed that areas with higher taxi movement concentrations were given priority when it came to gateway location.
- **Geographic Relevance:** The greatest deployment scenario especially targeted key roadways, highways, and urban centers, including the port of Civitavecchia and the airports of Fiumicino and Ciampino.
- **Provider Insights:** A realistic and workable deployment model was ensured by using the regions that the provider considered crucial as a guide.

4.4.3 Coverage Insights

Different coverage patterns resulted from the different amount of gateways:

- **2 Gateways:**
  - Restricted coverage that is centered in Rome's center.
  - Designed to evaluate the difficulties of minimal network infrastructure.

- **20 Gateways:**
  - Increased coverage in the center region while excluding the surrounding areas.
  - A sensible ratio of coverage to infrastructure cost.
- **50 Gateways:**
  - Reached almost total coverage over the city of Rome.
  - Enabled a thorough assessment of `Edge2Lora`'s performance in a highly networked setting.

This phased implementation approach guarantees that the research assesses `Edge2Lora`'s flexibility and effectiveness in a range of network configurations, from sparse to dense coverage.

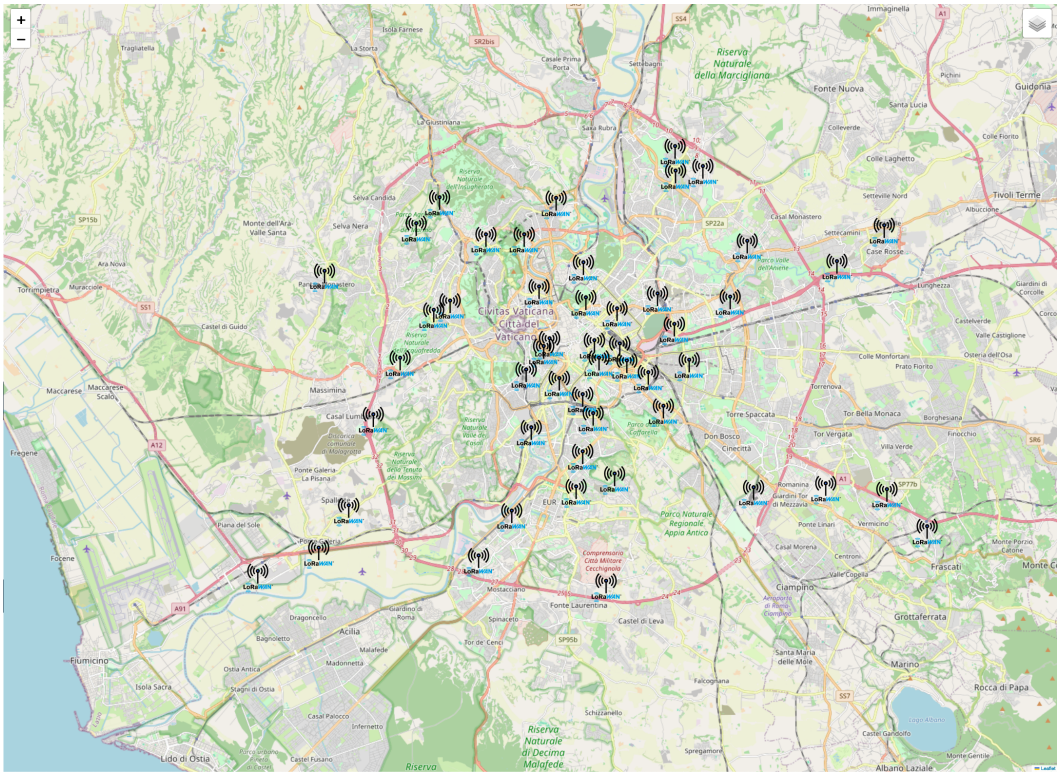


Figure 4.3. Gateways position

Figure 4.3 shows the locations of the gateways in the city of Rome; it includes all 50 of the gateways that are part of the scenario. Let's start with the case where there are two gateways located in the city center. The farther one travels from the heart of Rome, the more gateways there are.

Based on a model that reduces the coverage radius as the distance from the center of Rome grows, Figure 4.4 shows an estimate of the coverage radius of LoRaWAN gateways. Existing literature, especially the study reported in [3], is the source of the reference coverage radius values. LoRa's capacity to traverse vast areas with no

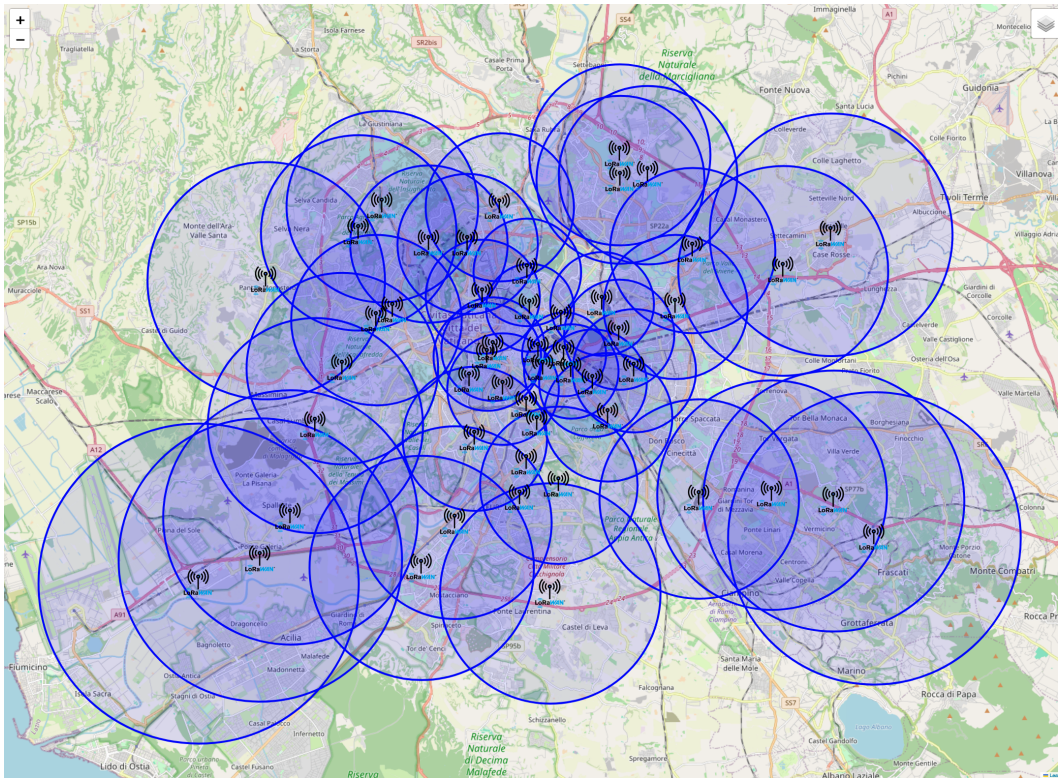


Figure 4.4. Gateways position

infrastructure is demonstrated by the fact that its transmission range in rural areas can reach several kilometers.

