UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Orquestación de servicios LoRaWAN

Entorno de Kubernetes para su comunicación con pasarela LoRaWAN

Autor
Rubén Orihuela Romero

Directores
Jorge Navarro Ortiz
Natalia Chinchilla Romero

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 16 de junio de 2025

Orquestación de servicios LoRaWAN

Entorno de Kubernetes para su comunicación con pasarela LoRaWAN

Autor
Rubén Orihuela Romero

Directores
Jorge Navarro Ortiz
Natalia Chinchilla Romero

Orquestación de servicios LoRaWAN: Entorno de Kubernetes para su comunicación con pasarela LoRaWAN

Rubén Orihuela Romero

Palabras clave: IoT, LoRaWAN, Kubernetes, clúster, ChirpStack, gateway, servidor, pod, deployment, service, volume, configmap.

Resumen

En los últimos años, el Internet de las Cosas (IoT) ha tenido una gran evolución, permitiendo conectar una gran cantidad de dispositivos entre sí y con las personas. Debido a su avance, ha cobrado gran relevancia el protocolo *Long Range Wide Area Network* (LoRaWAN) para redes Low *Power Wide Area Network* (LPWAN), que facilita la comunicación a larga distancia y un bajo consumo de energía.

A su vez, en plena era de la transformación digital, la automatización de redes mediante orquestadores ha cobrado una gran relevancia por su capacidad para gestionar redes distribuidas de forma ágil y escalable, así como por su capacidad de reducir tiempos de despliegue.

En este proyecto, se describe la integración de una red LoRaWAN mediante un sistema de orquestación de contenedores que despliega las aplicaciones en los nodos de un clúster, permitiendo escalar la red y gestionarla reiniciando automáticamente los contenedores que fallen para garantizar la disponibilidad. Se explicará la configuración e implementación de los componentes de red, además de una serie de pruebas de conectividad para demostrar el correcto funcionamiento.

LoRaWAN services orchestration: Kubernetes environment for communication with LoRaWAN gateway

Rubén Orihuela Romero

Keywords: IoT, LoRaWAN, Kubernetes, clúster, ChirpStack, gateway, servidor, pod, deployment, service, volume, configmap.

Abstract

In recent years, the Internet of Things (IoT) has undergone significant developments, enabling a large number of devices to connect to each other and to people. Due to its advancements, the Long Range Wide Area Network (LoRaWAN) protocol for Low Power Wide Area Networks (LPWAN) has gained significant relevance, facilitating long-distance communication and low power consumption.

At the same time, in the midst of the digital transformation era, network automation through orchestrators has gained significant importance due to its ability to manage distributed networks in an agile and scalable manner, as well as its ability to reduce deployment times.

This project describes the integration of a LoRaWAN network using a container orchestration system that deploys applications to cluster nodes, enabling network scaling and management by automatically restarting failed containers to ensure availability. The configuration and implementation of network components will be explained, along with a series of connectivity tests to demonstrate proper operation.

Yo, **Rubén Orihuela Romero**, alumno de la titulación de Ingeniería de Tecnologías de Telecomunicación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75928553X, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Rubén Orihuela Romero

Granada a 16 de junio de 2025.

D. **Jorge Navarro Ortiz**, profesor titular del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Natalia Chinchilla Romero**, estudiante de doctorado del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Orquestación de servicios LoRaWAN, Entorno de Kubernetes para su comunicación con pasarela LoRaWAN***, ha sido realizado bajo su supervisión por **Rubén Orihuela Romero**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 16 de junio de 2025.

Los directores:

Jorge Navarro Ortiz **Natalia Chinchilla Romero**

Agradecimientos

En primer lugar me gustaría agradecer a mis padres y mi hermano, por el apoyo recibido durante toda mi etapa como estudiante durante todos estos años. Pues sin ellos no hubiese sido posible formarme y desarrollarme como estudiante.

También agradecer a todos mis compañeros del grado, pues todos estos años nos hemos apoyado entre nosotros y han sido parte muy importante en mi camino. Personas nuevas en mi vida que siguen presentes y otras de las que quedan en el recuerdo muchos momentos pasados juntos.

Por último, agradecer a mi tutor Jorge Navarro, al que le debo esta elección de TFG y la oportunidad de aprender más sobre herramientas muy presentes en el mundo profesional actual.

Índice general

Índice de figuras

Capítulo 1

Introducción

De modo introductorio se va a exponer el contexto actual de las redes LoRaWAN y la orquestación de sus servicios, los motivos de la realización de este proyecto y los objetivos que se pretenden conseguir.

1.1. Contexto y motivación

En el mundo actual, con la expansión del Internet de las cosas (IoT), se ha visto impulsada la necesidad de buscar métodos para realizar comunicaciones eficientes y de bajo consumo. En este contexto, han emergido las redes LoRaWAN como una tecnología que reúne los requisitos de conexión de dispositivos a larga distancia con un bajo consumo de energía. Sin embargo, para desplegar y gestionar estas redes, se presentan desafíos como la escalabilidad y la uniformidad en la configuración de múltiples componentes [1].

Es por ello, que este proyecto se centra en la automatización del despliegue de redes LoRaWAN mediante el orquestador de contenedores Kubernetes, que se trata de una herramienta que permite gestionar de una manera eficiente aplicaciones distribuidas. Con este proyecto, se busca diseñar e implementar una arquitectura que optimice y simplifique el proceso de despliegue de la red, para posteriormente evaluar su rendimiento en términos de tiempo de despliegue y utilización de recursos.

Entre los diferentes motivos que nos llevan a implementar la automatización de despliegues en redes LoRaWAN mediante Kubernetes, están los beneficios como la reducción de los posibles errores humanos, la aceleración en los tiempos de despliegue y la mejora que supone para la consistencia de la configuración.

Los beneficios que nos ofrece utilizar contenedores son [2]:

- Agilidad en la creación y despliegue de aplicaciones, brindando una mayor facilidad y eficiencia al crear imágenes de contenedor en lugar

de máquinas virtuales.

- Desarrollo, integración y despliegue continuo, lo que permite que la imagen de contenedor se construya y se despliegue de forma frecuente y confiable, facilitando los *rollbacks*, ya que la imagen es inmutable.
- Separación de tareas entre *Dev* y *Ops*, ya que permite crear imágenes de contenedor al compilar y no al desplegar, desacoplando la aplicación de la infraestructura.
- Observabilidad, presentando la información y métricas del sistema operativo, así como la salud de la aplicación y otras señales.
- Consistencia entre los entornos de desarrollo, pruebas y producción. Por ejemplo, la aplicación funciona de igual forma localmente y en la nube.
- Portabilidad entre nubes y distribuciones.
- Administración centrada en la aplicación, lo que eleva el nivel de abstracción del sistema operativo y del hardware virtualizado a la aplicación, que funciona en un sistema con recursos lógicos.
- Las aplicaciones se separan en piezas pequeñas e independientes (microservicios), que pueden ser desplegadas y administradas de forma dinámica. No como ocurre en una aplicación que opera en una sola máquina de gran capacidad.
- Aislamiento de recursos, que permite que el rendimiento de la aplicación sea más predecible.
- Utilización de recursos, que permite mayor eficiencia y densidad.

1.2. Objetivos

El objetivo principal que se busca conseguir es diseñar e implementar un sistema automatizado para el despliegue de una red LoRaWAN, haciendo uso del orquestador de contenedores Kubernetes, para posteriormente realizar una monitorización de su rendimiento con el fin de evaluar la eficiencia y escalabilidad del despliegue.

Otros objetivos que se pretenden conseguir son:

- Desarrollar una arquitectura que contenga los componentes clave de LoRaWAN (servidor de red, servidor de aplicación, pasarela y mota) en un entorno de contenedores.
- Identificar y seleccionar las herramientas necesarias para implementar la arquitectura.

- Configurar un clúster de Kubernetes para orquestar y gestionar los contenedores que contienen los componentes de red.
- Crear *scripts* y configuraciones automatizadas, con el fin de desplegar los contenedores de manera eficiente y reproducible.
- Diseñar *dashboards* y alertas que permitan visualizar el rendimiento de la red en tiempo real y detectar posibles fallas.

Capítulo 2

Estado del arte

En este capítulo se van a diferenciar tres puntos. Un primer punto, donde se van a tratar las distintas alternativas para desplegar una red LoRaWAN, un segundo punto donde se tratan las distintas tecnologías de orquestación y despliegue de contenedores, y por último, las herramientas seleccionadas para el proyecto y su motivo.

2.1. Alternativas para desplegar redes LoRaWAN

A continuación, se muestran las distintas opciones tecnológicas para desplegar la red LoRaWAN. Las cuales son de código abierto.

2.1.1. Loriot

Loriot es una empresa que proporciona una solución para desplegar, operar y gestionar redes LoRaWAN [3]. Facilita servidores de red que hacen uso de la tecnología LoRaWAN o de mioty, que se trata de una tecnología LPWAN orientada al IoT [4].

Las arquitecturas con un servidor de red LoRaWAN o mioty son semejantes, sólo varían ciertos aspectos. En LoRaWAN se usan *gateways* y sensores, mientras que en mioty se utilizan estaciones base y dispositivos finales. Si se utiliza el servidor LoRaWAN, se encarga de controlar la infraestructura, la seguridad de la red, el encaminamiento de los mensajes y la autenticación de los terminales [5]. Por su parte, al utilizar el servidor de red mioty, este facilita el manejo de la red, la recolección de datos de los terminales de la red, el procesamiento de los datos, la seguridad, la encriptación y la integración e interoperabilidad con otros sistemas y plataformas [6]. Ambas opciones de red cuentan con un servidor de aplicación que permite analizar los datos de la red. En la figura 2.1 se muestra la arquitectura del servidor de red mioty:

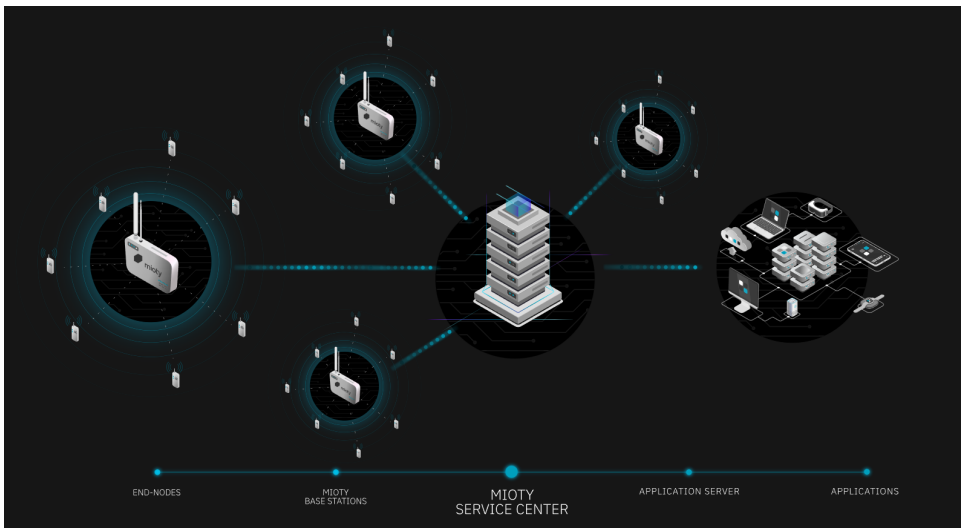

Figura 2.1: Arquitectura servidor de red mioty [7]

2.1.2. ChirpStack

ChirpStack es un servidor de red y de aplicación LoRaWAN utilizado para configurar tanto redes públicas como privadas. ChirpStack proporciona una interfaz web en la que se pueden gestionar los *gateways*, los dispositivos o configurar integraciones de datos con los distintos proveedores de nube, bases de datos y servicios empleados para gestionar los datos proporcionados por los dispositivos [8].

Tiene la capacidad de soportar dispositivos LoRa tanto de clase A, clase B y clase C. En cuanto a su arquitectura, como se puede observar en la figura 2.2, consta de un *Gateway Bridge* que se conecta a los *gateways* LoRa mediante el puerto UDP 1700, que envía los paquetes al resto del servidor. Luego, contiene un bróker MQTT en el puerto 1883 o 8883 si se usa MQTTS, que permite usar el protocolo MQTT para enviar mensajes al servidor [9].

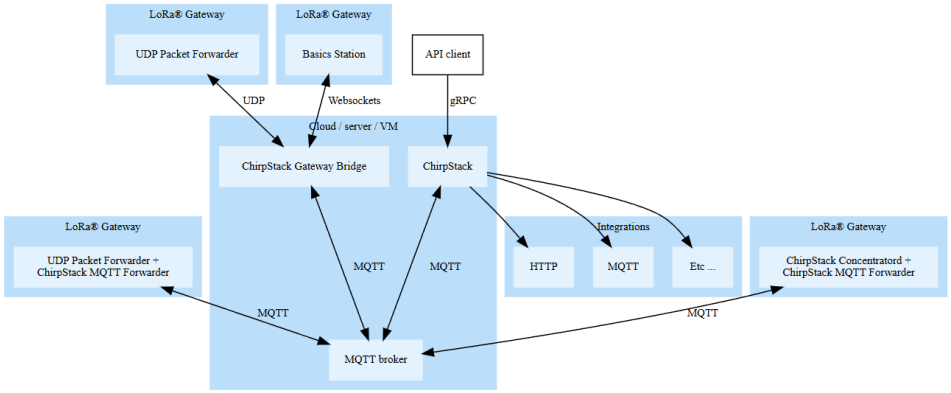

Figura 2.2: Arquitectura servidor de red ChirpStack [9]

2.1.3. The Things Network

The Things Network (TTN) ofrece una serie de herramientas de código abierto y una red global para construir aplicaciones de IoT con un bajo coste, con la máxima seguridad y escalable [10]. TTN sigue una arquitectura de microservicios por API, ideal para alta disponibilidad y confiabilidad [11].

Entre los componentes principales de TTN, podemos encontrar un servidor de red y de aplicación, donde podemos autenticarnos a través de su *Join Server*. También consta del *Identity Server*, que registra todas las entidades que forman parte de la red. La red incluye un bróker MQTT que realiza una función parecida a la de ChirpStack. En la figura 2.3 se muestra su arquitectura de red [11].

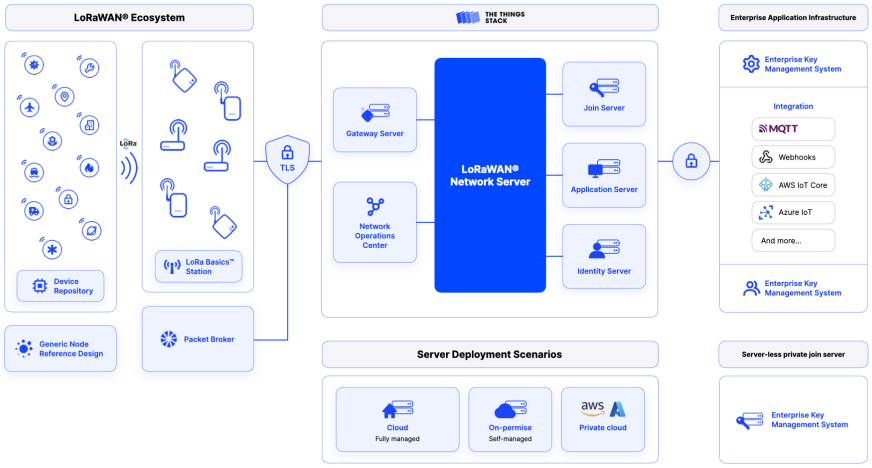

Figura 2.3: Arquitectura de The Things Network [11]

2.2. Tecnologías de orquestación y despliegue de contenedores

Seguidamente, se muestran las distintas opciones tecnológicas de orquestación y despliegue de contenedores.

2.2.1. Docker Swarm

Docker Swarm es un orquestador de contenedores nativo de Docker. Está integrado con Docker, lo que facilita su uso al aprovechar las herramientas de Docker sin tener que recurrir a otras interfaces o comandos. Por esta razón, es una buena opción para proyectos pequeños, ya que facilita la gestión y configuración. Por el contrario, en lo que se refiere a la escalabilidad y gestión de fallos, no es tan potente como otras opciones y resulta insuficiente para proyectos IoT que requieran una gestión y escalado avanzados [12].

A continuación, en la figura 2.4 se muestra su arquitectura:

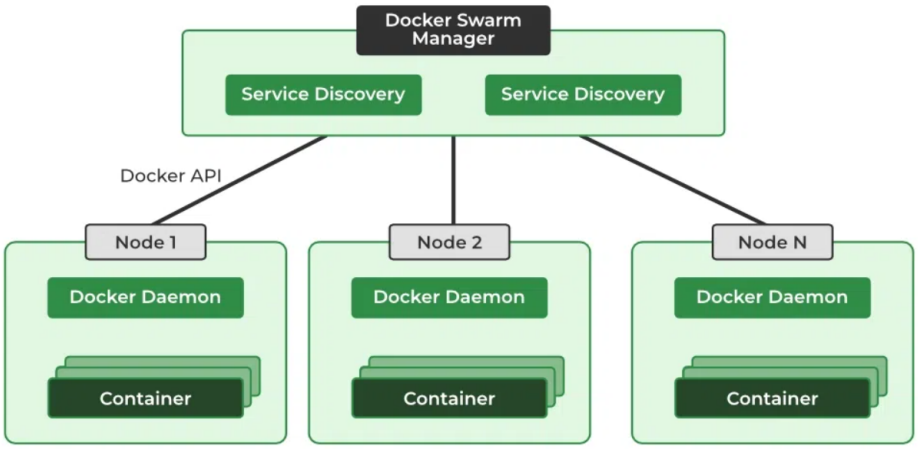

Figura 2.4: Arquitectura orquestador Docker Swarm [13]

2.2.2. Apache Mesos

Apache Mesos es una plataforma de gestión de clústeres diseñada para la gestión y el escalado de aplicaciones a gran escala. Mesos está diseñado para tener una gran eficiencia en la distribución de recursos en clústeres de gran tamaño.

Entre las ventajas más notorias de Mesos, está su capacidad para soportar múltiples *frameworks* de orquestación en un tan solo clúster, incluyendo Kubernetes y Marathon, lo que permite a las organizaciones gestionar aplicaciones en contenedores y no contenedores (como Hadoop) de manera unificada. Pese a la gran potencia y flexibilidad que presenta Mesos, este es bastante complejo de configurar y operar en comparación a otros orquestadores como Kubernetes o Docker Swarm. Su complejidad la hace menos interesante para proyectos pequeños o que cuenten con personal con poca experiencia técnica [14].