Furthermore, according to [5], another study that looks into LoRa transmission power, LoRaWAN can reach transmission ranges of up to roughly 19 kilometers in rural areas. This, however, does not accurately represent urban settings like Rome. The LoRa transmission range is typically much less in more suburban or developed regions, including those that surround the city. The projected coverage radius for these regions is set at roughly 6 kilometers. This discrepancy emphasizes how infrastructure, urban density, and environmental conditions affect LoRaWAN network performance.

Chapter 5

Evaluation of Experimentation Scenarios

Subsequent to the construction of the algorithm and the loading of the dataset, the next phase of the study involved the implementation of Reinforcement Learning and a comparative analysis with Greedy, random and a basic algorithm.

Finding the best processing gateway for individual requests coming from end devices was the original problem formulation for this study. These requests were thought of as distinct occurrences that matched the data entries (rows) in the system snapshots exactly. Reinforcement Learning (RL) was the main methodological strategy chosen to tackle this per-request assignment problem.

But once the RL algorithm was first implemented and certain experimental tests were conducted, it was determined that the problem as originally formulated was not difficult enough to justify the complexity of an RL solution. Using simpler heuristics, it was found that the problem of assigning individual, independent requests to a gateway could be solved deterministically. In particular, preliminary results indicated that a method like the previously stated Greedy algorithm could accomplish efficient, although not ideal, allocation for this specific situation.

In order to find a more difficult situation where the adaptive and learning capabilities of an artificial intelligence technique, such reinforcement learning (RL), would be more suitably justified and advantageous, this realization required a re-evaluation and improvement of the problem scope. As a result, the problem was rephrased. In this updated formulation, incoming messages from endpoints were aggregated into batches or "windows" that were predetermined. As a result, the clever algorithms' goal was changed from choosing the processing location for each individual message to choosing the best gateway for a window of communications. Additionally, depending on changing system conditions and performance goals, this involved the more difficult choice of whether to move an existing window of communications from its present gateway to a different one. This change made the decision-making process much more dynamic and dimensional, which made it a better testbed for the suggested RL technique.

5.1 Simulation Setup

A simulation environment that closely mirrored actual operating conditions was painstakingly created in order to assess the efficacy of the suggested algorithms. A broader dataset of system snapshots served as the source of the simulation's input data. The basis of the simulated message traffic was formed by randomly selecting 16 different snapshot rows from this dataset, each of which represented a distinct message type or event transmitted by an end-device to one or more gateways. After that, this choice was used to produce roughly 145,000 distinct message processing events in total workload. Fifty separate gateways made up the simulated network topology used in these research.

The message events, which were produced from the chosen snapshot rows, were processed sequentially as part of the simulation technique. A dictionary data structure, indexed by the distinct identifier of the originating end-device, was used to organize and track messages. As a result, messages could be logically grouped into "windows" that were specific to each device. The system's detection of a "movement," or change in a device's status or context in relation to its previously processed messages, was a crucial decision point in the simulation. After detecting this, the active load balancing algorithm decided whether to keep the current assignment or move the message window linked to that device to a different gateway.

The method evaluated the resource availability of the currently allocated gateway once a message window for a particular device ended (that is, after a predetermined amount of messages for that device had been aggregated). The necessary resources were allotted (i.e., notionally removed from its available capacity) if the gateway had enough residual capacity to meet the window's processing demands. On the other hand, the algorithm started a procedure to choose a new, better gateway for that window if the resources of the existing gateway were inadequate.

A multi-threaded strategy was used to execute resource management in the simulation in order to replicate the temporal and dynamic features of resource contention and utilization in real-world systems. A timer was connected to each thread, which represents the processing of a message window. This timer system mimicked the limited amount of time needed to process messages. When the timer for a thread ended, indicating that processing for that window had finished, the resources assigned to that job were released and put back into the resource pool of the corresponding gateway. With this method, the transient nature of resource utilization during active message handling was more accurately simulated.

The simulation framework saves algorithmic outputs into two separate Comma-Separated Value (.csv) files for later analysis and performance visualization. The first of these files records a thorough event log, in which each item denotes a distinct message that has arrived and is being handled by the system. The main purpose of this granular data is to produce in-depth visualizations of the behavior of individual end devices and how they interact with the gateways; these will be shown and covered in Section 5.2.

The second.csv file, on the other hand, compiles data at the level of finished message windows. The outcomes of the decision-making process for a particular window are summarized in each row of the later file, together with a breakdown of the related expenses and other pertinent performance indicators. The comparative

performance graphs (such the ones shown in Section 5.3 that show the effectiveness of the various algorithms over the simulated episodes) are mostly constructed using this window-level summary data.

Table 5.1. Description of csc files elements

CSV Column Header	Description
`score`	cost metric for the current window/state.
`dist`	A distance-related metric (e.g., device to selected gateway).
`selected_gw_power`	The configured resource capacity or power level of the gateway that was selected to process the current message window.
`reward`	The numerical reward signal received by the Reinforcement Learning agent following its last action taken in the preceding state, used to guide the learning process.
`mean_gateway_powers`	The average resource capacity or power level calculated across all gateways at the time the current decision was made.
`mean_score`	A running average or smoothed value of the 'score' metric, calculated over a history of preceding episodes or time steps.
`time_of_computing`	The computational time, taken by the algorithm to make the current gateway selection decision.
`selected_gateway`	The unique identifier (ID) of the gateway chosen by the load balancing algorithm for processing the current message window.
`current_assigned`	The unique identifier (ID) of the gateway to which the end-device was assigned *before* the current algorithmic decision was enacted.
`dev_addr`	The unique identifier for the specific end-device whose message is currently being considered.
`moved`	A boolean value ('True' or 'False') indicating whether the end-device changes its position.
`num_messages`	The number of individual messages aggregated within the current message window being evaluated by the algorithm.
`near_matrix`	A list (or representation of a matrix) containing the identifiers of gateways considered to be in proximity to the end-device