A continuación, en la figura 2.5 se muestra su arquitectura:

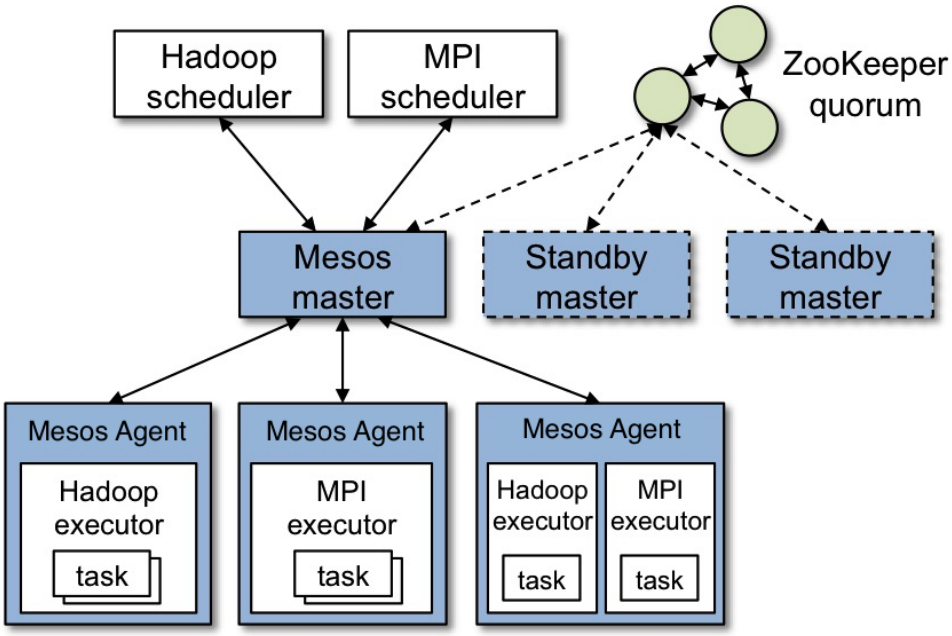

Figura 2.5: Arquitectura orquestador Apache Mesos [15]

2.2.3. Kubernetes

Kubernetes es en la actualidad el orquestador de contenedores más utilizado en entornos de producción, sobre todo en arquitecturas distribuidas y proyectos a gran escala como son los relacionados con IoT. Kubernetes en un principio fue desarrollado por Google, para posteriormente ser donado a la *Cloud Native Computing Foundation* (CNCF). Su uso ha ido creciendo cada vez más debido a su capacidad para gestionar aplicaciones en contenedores de manera automatizada y escalable.

Entre las características de Kubernetes está la gestión automática de despliegues, donde facilita la implementación, actualización y escalado de aplicaciones en contenedores sin que hayan interrupciones. Para ello, implementa los Deployments y Replicasets que permiten que las aplicaciones estén disponibles y se realicen las actualizaciones sin interrupciones. Otra característica es la escalabilidad que presenta, que es muy útil en el contexto de IoT, permitiendo escalar aplicaciones horizontalmente y verticalmente de manera automática dependiendo de la carga que haya. También, Kubernetes contiene mecanismos de autocorrección y recuperación ante fallos. Si un contenedor falla, Kubernetes de forma automática reinicia los contenedores afectados en otro nodo disponible. Por último, otra característica que presenta Kubernetes es que no depende de la infraestructura que haya, por lo tanto, permite que las aplicaciones se puedan desplegar y gestionar en varias

nubes o localmente. Estas características permiten una gran disponibilidad en los proyectos que la requieran [16].

A continuación, en la figura 2.6 se muestra su arquitectura:

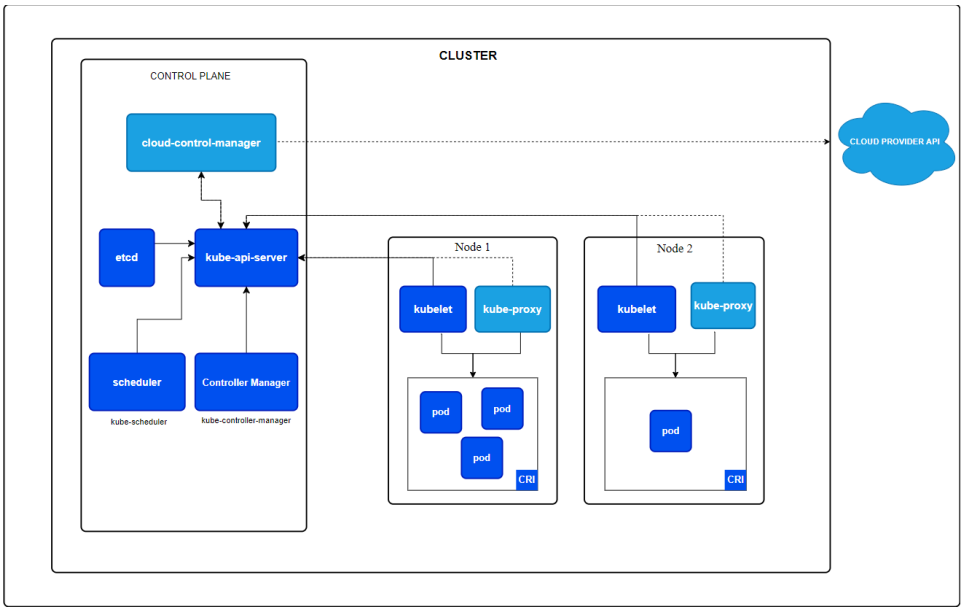

Figura 2.6: Arquitectura orquestador Kubernetes [17]

2.3. Herramientas seleccionadas

Una vez comentadas las distintas alternativas existentes, se va a exponer cuáles de ellas se han escogido para el despliegue de la red LoRaWAN y la orquestación de contenedores.

Para el despliegue de la red LoRaWAN se ha seleccionado la herramienta ChirpStack. Esta elección se debe a que ChirpStack ofrece más control de la red, permite el uso de nuestros propios dispositivos, más cantidad de ellos y se trata de una herramienta gratuita de código abierto.

Por su parte, para la orquestación y despliegue de contenedores se ha seleccionado el orquestador Kubernetes. Esta elección se ha realizado en base a varios aspectos a favor de Kubernetes, como su gestión automática de despliegues, sus mecanismos de autocorrección o su adecuación a los entornos IoT, sin mencionar que Kubernetes se trata de la tecnología de orquestación más presente en la actualidad.

Capítulo 3

Fundamentos teóricos

En este capítulo, se van a explicar los fundamentos teóricos de LoRaWAN y el orquestador Kubernetes.

3.1. Conceptos de LoRaWAN

LoRaWAN (*Long Range Wide Area Network*) define un protocolo de comunicación destinado a redes de baja potencia y larga distancia, diseñado específicamente para aplicaciones IoT, ya que apunta a sus requisitos clave como pueden ser la comunicación bidireccional, la seguridad extremo a extremo, la movilidad y los servicios de localización. LoRaWAN opera en bandas de radiofrecuencia sin licencia y tiene un largo alcance (hasta 15 km en áreas rurales), bajo consumo de energía y la capacidad de conectar una gran cantidad de dispositivos a una sola red, permitiendo una red escalable y flexible para diversas aplicaciones de IoT [18].

3.1.1. Arquitectura de la red LoRaWAN

La arquitectura red de LoRaWAN está implementada en topología estrella, en la cual las pasarelas (*gateways*) comunican los dispositivos finales (denominados terminales) y el servidor de red.

Tal y como se muestra en la figura 3.1, los *gateways* se conectan al servidor haciendo uso de conexiones IP, retransmitiendo las tramas RF. A su vez, la comunicación inalámbrica aprovecha el largo alcance de la capa física LoRa y realiza un enlace de un solo salto entre el dispositivo terminal y los *gateways*. El sistema soporta la comunicación bidireccional y el direccionamiento de multidifusión, que permite hacer un uso eficiente del espectro durante tareas como actualizaciones de *firmware* por aire (FOTA) [19].

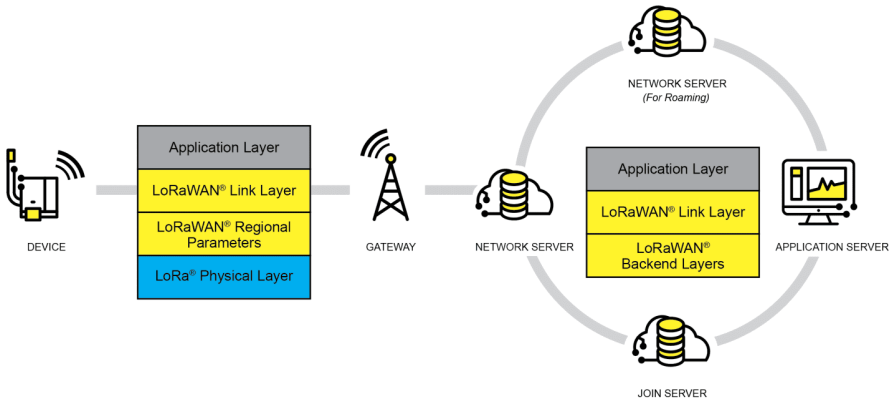

Figura 3.1: Arquitectura red LoRaWAN [19]

3.1.2. Seguridad en LoRaWAN

En la figura 3.2 se muestra cómo entre el terminal y el servidor de red los mensajes constan de integridad y se autentican mediante una *NwkSKey*. Por su parte, entre el terminal y el servidor de aplicación, los mensajes van encriptados mediante una *AppSKey*.

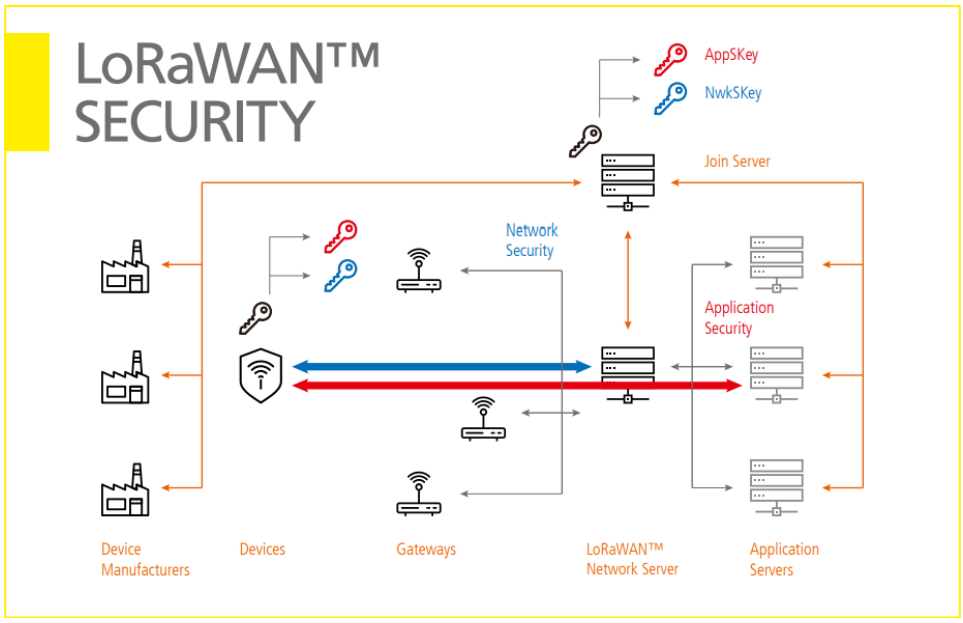

Figura 3.2: Seguridad red LoRaWAN [20]

Al enviar una trama, primero se realiza una encriptación simétrica con la *AppSKey*, para posteriormente, añadirle los parámetros de la autenticación. En la autenticación se añade el MIC (*Message Integrity Control*), que depende de la trama encriptada y de la *NwkSKey*. Una vez llegue la trama al servidor de red, si la *NwkSKey* coincide con la del terminal, el MIC debe ser el mismo. Cuando llega al servidor de aplicación, al usarse una encriptación simétrica, la *AppSKey* debe ser en este caso la misma con la que se cifró el mensaje [20]. En la figura 3.3 se muestra un ejemplo de trama de seguridad LoRaWAN:

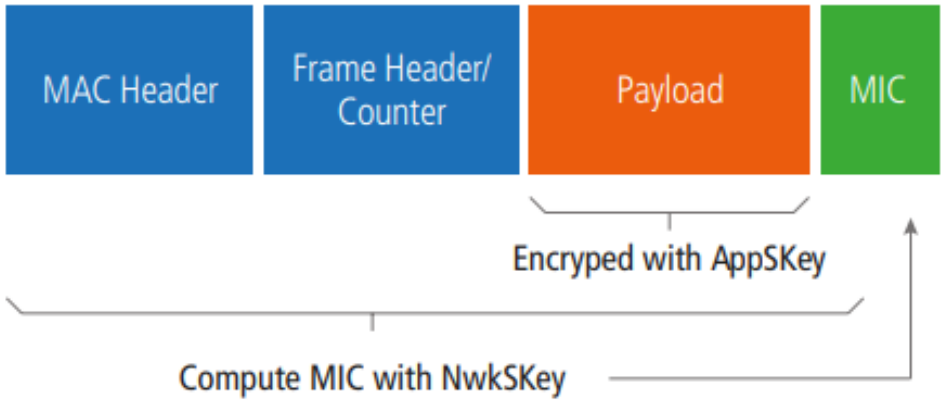

Figura 3.3: Trama seguridad LoRaWAN [20]

Para que un dispositivo pueda enviar y recibir mensajes debe estar activado. Independientemente del tipo de activación, el dispositivo contiene la siguiente información: *Device Address (DevAddr)*, *Application Identifier (AppEUI)*, *Network Session Key (NwkSKey)* y *Application Session Key (AppSKey)*.

Tal y como se indica en la figura 3.4, existen 2 tipos de activación [21]:

- **Over-The-Air Activation (OTAA)**: el dispositivo final crea una nueva sesión, genera on el servidor de red una nueva dirección y las claves de sesión (*NwkSKey* y *AppSKey*). El dispositivo deriva dichas claves de sesión durante el proceso de unión mediante dos tramas denominadas *Join-request* y *Join-accept* utilizando la siguiente información: identificador único de dispositivo (*DevEUI*), identificador de aplicación de destino (*AppEUI*) y clave AES de 128 bits (*AppKey*).
- **Activation By Personalization (ABP)**: es menos seguro que OTAA y el dispositivo debe de ser previamente personalizado de los siguientes parámetros: dirección del dispositivo (*DevAddr*), identificador de la aplicación (*AppEUI*), clave de sesión de red (*NwkSKey*) y clave de sesión de la aplicación (*AppSKey*).

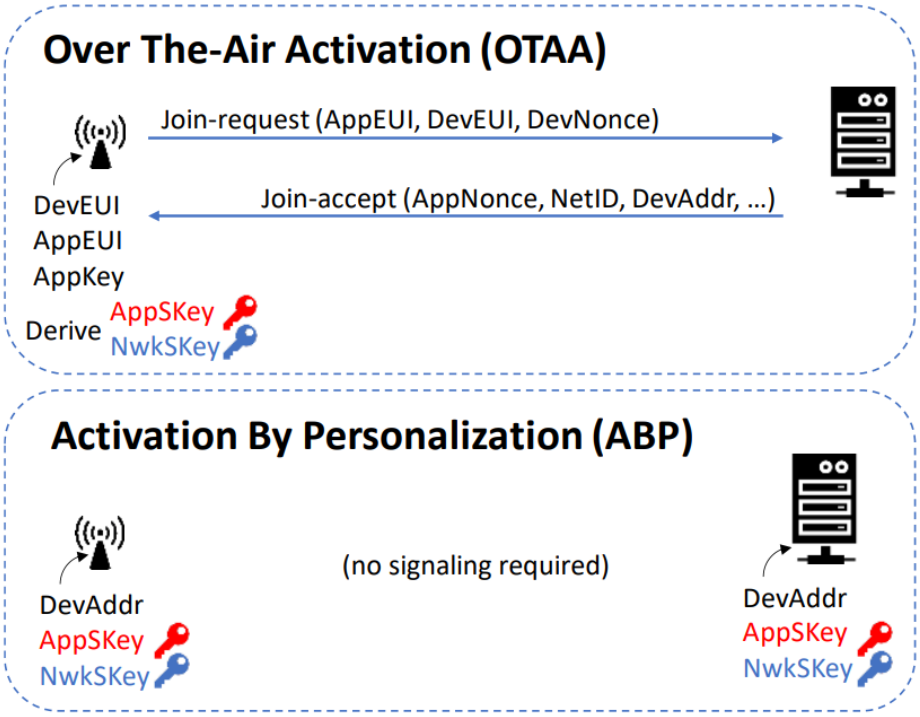

Figura 3.4: Activación OTAA y ABP [21]

3.1.3. Clases de dispositivos LoRaWAN

Los terminales se clasifican en 3 clases, su consumo y su capacidad de recibir mensajes de *downlink* [22]:

- **Clase A**: todos los terminales LoRaWAN implementan la clase A. Cada uno de los terminales puede transmitir (*uplink*) al *gateway* sin atender a la disponibilidad del *gateway*. Sólo permiten la comunicación descendente en 2 ventanas de recepción que se abren tras una transmisión. Estos tipos de terminales están pensados para la comunicación unidireccional de la mota a la red y son los que menos potencia consumen, por lo tanto, tienen una gran eficiencia energética. En la figura 3.5 se muestra el esquema para los terminales de clase A:

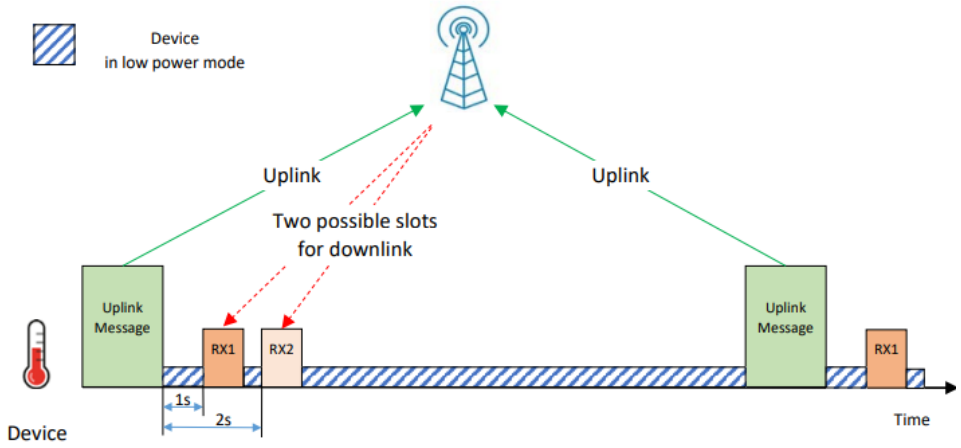

Figura 3.5: LoRaWAN clase A [22]