5.2 Windowing Logic

The main operational mechanics in the simulated environment are explained in detail in this section. It will mostly concentrate on two related topics. First, a detailed explanation will be given of how device-specific message windows are dynamically filled from the data stream. This will cover the requirements for message aggregation, window initiation, and the circumstances that indicate a finished window that is prepared for processing. This method will be demonstrated using real-world examples to guarantee clarity. Second, the section will give an example of how each of the load balancing algorithms examined in this paper is based on different operational logic. From the perspective of two distinct devices—specifically, the

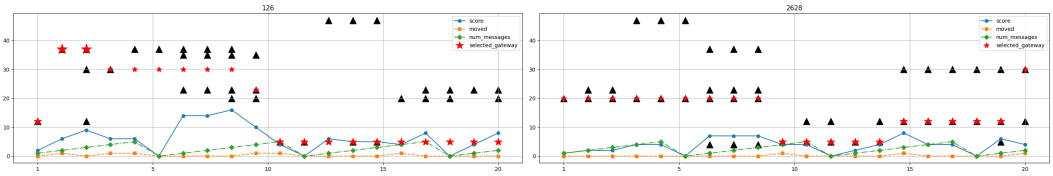
device (126) that moves the most during the simulation and the device (2628) that moves the least, the characteristic behaviors and decision-making processes of these algorithms when faced with particular window states or system conditions will be illustrated using representative examples, elucidating their functional differences and anticipated interactions within the system.

Table 5.2. Condensed Legend for Algorithm Performance Figures

Visual	**Label**	**Description**
Blue line, circles	`score`	Represent the cost of the specific message, comprehends also the cost of moving the windows if GW changes.
Orange dashed line, squares	`moved`	Binary indicator: '1' = device changed position; '0' = no change position.
Green dash-dot line, diamonds	`num_messages`	Number of messages in device's current processing window.
Red star markers	`selected_gateway`	ID of the gateway chosen by the algorithm for the device's window.
Solid black triangles	*Candidate GW.*	Set of available/candidate gateways for selection. (Visual plot element).

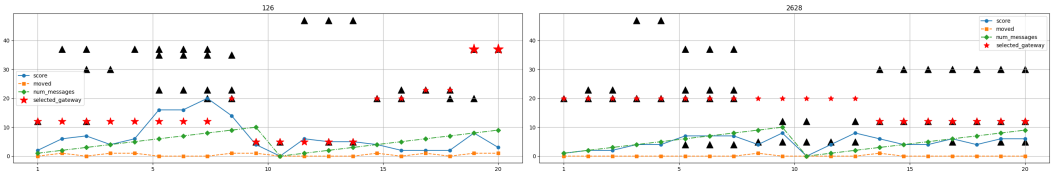
5.2.1 Random

The Random Load Balancing Algorithm's operational behavior for two different end-devices, designated Device 126 and Device 2628, respectively, is demonstrated in the figures below. With the X-axis representing 20 successive decision points (such as message window completions or re-evaluation triggers), these graphics show the order of decisions and system states. Gateway identities, message counts, and an observed system score are among the parameters that are quantified on the Y-axis.



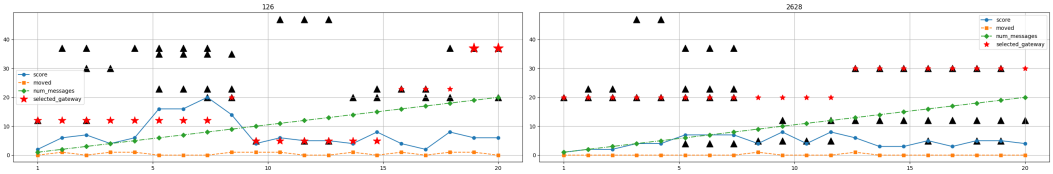
(a) (b)

Figure 5.1. Window size 5 for Random algorithm



(a) (b)

Figure 5.2. Window size 10 for Random algorithm



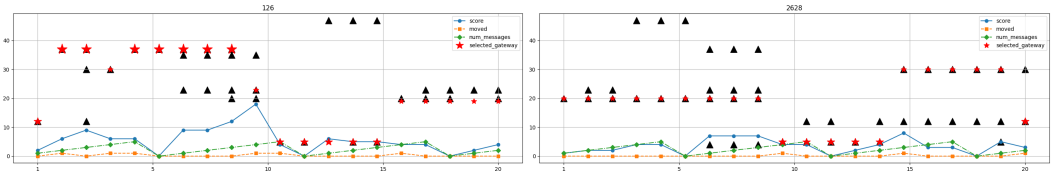
(a) (b)

Figure 5.3. Window size 20 for Random algorithm

These examples highlight the key feature of the Random Algorithm: choosing a gateway, especially when a "move" is started, is a probabilistic process that draws from the pool of possible gateways that are now available. At every choice point, the chosen_gateway (red star) always lines up with one of the accessible gateways (black triangles). These random assignments, not the selection process itself, are what cause the variations in the score and num_messages. The algorithm's stochastic choice to keep the current gateway or choose a new one is appropriately reflected by the moved variable. The algorithm's simplicity and decision-making logic's independence from system state measurements are confirmed by these visualizations.

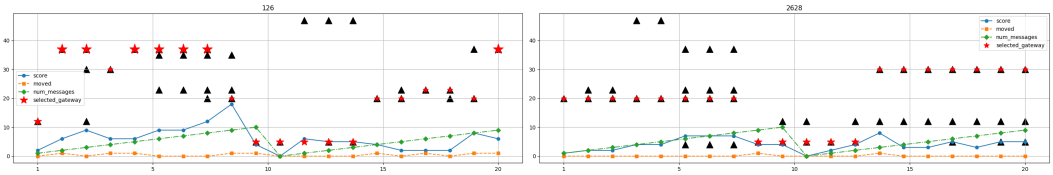
5.2.2 Greedy

We now view instances of the execution trail of the Greedy Load Balancing Algorithm for end-devices 126 and 2628, respectively. On the X-axis, these figures plot the algorithm's choices and the system metrics that are produced throughout a series of 20 distinct decision points or evaluation intervals. Gateway identities and other quantitative measures are represented on the Y-axis.