- **Clase B**: los terminales de esta clase aparte de recibir los mensajes en las ventanas tras una transmisión, reciben mensajes en ventanas programadas. Para que el terminal abra su ventana de recepción en el tiempo programado, el *gateway* debe mandarle un mensaje *beacon* de sincronización de tiempo. Esto hace que el servidor sepa cuando está esperando un mensaje *downlink*. En la figura 3.6 se muestra el esquema para los terminales de clase B:

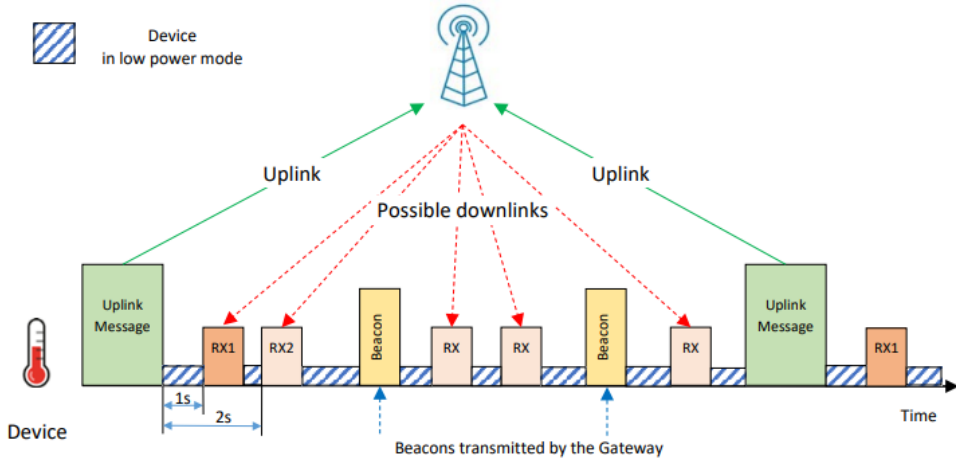

Figura 3.6: LoRaWAN clase B [22]

- **Clase C**: son los terminales que tienen menor latencia de *downlink*, pero menor duración de batería debido a su alto consumo. Esto se debe a que sus ventanas de recepción siempre están abiertas, salvo durante las transmisiones. En la figura 3.7 se muestra el esquema para los terminales de clase C:

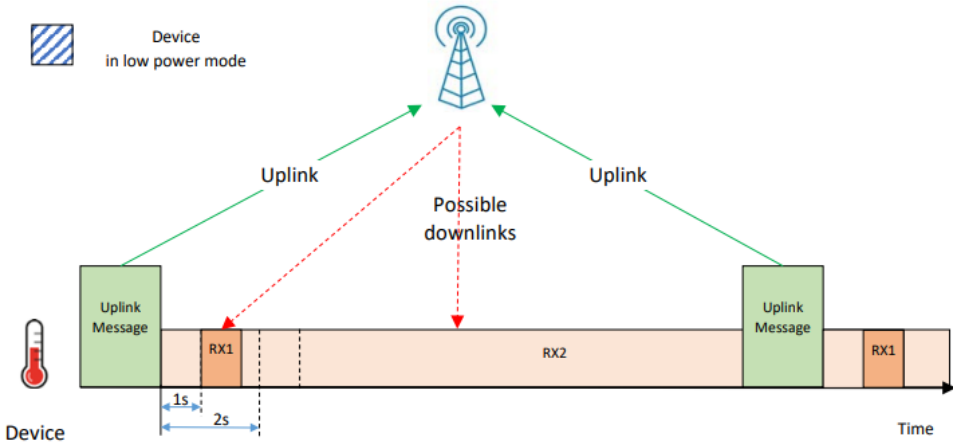

Figura 3.7: LoRaWAN clase C [22]

3.1.4. Message Queing Telemetry Transport

El protocolo MQTT ha adquirido una gran importancia en el mundo del IoT gracias a su sencillez y ligereza, características esenciales en dispositivos de IoT que se caracterizan por tener limitaciones de potencia, consumo y ancho de banda. MQTT es un protocolo de comunicación M2M (*machine-to-machine*) que se ejecuta sobre TCP/IP, basado en el método publicación/suscripción. Existen 2 tipos de sistemas que son los clientes y los brókers, donde los primeros necesitan conectar con el bróker, que es el responsable de recibir comunicaciones y enviarlas. La conexión se mantiene activa hasta que el cliente la finalice, empleando MQTT en el puerto 1883 y 8883 cuando va sobre TLS [23].

Los tipos de mensaje que se envían son: *CONNECT*, *CONNACK*, *PUBLISH*, *PUBACK*, *PUBREC*, *PUBREL*, *PUBCOMP*, *SUBSCRIBE*, *UNSUBSCRIBE*, *SUBACK*, *UNSUBACK*, *PINGREQ*, *PINGRESP* y *DISCONNECT*.

Existen 3 niveles de calidad de servicio que permiten elegir entre minimizar la transmisión de datos y maximizar la fiabilidad [24]:

- **QoS 0**: ofrece la mínima cantidad de transmisión de datos, donde cada mensaje se envía una vez al suscriptor sin saber si llegó al destinatario.
- **QoS 1**: el bróker trata de entregar el mensaje y espera una confirmación del suscriptor. Si no se confirma el mensaje en un periodo de tiempo, este vuelve a ser enviado, garantizando que se entregue al menos una vez.
- **QoS 2**: el cliente y el bróker emplean una comunicación *4-way handshake* para garantizar que se recibe el mensaje una vez. En la figura 3.8 se muestra un ejemplo:

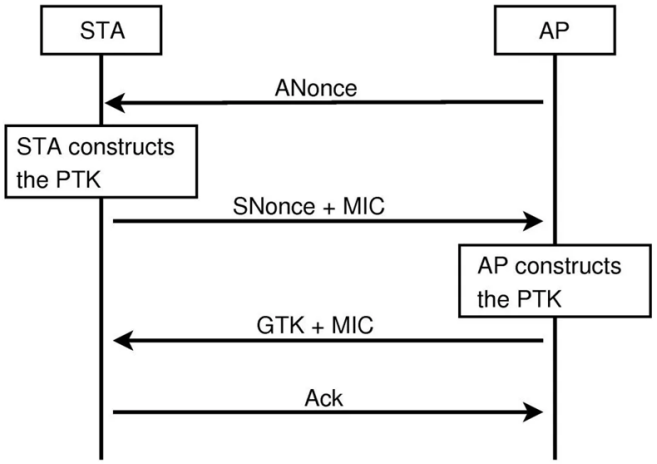

Figura 3.8: Ejemplo comunicación *4-way handshake* [25]

3.2. Orquestación de contenedores y Kubernetes

Un orquestador es un sistema que despliega y administra aplicaciones, permitiendo responder de manera dinámica a los cambios que ocurran dentro de éstas. Kubernetes es una herramienta importante de orquestación para automatizar las distintas tareas y, a continuación, se va a detallar su arquitectura y principales componentes con el fin de gestionar y operar en un clúster [26].

3.2.1. Arquitectura de Kubernetes

Atendiendo a lo explicado en el capítulo anterior, la arquitectura de un clúster de Kubernetes (figura 3.9) se compone de un conjunto de máquinas llamadas nodos, donde se despliegan las aplicaciones en contenedores.

Se diferencian dos tipos de nodos, nodo maestro (*Master Node*) y nodo de trabajo (*Worker Node*). El nodo maestro trabaja en el plano de control y se compone de [27]:

- **Kube-apiserver**: el servidor de la API permite exponer la API de Kubernetes, recibiendo peticiones y actualizándolas según su estado en etcd.
- **Etcd**: almacena los valores de clave consistente y de alta disponibilidad para todos los datos del servidor API.
- **Kube-scheduler**: busca pods que aún no estén vinculados a un nodo y les asigna uno, teniendo en cuenta una serie de factores.
- **Kube-controller-manager**: ejecuta los controladores (nodo, replicación, *endpoint*, *tokens* y cuentas de servicio) de Kubernetes. También puede ser integrado por el proveedor de *cloud*, denominándose *cloud-controller-manager*.

Por su parte, en los nodos de trabajo se distinguen los siguientes componentes [27]:

- **Kubelet**: agente que se ejecuta en cada nodo y se asegura de que los pods estén funcionando, así como sus contenedores.
- **Kube-proxy**: mantiene reglas de red en los nodos para implementar servicios.
- **Container Runtime**: software responsable de ejecutar los contenedores.