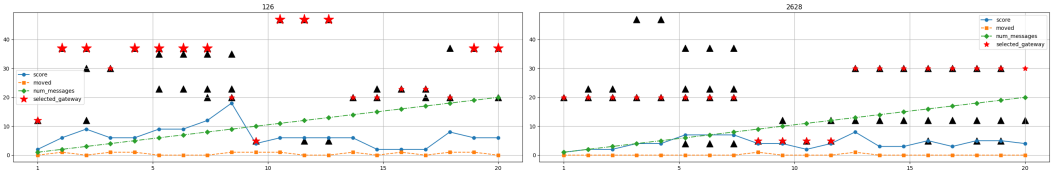
(a) (b)

Figure 5.4. Window size 5 for Greedy algorithm



(a) (b)

Figure 5.5. Window size 10 for Greedy algorithm



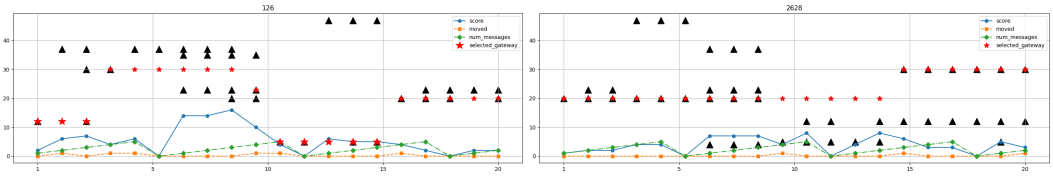
(a) (b)

Figure 5.6. Window size 20 for Greedy algorithm

In contrast to a random approach, these Greedy Algorithm visualizations show a more structured, deterministic decision-making process. A specified set of hierarchical criteria is used to select the selected_gateway (red star) from among the available candidates (black triangles) when a re-evaluation is initiated (shown by moved=1). The particular restriction is that the newly chosen gateway must differ from the one that was previously assigned. With the algorithm trying to choose gateways that meet its optimization goals (such as the best proximity, quality, and resources), the score variable should ideally represent the result of these avaricious decisions. Intervals without re-evaluation triggers are shown by the periods where moved is '0'. When asked to reevaluate, these examples demonstrate the algorithm's approach of selecting locally optimal options from the given alternatives.

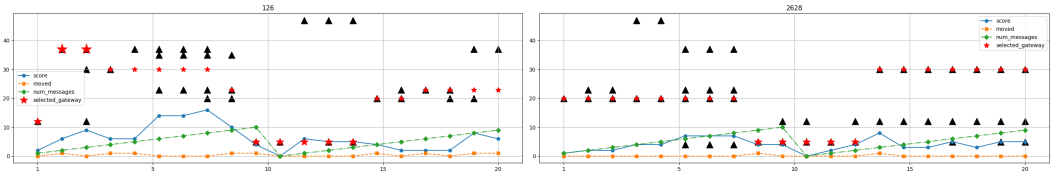
5.2.3 RL

The operational dynamics of the load balancing algorithm based on Reinforcement Learning (RL) are demonstrated in the other images for Device 126 and Device 2628, two example end devices. For 20 successive decision epochs (such as message window processing cycles) on the X-axis, these charts show the algorithm's decisions and the resulting effects on system metrics. These metrics are quantified on the Y-axis.



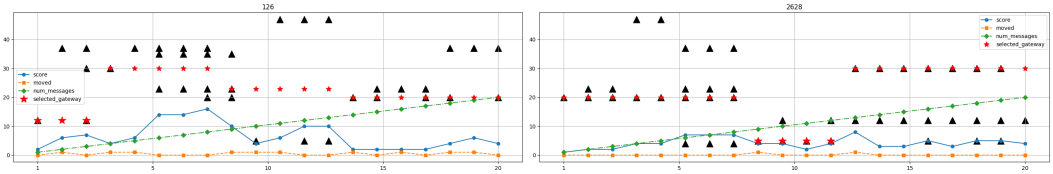
(a) **(b)**

Figure 5.7. Window size 5 for RL algorithm



(a) **(b)**

Figure 5.8. Window size 10 for RL algorithm



(a) **(b)**

Figure 5.9. Window size 20 for RL algorithm

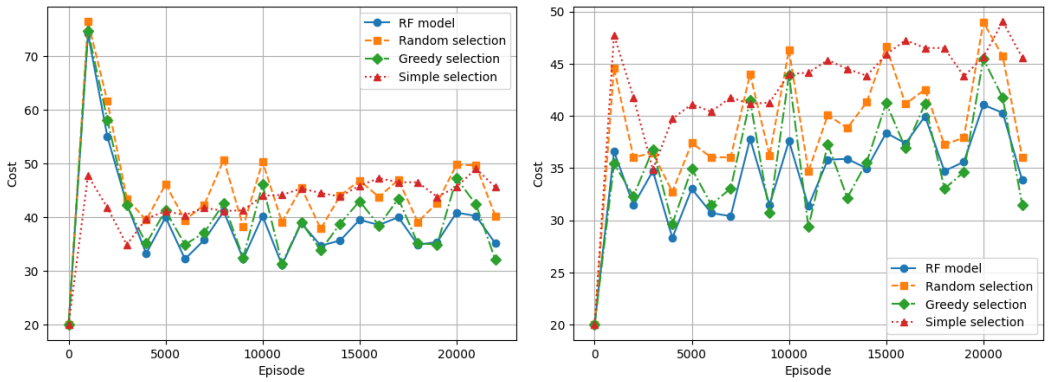
These charts show how the RL algorithm behaves in comparison to more straightforward heuristic or random approaches. A learn policy that aims to maximize the score (or related rewards) over time informs the RL agent's choices about the moved status and selected_gateway. The decisions are meant to be strategically beneficial in the long term rather than just being a fixed reaction to the current situation. Score fluctuations have a direct impact on learning, either encouraging or discouraging particular state-action combinations. An adaptive system learning to navigate a complicated environment by making successive decisions to attain a given target (e.g., sustained high score) is consistent with these 20-epoch snapshots, even though they just offer a glimpse into the agent's behavior.

5.3 Simulation Results and Comparative Performance Analysis

Following the previous sections' descriptions of the experimental setup, simulation environment, and working logic of the various load balancing algorithms—the Simple, Random, Greedy, and Reinforcement Learning (RL) approaches—this section now provides the empirical findings from their implementation. Comparing and quantitatively assessing these algorithms' performance qualities is the main goal here. Individual gateway resource capacities were established at two different fixed levels, 100 units and 300 units, on different experimental runs in these assessments, which were carried out under precisely defined simulation circumstances. This method made it possible to analyze algorithmic performance and adaptability in a range of resource availability scenarios.

5.3.1 Performance Analysis with Window Size 5

The performance of the tested load balancing algorithms—RF model (Reinforcement Learning), Random selection, Greedy selection, and Simple selection—when set up with a message window size of five is shown and examined in this paragraph. Two different gateway resource capacity scenarios are used to evaluate the performance: 100 units (Figure 5.10(a)) and 300 units (Figure 5.10(b)). Plotting 'Cost' versus 'Episode' (which represents development through the simulation or learning iterations) is the main comparative statistic; a lower Cost value denotes better performance.