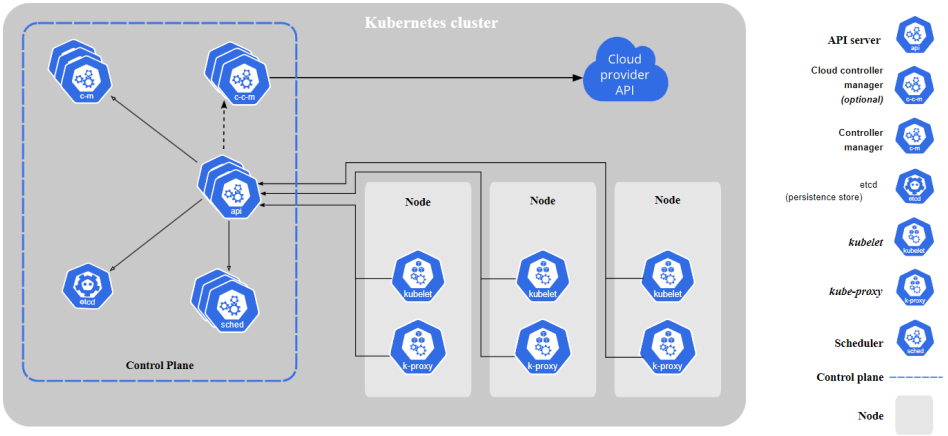

Figura 3.9: Clúster de Kubernetes [27]

3.2.2. Elementos de Kubernetes

Existen multitud de elementos con el fin de gestionar aplicaciones en el orquestador de Kubernetes. A continuación, se van a detallar los más comunes y utilizados en el proyecto.

Los elementos de Kubernetes se pueden clasificar dependiendo de la función que realicen, tal y como se presenta a continuación [27].

Cargas de trabajo

Una carga de trabajo no es otra cosa que una aplicación que se ejecuta en Kubernetes, dentro de pods que actúan en conjunto.

- **Pods**: son las unidades más pequeñas que se pueden crear y desplegar en Kubernetes.
- **ReplicaSets**: mantienen un conjunto estable de réplicas de pods en ejecución en todo momento, con el fin de garantizar la disponibilidad de un número de pods idénticos.
- **Deployment**: proporciona actualizaciones declarativas para los pods y los replicasets. Engloba en su manifiesto tanto los pods como sus réplicas.

Servicios, balanceo de carga y redes

En Kubernetes no es necesario modificar la aplicación para hacer uso de algún mecanismo que descubra servicios desconocidos, ya que proporciona a sus pods su propia IP y DNS, pudiendo balancear la carga entre ellos.

- **Service**: es el objeto de la API de Kubernetes que describe puertos y balanceadores de carga para un conjunto de pods. Su creación viene motivada por la problemática que surje a la hora de hacer uso de un deployment, el cual puede correr un conjunto de pods que puede ser diferente al conjunto de pods que corra la aplicación en otro momento posterior, ya que los pods se crean y destruyen dinámicamente. En la figura 3.10 se muestra el esquema del funcionamiento del elemento service:

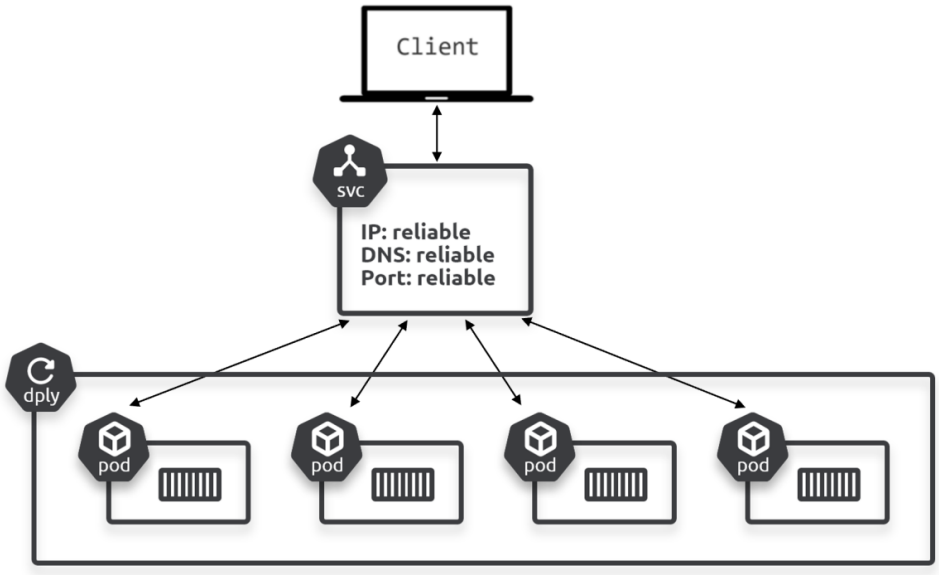

Figura 3.10: Esquema del elemento *service* [26]

Almacenamiento

- **Volume**: permite que se guarde el estado de los pods ante una falla, pues los archivos en disco de un contenedor son temporales y se pierden, ya que Kubelet reinicia y limpia el contenedor. También permite que se produzca un almacenamiento compartido entre contenedores. Por su parte, si requerimos que no se borren datos al dejar de existir un pod, se hace uso de los *persistent volume* (PV) y los *persistent volume claim* (PVC), donde este último da la posibilidad al usuario de solicitar el almacenamiento que desee. En la figura 3.11 puede observarse un esquema del elemento volume:

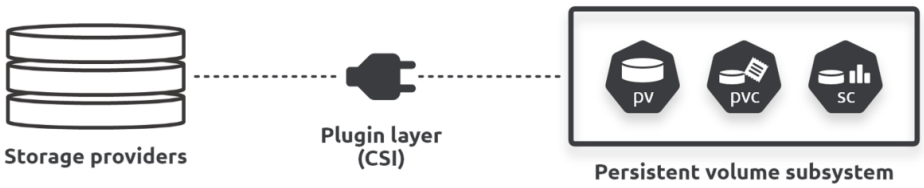

Figura 3.11: Esquema del elemento *volume* [26]

Configuración

- **ConfigMap**: es un objeto de la API que se usa para almacenar datos de formato clave-valor, que los pods pueden usar como variables de entorno, como argumentos de la línea de comandos o como fichero de configuración en un volumen. Si se quisiera almacenar datos encriptados, se haría uso de un *secret*, pues configmap no proporciona encriptación. En la figura 3.12 se puede observar el esquema de configmap.

  Lo que permite configmap es crear una configuración separada del código de la aplicación, de esta forma, puede estar corriendo la aplicación en tráfico real y, a su vez, disponer del código localmente en un entorno de desarrollo.

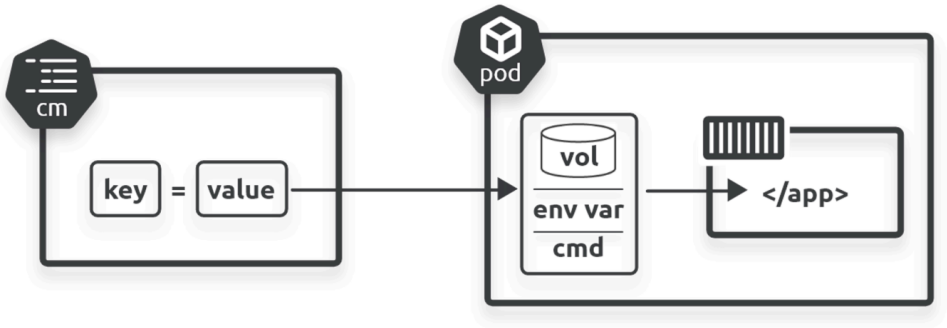

Figura 3.12: Esquema del elemento *configmap* [26]

Capítulo 4

Arquitectura e implementación del sistema

En este capítulo, se va a explicar cómo se ha realizado el diseño, así como la implementación de Kubernetes para llevar a cabo el despliegue de la red LoRaWAN.

4.1. Arquitectura general del sistema de despliegue automatizado

El proyecto tiene como propósito crear una red en la que dispositivos LoRaWAN puedan enviar mensajes a sus servidores de red y aplicación, donde se desplegarán dichos servicios mediante un clúster de Kubernetes. En la figura 4.1 se muestra el diagrama de la arquitectura de red, en el que se van a abordar las funciones que tiene cada elemento.

- **Mota**: envía mensajes LoRa cada cierto tiempo, simulando ser un dispositivo que capta información, como puede ser un sensor.
- **Gateway**: es el responsable de recibir los mensajes vía LoRa transmitidos por la mota y reenviarlos al servidor LoRaWAN. El dispositivo empleado es el *gateway* Laird Sentrius RG1xx.
- **Clúster**: su función es crear y administrar los pods donde están alojados los diferentes elementos de la red de depliegue, la cual expone también las direcciones IP de los servicios para acceder a ellos. Se ha utilizado K8s (Kubernetes).
- **Servidor LoRaWAN**: en el proyecto se ha empleado la plataforma ChirpStack, que permite una monitorización completa. Se reciben los mensajes en el puerto UDP 31700, el cual corresponde al servidor de red y los reenvía al servidor de aplicación. Se emplea el puerto UDP

31700 debido a que usamos *Destination Network Address Translation* (DNAT), que permite redirigir el tráfico entrante en el puerto 31700 hacia el contenedor que escucha en el puerto 1700.

Para permitir la comunicación entre el *gateway*, el PC local y la máquina virtual, se les ha asignado las IPs internas 192.168.111.1, 192.168.111.2 y 192.168.111.3 respectivamente.