(a) GWP 100 **(b)** GWP 300

Figure 5.10. Simulation with window size 5

- **Figure 5.10(a):** Comparative performance is shown in this figure when each gateway's resource capacity is restricted to 100 units. A number of important observations can be made:
  - **Initial Cost Spike:** At the start of the simulation (around Episode 0–2500), all methods show a notable initial cost surge. With a cost of roughly 75–80, the Random Selection and Greedy Selection algorithms exhibit the most noticeable rises. The Simple selection process has the

lowest initial peak of the four, at about 48, but the RF model also has a significant initial spike, peaking slightly lower. The system stabilizing, initial resource contention prior to efficient load distribution, or, in the case of the RF model, an initial exploratory phase are all possible explanations for this initial surge.

- **RF Model (Reinforcement Learning):** The RF model has a distinct learning tendency after the initial phase. In the later episodes (post-Episode 5000), its cost typically varies between about 30 and 40, gradually declining and stabilizing at a far lower level than the Random and Greedy selection algorithms. This implies that over time, the RL agent successfully picks up a policy to reduce cost.
- **Random and Greedy Selection:** Throughout the simulation, the cost profiles of the greedy and random selection algorithms are higher and more erratic than those of the RF model in the later phases. Their expenses usually surpass forty and exhibit notable oscillations, frequently exhibiting comparable performance. They typically function within a cost range of approximately 35 to 55 after the initial surge.
- **Simple Selection:** Following its first surge, the Simple selection algorithm has a clear cost pattern, often ranging between 35 and 50. This algorithm works under the assumption that gateway saturation is disregarded, as mentioned in Chapter [see the prior chapter for an explanation of this]. Because of this, its stated "Cost" might not accurately reflect the penalties that other algorithms might experience or attempt to avoid due to resource exhaustion. Because it does not account for the effects of possible overload in its decision-making or cost calculation, this feature may help to explain why its cost does not reach the extreme peaks of Random or Greedy during strong contention.

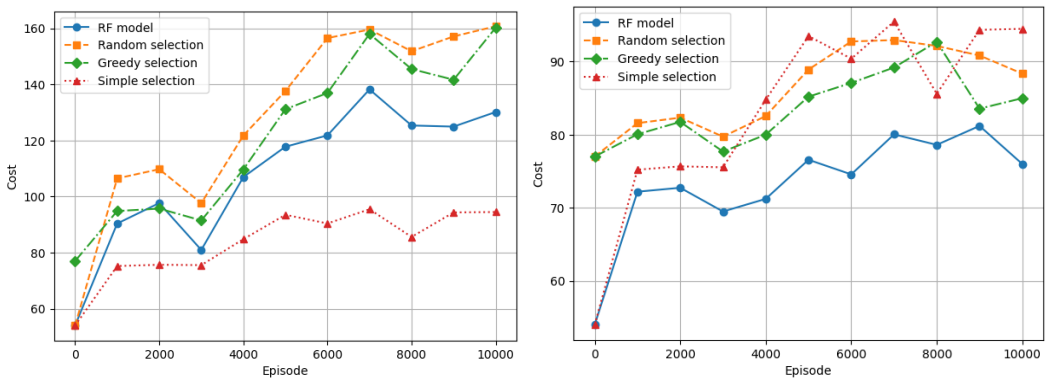
- **Figure 5.10(b):** demonstrates the algorithm's performance with a 300 unit increase in gateway resource capacity:
  - **Reduced Initial Cost Spike:** The reduction of the first cost spike for all algorithms in comparison to the 100-unit capacity scenario is a major effect of more resources. Although there are still spikes, they are noticeably smaller in magnitude. For example, the simple selection peaks about 48, while the greedy and random selections peak between 45 and 48. At about 37, the RF model displays the lowest initial peak.
  - **RF Model (Reinforcement Learning):** The RF model continues to perform better in the later episodes with more resources. It rapidly converges to a lower cost range, typically ranging from about 28 to 38. A more predictable and affordable approach seems to be made possible by the extra resources.
  - **Random and Greedy Selection:** In the later stages of the simulation, these algorithms nevertheless show higher average costs and more volatility than the RF model, while having more resources (their overall cost range is lower than in the 100-unit scenario, usually 30-48).

– **Simple Selection:** The enhanced resource capacity also helps the cost profile of the Simple selection method, which mostly varies between 35 and 48. Its feature of ignoring gateway saturation is still relevant; as resources increase, the practical drawbacks of this assumption—which isn't directly penalized in its own cost—are naturally mitigated, bringing its performance curve closer to or occasionally better than that of Random and Greedy, whose costs may represent attempts to manage resources that are now less constrained.

The reinforcement learning (RF) model continuously shows that it can learn and attain a lower average cost than the greedy and random selection algorithms in both resource capacity situations for a window size of five, especially in the later episodes. The operational cost for all algorithms is often reduced when the gateway resource capacity is increased from 100 to 300 units, and the first cost spikes are considerably lessened. Although the performance of the Simple selection algorithm is enhanced with additional resources, it should be understood that gateway saturation is not taken into consideration, which could obscure actual system stress under heavy load that other algorithms try to solve. The flexibility of the RF model seems to offer a clear benefit in terms of reducing the specified Cost metric during the simulation.

5.3.2 Performance Analysis with Window Size 10

When the message window size is extended to 10, this paragraph describes how well the four load balancing algorithms—RF model, Random selection, Greedy selection, and Simple selection—perform. Once more, two different gateway resource capacity scenarios are used for the evaluation: 100 units (Figure 5.11(a)) and 300 units (Figure 5.11(b)). The primary success measure is still the 'Cost' metric displayed against 'Episode,' where lower values are preferred.



(a) GWP 100 **(b)** GWP 300

Figure 5.11. Simulation with window size 10

- **Figure 5.11(a):** The algorithms behave differently when the gateway resource capacity is limited to 100 units and the window size is set to 10:

  – **Initial Cost and Trend:** The initial cost of all algorithms is roughly 55 units. Most algorithms do not exhibit a sharp initial rise followed by a

drop, in contrast to the W=5 situation. Rather, over the course of 10,000 episodes, both greedy and random selection exhibit a consistent and noteworthy increase in cost, pointing to a growing decline in performance.

- **RF Model (Reinforcement Learning):** Additionally, the cost of the RF model first increases, going from about 55 to about 95 by Episode 2000. After stabilizing its cost, which veers between about 80 and 100 before exhibiting a minor increasing trend once more, it eventually reaches about 130 by Episode 10,000. Even though it is expensive, after the first ramp-up, it usually stays less expensive than Random and Greedy picks.
- **Random and Greedy Selection:** Under these difficult circumstances, these algorithms perform poorly (W=10, GWP=100). By Episode 10,000, their costs had skyrocketed, with both greedy and random selection reaching roughly 160 and 160, respectively. This implies that these less complex heuristic algorithms find it difficult to efficiently manage the load with wider windows and constrained resources, which has serious financial ramifications.
- **Simple Selection:** Out of the four algorithms, the Simple selection method has the lowest cost profile in this extremely limited situation. Its cost goes up from about 55 to a range of about 75 to 95, where it mostly stays for the simulation. The design of the Simple selection algorithm disregards gateway saturation, as was previously explained (Chapter 3). Other algorithms probably suffer significant costs because of resource exhaustion and conflict when the windows are larger (W=10) and the resources are severely limited (GWP=100). Rather than necessarily reflecting improved overall system efficiency or preventing actual resource overload, the Simple selection algorithm's reported reduced cost in this case may simply represent its disregard for these saturation-induced penalties in its cost evaluation.

- **Figure 5.11(b):** Performance changes are visible when the gateway's resource capacity is increased to 300 units while maintaining a window size of 10:
  - **Initial Cost and Trend:** The initial cost, which is about 55 units, stays the same. All algorithms show an initial cost rise, but the size and future trajectory are far more controlled than in the case of 100 units of capacity.
  - **RF Model (Reinforcement Learning):** Effective learning and adaptability with more resources are demonstrated by the RF model. Its cost increases to roughly 75 in the first few episodes (between 2000 and 5000), but then it clearly stabilizes and even slightly decreases, operating primarily between 70 and 80 in the subsequent episodes (after 5000). From about Episode 2500 forward, it continuously achieves the lowest Cost of all the algorithms.
  - **Random and Greedy Selection:** Greedy selection and random selection work significantly better with 300 resource units than with 100. Although they continue to rise from the beginning, their costs generally level off between 80 and 95. They avoid the severe cost escalation observed

in the more limited scenario, although they typically show greater costs than the RF model.

– **Simple Selection:** Additionally, the cost of the Simple selection process first rises to about 95. After that, it varies, frequently outperforming the Random and Greedy approaches in the later phases or performing on par with them, but significantly better than the RF model. Since genuine saturation events are probably less common for all algorithms, the "advantage" of avoiding saturation is less noticeable in its stated cost when resources are more plentiful. This makes it possible for the more flexible RF model to show off its advantages.

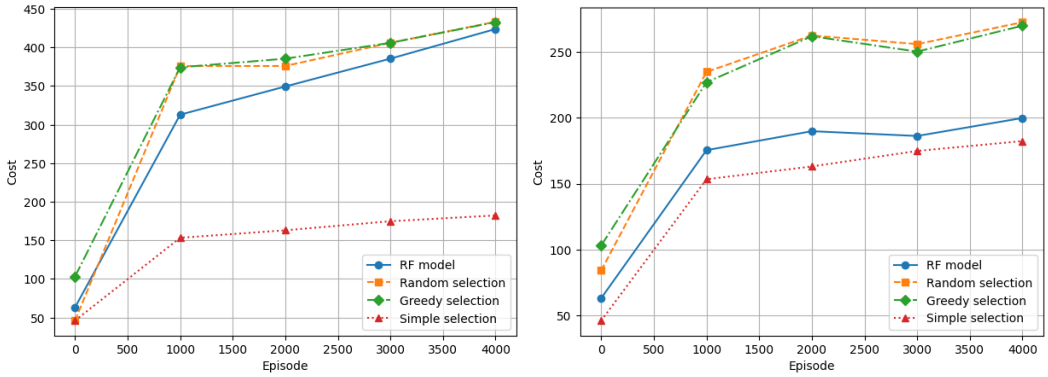
The accessibility of gateway resources is a crucial consideration for a window size of 10.

- **Under limited resources (100 units),** The RF model makes an effort to better control the rising expenses than Random and Greedy, which do a very bad job. Due in large part to its intentional ignorance of saturation penalties, which have a significant impact on other algorithms in this limited setting, the Simple selection algorithm reports the lowest cost.
- **With increased resources (300 units),** Every algorithm works better. With the lowest cost achieved and maintained, the RF approach is unquestionably the most successful. Simple selection likewise functions in this intermediate range, no longer exhibiting a clear cost advantage over Random and Greedy selections, which operate at a moderately higher cost.

These results imply that although the learning capabilities of the RF model offer considerable advantages, particularly in situations where resources are more abundant, the cost of the Simple selection algorithm must be interpreted carefully in light of its underlying assumptions, especially in situations where resources are severely limited and processing windows are longer. Compared to a window size of 5, a window size of 10 often results in higher operating expenses and more difficulties for all load balancing techniques.

5.3.3 Performance Analysis with Window Size 20

With a message window size of 20, this subsection looks at how well the four load balancing algorithms—RF model, Random selection, Greedy selection, and Simple selection—perform in the most taxing scenario. Gateway resource capacity of 100 units (Figure 5.9(a)) and 300 units (Figure 5.12(b)) are taken into account in the evaluation. Noting that the simulation data for this window size extends to 4000 episodes, a shorter duration than for smaller window sizes, the 'Cost' metric (lower is better) is plotted against 'Episode,' possibly reflecting the increased difficulty or earlier stabilization of trends under these conditions. When compared to lesser window sizes, the total cost values are noticeably higher.