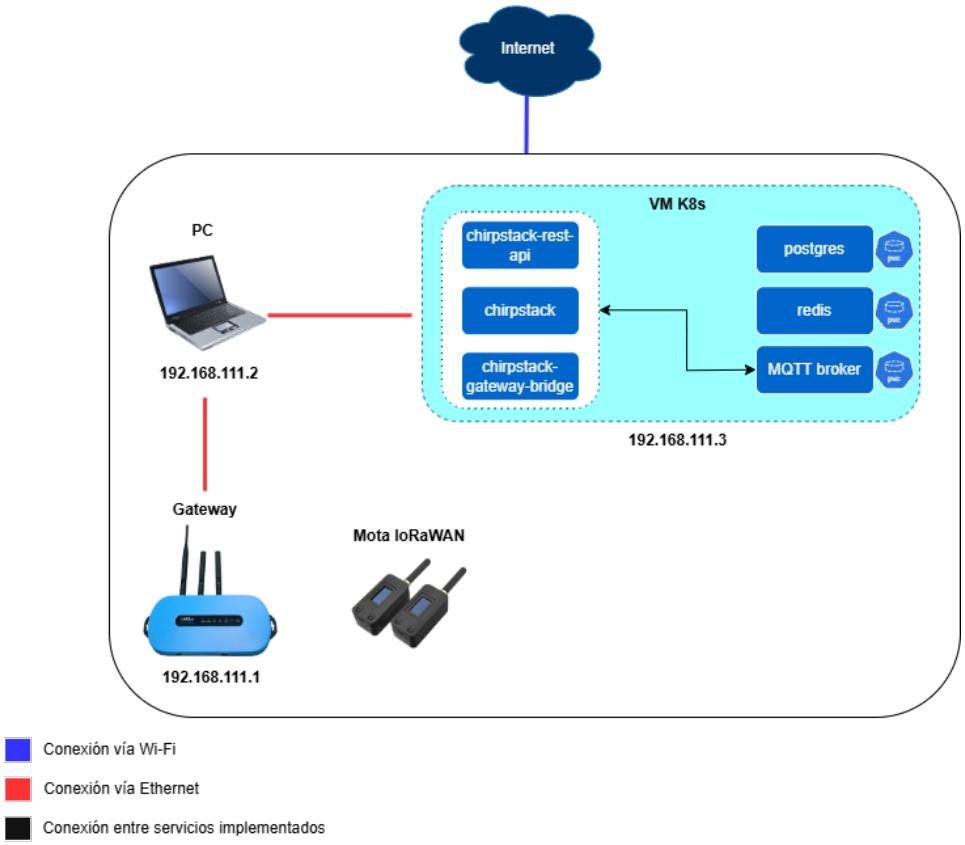

Figura 4.1: Diseño de la arquitectura de red [28]

4.2. Configuración del gateway

A la hora de configurar el *gateway* Laird, en primer lugar se le asigna la dirección IP estática 192.168.1.38 en la red del router doméstico, de tal forma que siempre se pueda acceder al mismo vía Wi-Fi. Seguidamente, tal y como se muestra en la figura 4.2, accedemos al apartado Wi-Fi de la interfaz del *gateway*, donde conectamos con la red doméstica.

En el caso de estar fuera del alcance de nuestra red doméstica, haremos *tethering* mediante el teléfono móvil, de tal forma que el *gateway* conecte de igual forma a la red doméstica simulada por el móvil.

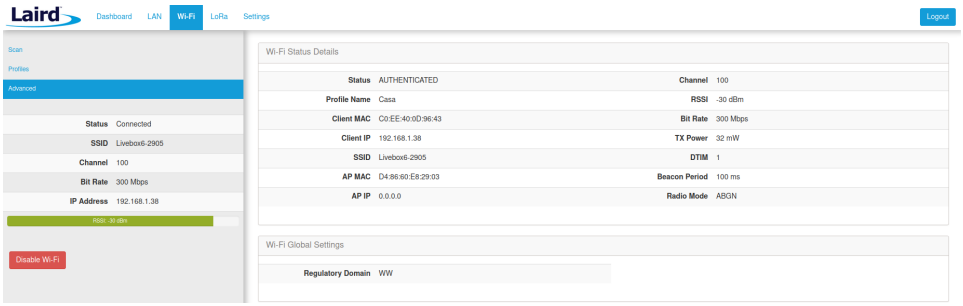

Figura 4.2: Configuración Wi-Fi del *gateway*

Como se puede observar en la figura 4.3, se prosigue con la configuración LAN del *gateway*, en la cual se asigna la dirección IP 192.168.111.1 dentro de una nueva subred para poder comunicar con el resto de equipos vía *ethernet*.

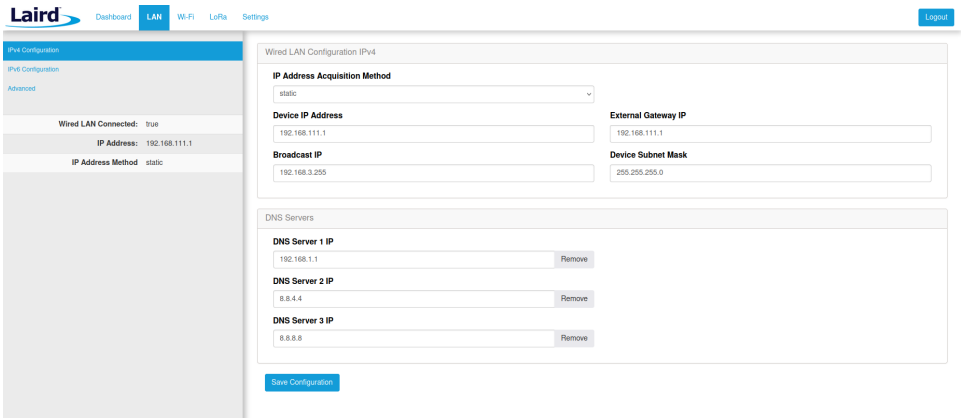

Figura 4.3: Configuración LAN del *gateway*

Por su parte, tal y como se muestra en la figura 4.4, en el apartado de configuración LoRa del *gateway*, se hace uso de la opción de *forwarder* hacia la dirección IP 192.168.111.3, perteneciente a la máquina virtual donde está contenido el clúster de Kubernetes, a la que se redirige el tráfico entrante a través del puerto UDP 31700 hacia el puerto 1700 en el que escucha el contenedor, haciendo uso de DNAT.

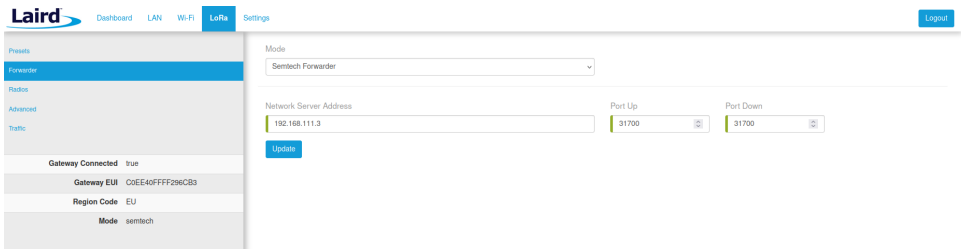

Figura 4.4: Configuración LoRaWAN del *gateway*

Por último, en la figura 4.5 se muestra el *dashboard* del *gateway*, que permite visualizar las conexiones configuradas. Además de ello, se puede observar el modelo de *gateway* con su versión de *firmware*, así como un apartado de ajustes donde, por ejemplo, podemos cambiar las credenciales por defecto de acceso al equipo.

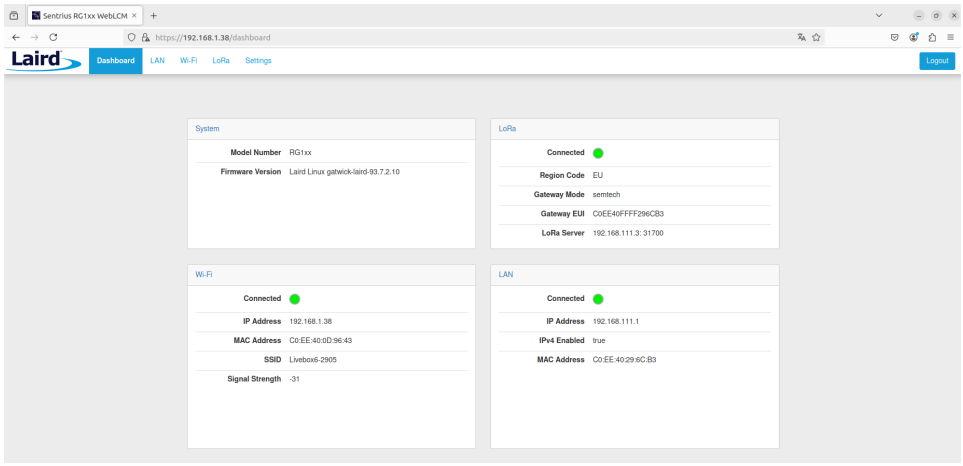

Figura 4.5: *Dashboard* del *gateway*

4.3. Integración con Kubernetes

El clúster de Kubernetes se encarga del correcto funcionamiento de la red, de su disponibilidad y del despliegue de los servicios. El clúster se encuentra alojado en una máquina virtual con sistema operativo Ubuntu 22.04, creada en la herramienta Oracle VM VirtualBox. A continuación, se describen los componentes que forman el clúster de Kubernetes, donde se van a explicar en detalle las funciones que realiza cada uno de ellos.

4.3.1. Redis

Dentro de la arquitectura, Redis se implementa como base de datos en memoria y caché, optimizando el rendimiento ya que reduce la carga sobre PostgreSQL y acelera las operaciones de lectura.