(a) GWP 100 **(b)** GWP 300

Figure 5.12. Simulation with window size 20

- **Figure 5.12(a):** It is quite difficult to keep a window size of 20 while lowering the gateway resource capacity to 100 units:
  - **Initial Cost and Extreme Escalation:** While the greedy selection method starts significantly higher, at about 100 units, the RF model, random selection, and simple selection algorithms start with a cost of about 50–60 units. The cost escalation for the RF model, random selection, and greedy selection is consequently very harsh and quick. The cost of the RF model surpasses 300 by Episode 1000, whilst Random and Greedy go close to 375.
  - **RF Model, Random, and Greedy Selection:** According to the Cost metric, these three algorithms show catastrophic performance deterioration under these extremely limited circumstances. Over the course of the 4000 episodes, their costs continue to grow dramatically, reaching levels much beyond 400 (around 420-440). In this case, there is little difference between them; they all seem overburdened by the heavy workload and scarce resources.
  - **Simple Selection:** The cost of the Simple selection method, on the other hand, increases considerably more modestly from its initial beginning point of about 50. By Episode 1000, it has increased to about 150, and by Episode 4000, it has gradually increased even more to about 180. Compared to the other three algorithms, it reports a significantly lower cost. This noticeable distinction highlights the significance of its fundamental design tenet: disregarding gateway saturation. The additional algorithms have significant costs that represent this saturation in a world that is probably marked by severe resource exhaustion (W=20, GWP=100). This parameter makes the Simple selection process appear much more "efficient" because it does not account for saturation penalties in its cost. However, if resources are truly depleted, this does not always transfer into actual system stability or quality of service.
- **Figure 5.12(b):** The following behaviors are seen with a big window size of 20 and a comparatively higher gateway resource capacity of 300 units:

- **Initial Cost and Trajectory:** Greedy selection starts higher at about 100 units, while RF model, random selection, and simple selection start around 50–60 units. These starting prices are comparable to the 100-unit case. Although not as severe as in the 100-unit scenario, all algorithms show a significant increase in cost during the first episodes.
- **RF Model (Reinforcement Learning):** By Episode 1000, the cost of the RF model has risen quickly to around 180. Following this, its cost exhibits a degree of stability, ranging for the last 4000 instances recorded between about 180 and 200. After the initial increase, its Cost continuously remains lower than that of Random and Greedy choices.
- **Random and Greedy Selection:** By Episode 1000, the cost of both greedy and random selection has sharply increased to between 230 and 240. Though more gradually, their costs keep rising, reaching 270–280 by Episode 4000. Even with 300 resource units, these algorithms are unable to handle the tremendous load imposed by W=20, outperforming the RF model and simple selection.
- **Simple Selection:** By Episode 1000, the cost of the simple selection process has similarly increased from its initial value to roughly 155. After then, it increases more gradually until it reaches about 185 by Episode 4000. Its reported cost in this more resource-rich scenario is frequently on level with or marginally superior to the RF model in the subsequent episodes. Although the costs for all algorithms remain high, the availability of 300 resource units may prevent the acute saturation that would more clearly distinguish its cost from algorithms that account for it.

All investigated methods are strongly challenged by the high-intensity effort represented by the window size of 20, which results in much higher costs than those seen with smaller window sizes.

- The system seems overloaded for the RF model, Random, and Greedy algorithms under extremely constrained resources (100 units, Figure 5.12(a)), with their costs rising to extremely high levels. Because it ignores saturation, the Simple selection method provides a far lower and more constant cost. This demonstrates how its stated cost under such pressure is essentially a design artifact and does not accurately represent the level of system overload.

- The RF model exhibits a capacity to manage costs more effectively than Random and Greedy selections, attaining a relatively consistent, if high, cost, with more available resources (300 units, Figure 5.12(b)). Since the enhanced resources probably mitigate the most severe saturation effects, the Simple selection method also returns a competitive cost in this situation that is comparable to the RF model.

These findings imply that the definition of "Cost" and the underlying presumptions of each algorithm become crucial for interpretation for very large window widths, particularly when resources are limited. The Simple selection appears advantageous

by its own cost metric, even though the RF model exhibits learning and adaptive advantages in less extreme conditions or with more resources. For example, W=20 with GWP=100 pushes traditional load balancing approaches (which take resource limitations into account) into high-cost states.

Chapter 6

Conclusions

Low Power Wide Area Networks (LPWANs), the foundation of many extensive Internet of Things (IoT) installations, especially in smart city applications, are the fast developing field in which this thesis is based. The efficient processing of communication from mobile end devices, which adds significant dynamism and complexity to resource management, is a crucial challenge within these networks. In light of this, the present study examined a complex mobility scenario, focusing primarily on improving the state-of-the-art in intelligent network control. For this reason, a large amount of this work was devoted to the design and implementation of a Reinforcement Learning (RL) algorithm. The main objective of this project was to learn more about how contemporary AI technologies might enhance network traffic orchestration, particularly with regard to the fair distribution and handling of requests from mobile endpoints moving through a smart city environment. To evaluate how such an AI-driven method could result in more reliable and effective network operations, realistic scenarios had to be simulated.

6.1 Key Takeaways from the Research

I learned a lot about the challenges of planning and assessing IoT networks throughout this project, especially when it comes to mobility scenarios. In order to develop the RL algorithm, a thorough comprehension of:

- How realistic mobility patterns affect network performance indicators such as packet loss, gateway load, and handover efficiency, as well as how they help simulate device behaviors.
- The significance of optimizing gateway load balancing to prevent request loss and preserve increased system efficiency.
- The challenge of building an AI system capable of handling a real-world scenario.

In addition, I gained a deeper understanding of LoRaWAN architecture, edge computing ideas, and the significance of developing frameworks that overcome the inherent drawbacks of centralized systems. In order to handle new IoT difficulties,

our experience highlighted the need for interdisciplinary techniques that combine knowledge of networking, data science, and machine learning.

6.2 RL algorithm Limitations

Even if the suggested algorithm is well structured and makes a substantial addition to the area, there are still a number of issues that could be fixed:

- **Generalization:** Often very specialized and tailored to a particular topic, reinforcement learning algorithms can also generalize to recognize other scenarios.
- **Transfer of Learning:** It might be challenging to apply knowledge acquired in one setting to another. For every new context, RL frequently needs to be retrained from the beginning.

Resolving these issues would improve the RL algorithm's suitability for testing complex scenarios and enhancing LPWAN performance in practical implementations.

6.3 Future Works

The work in this thesis establishes the foundation for further investigations of Edge2LoRa load balancing and device-gateway assignment optimization. There are numerous avenues for more research, including:

- **Real-World Validation:** The performance and dependability of the improved `Edge4LoRa` architecture would be greatly improved by being implemented in real-world settings. These deployments could also point to areas that need improvement in terms of scalability and refinement.
- **RL Enrichment:** Although the current RL method was developed for the scenarios described in this thesis, it could be enhanced and modified to be useful in additional scenarios.

6.4 Closing Remarks

This thesis shows how edge computing and LoRaWAN networks can be combined to solve important scalability and mobility issues. This study advances IoT networking for dynamic high-density situations by expanding the `Edge2LoRa` framework and creating a realistic RL algorithm. The knowledge acquired here not only confirms that edge-enabled LPWANs are feasible, but it also creates opportunities for further study in the areas of robust data processing, intelligent resource allocation, and practical implementations. In order to build smarter, more adaptable networks for the IoT-driven future, innovation in this field is still ongoing.

Bibliography

[1] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. "Cloud Computing: An Overview". In: *Cloud Computing*. Ed. by Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 626–631. ISBN: 978-3-642-10665-1.

[2] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. "Challenges and Opportunities in Edge Computing". In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[3] Rúben Oliveira, Lucas Guardalben, and Susana Sargento. "Long range communications in urban and rural environments". In: *2017 IEEE Symposium on Computers and Communications (ISCC)*. 2017, pp. 810–817. DOI: 10.1109/ISCC.2017.8024627.

[4] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. "Low Power Wide Area Networks: An Overview". In: *IEEE Communications Surveys & Tutorials* 19.2 (2017), pp. 855–873. DOI: 10.1109/COMST.2017.2652320.

[5] Ramon Sanchez-Iborra, Jesus Sanchez-Gomez, Juan Ballesta-Viñas, Maria-Dolores Cano, and Antonio F. Skarmeta. "Performance Evaluation of LoRa Considering Scenario Conditions". In: *Sensors* 18.3 (2018). ISSN: 1424-8220. DOI: 10.3390/s18030772. URL: https://www.mdpi.com/1424-8220/18/3/772.

[6] YINGJIE LIANG YAPING CUI and RUYAN WANG. "Resource Allocation Algorithm With Multi-Platform Intelligent Offloading in D2D-Enabled Vehicular Networks". In: *IEEE Access SPECIAL SECTION ON EMERGING TECHNOLOGIES FOR DEVICE TO DEVICE COMMUNICATIONS* 7 (2019), pp. 21246–21253. DOI: 10.1109/ACCESS.2018.2882000.

[7] Ivan Fardin. "Design, Implementation and Evaluation of a Gateway-Device Coordination Protocol to enable Edge Computing over LoRaWAN". MA thesis. Sapienza University of Rome, 2021.

[8] Lorenzo Bracciale, Marco Bonola, Pierpaolo Loreti, Giuseppe Bianchi, Raul Amici, and Antonello Rabuffi. *CRAWDAD roma/taxi*. 2022. DOI: 10.15783/C7QC7M. URL: https://dx.doi.org/10.15783/C7QC7M.

[9] Ivan Fardin, Stefano Milani, Francesca Cuomo, and Ioannis Chatzigiannakis. "Enabling Edge Computing over LoRaWAN: A Device-Gateway Coordination Protocol". In: *Proceedings of the 12th ACM International Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications*. DIVANet

'22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 23–30. ISBN: 9781450394826. DOI: 10.1145/3551662.3560926. URL: https://doi.org/10.1145/3551662.3560926.

[10] LoRa Alliance. *LoRaWAN® Specification v1.1*. Available at https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1. Sept. 2023.

[11] Lorenzo Frangella. "Study of scenarios and experimentation of load balancing algorithms for edge computing over LoRawan". MA thesis. Sapienza University of Rome, 2024.

[12] Lorenzo Frangella, Stefano Milani, Domenico Garlisi, and Ioannis Chatzigiannakis. "Enhancing LoRaWAN Networks with Edge Computing: A Demonstration on a Large-Scale Scenario". In: *EWSN* (2024).

[13] Stefano Milani. "Edge2LoRa: A New Paradigm for Enabling Cloud Edge Computing Continuum over LoRaWAN". PhD thesis. Sapienza University of Rome, 2024.

[14] Stefano Milani, Domenico Garlisi, Carlo Carugno, Christian Tedesco, and Ioannis Chatzigiannakis. "Edge2LoRa: Enabling edge computing on long-range wide-area Internet of Things". In: *Internet of Things* 27 (2024), p. 101266. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2024.101266. URL: https://www.sciencedirect.com/science/article/pii/S2542660524002075.

[15] Christian Tedesco. "Analyzing Distributed Processing Applications in Edge Computing Environments: A Case Study with Realistic Implementation and Evaluation using a Lo-RaWAN Architecture". MA thesis. Sapienza University of Rome, 2024.

[16] URL: https://pandas.pydata.org/.

[17] URL: https://www.python.org/.

[18] URL: https://python-visualization.github.io/folium/latest/.

[19] URL: https://numpy.org/.

[20] URL: https://dash.plotly.com/.

[21] URL: https://matplotlib.org/.

[22] URL: https://github.com/Lora-net/packet_forwarder/blob/master/lora_pkt_fwd/readme.md.

[23] URL: https://learn.semtech.com/mod/page/view.php?id=138.

[24] URL: https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT.

[25] URL: https://learn.semtech.com/mod/page/view.php?id=48.

[26] URL: https://www.thethingsindustries.com/docs/hardware/gateways/concepts/udp/.

[27] URL: https://www.thethingsnetwork.org/docs/gateways/packet-forwarder/ttn/.

[28] URL: https://www.chirpstack.io/docs/chirpstack-gateway-bridge/configuration.html.

[29] URL: https://github.com/Lora-net/sx1302_hal/blob/master/packet_forwarder/readme.md.

[30] URL: https://www.thethingsindustries.com/docs/hardware/gateways/concepts/lora-basics-station/.

[31] URL: https://www.researchgate.net/publication/387185523_LoRaWAN_Protocols_in_IoT_Challenges_Innovationsand_Future_Directions.

[32] URL: https://www.preprints.org/manuscript/202412.1577/v1/download.

[33] URL: https://www.milesight.com/company/blog/packet-forwarder.

[34] URL: https://www.thethingsindustries.com/docs/hardware/gateways/concepts/lora-basics-station/.

[35] URL: https://d3tn.com/blog/posts/2024-12-19-lora-dtn/.

[36] URL: https://lever.co.uk/wireless-insights/lorawan-security-vulnerabilities/.

[37] URL: https://www.loriot.io/blog/LoRaWAN-coverage-issues.html.

[38] URL: https://forum.chirpstack.io/t/recommended-cpu-ram-for-gateway-bridge/23682.

[39] URL: https://users.ece.cmu.edu/~koopman/pubs/ray09_embedded_gateway_data_management.pdf.

[40] URL: https://www.thethingsindustries.com/docs/hardware/gateways/troubleshooting/.

[41] Carlo Carugno. "Design, Development and Evaluation of a LoRaWAN Packet Simulator for Testing Realistic Large-Scale Experimentation & LoRaWAN Deployments". MA thesis. Sapienza University of Rome.

[42] *Edge2Lora Github page*. URL: https://github.com/Edge2LoRa.