Redis se compone de un archivo de despliegue, que contiene la imagen y versión asegurando compatibilidad, una estrategia de actualización para que se reinicie por completo el pod durante una actualización y monta un volumen persistente para garantizar que los datos no se pierdan ante un reinicio. También se compone de un archivo de servicio, donde expone el puerto 6379 con *ClusterIP* para conexiones internas en el clúster. Por último, se compone de un archivo de almacenamiento, permitiendo la persistencia de datos de Redis, que solicita una capacidad de 100Mi (mebibytes).

La configuración detallada de Redis, incluyendo *Deployment*, *Service* y *PersistentVolumeClaim*, se encuentra en el anexo B.1.1.

4.3.2. PostgreSQL

PostgreSQL actúa como la base de datos del sistema, almacenando toda la información de ChirpStack.

En su archivo de despliegue contiene la imagen, sus credenciales, un volumen montado y vinculado al *PersistentVolumeClaim*, además de unos *scripts* SQL para crear usuarios y añadir extensiones. En su archivo de servicio, expone el puerto 5432 con *ClusterIP* para tener acceso interno en el clúster. Consta de un archivo de almacenamiento con una capacidad de 100Mi (mebibytes) y un archivo configmap que contiene *scripts* Bash que utiliza el archivo de despliegue, como se comentó antes.

La configuración detallada de PostgreSQL, incluyendo *Deployment*, *Service*, *PersistentVolumeClaim* y *ConfigMap*, se encuentra en el anexo B.1.2.

4.3.3. Mosquitto

Mosquitto actúa como bróker MQTT, proporcionando comunicación bidireccional entre el dispositivo terminal y el servidor ChirpStack.

En su archivo de despliegue, aparte de contener su imagen, contiene un *script* que copia archivos de configuración al volumen persistente. En su archivo de servicio, expone en el puerto estándar 1883 con LoadBalancer para proporcionar acceso externo del clúster. Tiene un archivo de almacenamiento con capacidad de 100Mi (mebibytes) que almacena la configuración mosquitto, otra de seguridad dinámica y un archivo de contraseñas. Por último, consta de un archivo configmap que contiene elementos para deshabilitar conexiones anónimas y usar autenticación por archivos de contraseñas, para almacenar el *hash* del usuario MQTTUSER, para definir roles y ACLs, y

el *script* comentado anteriormente para hacer que persistan los archivos de configuración ante reinicios.

La configuración detallada de Mosquitto, incluyendo *Deployment*, *Service*, *PersistentVolumeClaim* y *ConfigMap*, se encuentra en el anexo B.1.3.

4.3.4. ChirpStack Gateway

Es un componente esencial que conecta el *gateway* físico con la red ChirpStack, convirtiendo el protocolo UDP de los *gateways* a MQTT/JSON para ChirpStack, gestionando las comunicaciones tanto de *uplink* como de *downlink* y configurando para la banda EU868 (frecuencia europea de 868MHz).

En su archivo de despliegue contiene su imagen, *templates* para *topics* MQTT para tener una estructura organizada por *gateway* ID y tipo de mensaje, el puerto estándar para *Semtech Packet Forwarder* 1700 UDP y un archivo de configuración montado mediante configmap. En su archivo de servicio, expone el puerto 1700 UDP mapeado con *NodePort* 31700, accesible para el *gateway* externo y del tipo LoadBalancer con IP externa fija 10.0.2.202. Finalmente, consta de un configmap que configura la conexión MQTT mediante el bróker y las credenciales.

La configuración detallada de ChirpStack Gateway, incluyendo *Deployment*, *Service* y *ConfigMap*, se encuentra en el anexo B.1.4.

4.3.5. Servidor de ChirpStack

Se trata del componente principal, pues es el gestor de la red completa LoRaWAN, que se encarga de coordinar el *gateway* y el terminal, de procesar los mensajes de *uplink* y downlink, así como de implementar el protocolo LoRaWAN.

En lo que respecta a su archivo de despliegue, además de contener su imagen, consta de variables de entorno con mosquitto para conectar al bróker MQTT y conectar con el *gateway*, con postgres para conectar con el servidor de base de datos y con redis para conectar con su servidor caché. En su archivo de servicio, expone el puerto 8080 para conexiones gRPC/HTTP. La configuración se carga mediante un configmap que contiene todos los parámetros de red y seguridad.

La configuración detallada de ChirpStack, incluyendo *Deployment*, *Service* y *ConfigMap*, se encuentra en el anexo B.1.5.

4.3.6. ChirpStack Rest API

Este componente proporciona la interfaz de interacción con la plataforma ChirpStack, permitiendo gestionar centralizadamente el terminal, *gateway* y

usuarios mediante API REST. También permite la integración con aplicaciones externas.

Consta de un archivo de despliegue que porta la imagen y una serie de argumentos para realizar la conexión al servidor ChirpStack, para escuchar en todas las interfaces y permitir conexiones HTTP. Además costa de un archivo de servicio, que expone internamente el puerto 8090 y de tipo *ClusterIP*.

La configuración detallada de ChirpStack Rest API, incluyendo *Deployment* y *Service*, se encuentra en el anexo B.1.6.

4.4. Automatización

A propósito de automatizar el despliegue y la eliminación de las distintas aplicaciones que conforman el clúster de Kubernetes, se emplean los siguientes *scripts*.

En el *script deploy.sh*, se realiza una configuración inicial donde se indica que el *script* se ejecuta con *Bash*, la posibilidad de ejecutar el *script* desde otro directorio y un cambio de directorio donde se encuentra el *script* para asegurar las rutas de los YAML.

Avanzando en el código, se realiza la eliminación de *configmaps* anteriores que pudiesen entrar en conflicto con nuevas configuraciones. Una vez hecho, primero se despliegan los *configmaps* aplicando los archivos YAML correspondientes mediante el comando *microk8s kubectl apply -f configmaps/<nombre_del_archivo.yaml>*. Estos archivos definen configuraciones para lo que resta de despliegue.

Se prosigue con la creación de los volúmenes persistentes, ya que los *deployments* los usarán. Luego, se hace lo mismo para los servicios.

Por último, el *script* despliega los *deployments* en 2 partes, una primera que despliega servicios de infraestructura y, tras un intervalo de 5 segundos, se despliegan las aplicaciones de ChirpStack.

```
#!/bin/bash

SCRIPT=$(readlink -f $0)
SCRIPTPATH=`dirname $SCRIPT`
cd $SCRIPTPATH

# Delete previous configuration (configmaps)
microk8s kubectl delete configmap config-mosquitto 2> /dev/null
microk8s kubectl delete configmap config-postgres 2> /dev/null
microk8s kubectl delete configmap config-chirpstack 2> /dev/null
microk8s kubectl delete configmap config-chirpstack-gateway-bridge 2> /dev/null
```

```
# Create new configuration (configmaps)
microk8s kubectl apply -f configmaps/config-mosquitto.yaml
microk8s kubectl apply -f configmaps/config-postgres.yaml
microk8s kubectl apply -f configmaps/config-chirpstack.yaml
microk8s kubectl apply -f configmaps/config-chirpstack-gateway-bridge.yaml

# Create volumes
microk8s kubectl apply -f volumes/postgresqldata-persistentvolumeclaim.yaml
microk8s kubectl apply -f volumes/redisdata-persistentvolumeclaim.yaml
microk8s kubectl apply -f volumes/mosquittodata-persistentvolumeclaim.yaml

# Create services
microk8s kubectl apply -f services/mosquitto-service.yaml
microk8s kubectl apply -f services/postgres-service.yaml
microk8s kubectl apply -f services/redis-service.yaml
microk8s kubectl apply -f services/chirpstack-service.yaml
microk8s kubectl apply -f services/chirpstack-rest-api-service.yaml
microk8s kubectl apply -f services/chirpstack-gateway-bridge-service.yaml

# Create deployments
microk8s kubectl apply -f deployments/mosquitto-deployment.yaml
microk8s kubectl apply -f deployments/postgres-deployment.yaml
microk8s kubectl apply -f deployments/redis-deployment.yaml

sleep 5

microk8s kubectl apply -f deployments/chirpstack-deployment.yaml
microk8s kubectl apply -f deployments/chirpstack-rest-api-deployment.yaml
microk8s kubectl apply -f deployments/chirpstack-gateway-bridge-deployment.yaml
```

Por su parte, en el *script undeploy.sh* se eliminan los recursos del clúster de Kubernetes, manteniendo un orden seguro. Al igual que se hizo con *deploy.sh*, se realiza una configuración inicial.

Luego, los *deployments* son los primeros en ser eliminados mediante el comando *microk8s kubectl delete deployment <nombre_del_archivo>*. Siguiendo el orden contrario al realizado anteriormente, se prosigue con la eliminación de los servicios, luego de los *configmaps* y, por último, de los volúmenes persistentes.

```
#!/bin/bash

SCRIPT=$(readlink -f $0)
SCRIPTPATH=`dirname $SCRIPT`
cd $SCRIPTPATH
```

```
# Delete deployments
microk8s kubectl delete deployment mosquitto
microk8s kubectl delete deployment chirpstack
microk8s kubectl delete deployment postgres
microk8s kubectl delete deployment redis
microk8s kubectl delete deployment chirpstack-rest-api
microk8s kubectl delete deployment chirpstack-gateway-bridge

# Delete services
microk8s kubectl delete service mosquitto
microk8s kubectl delete service postgres
microk8s kubectl delete service redis
microk8s kubectl delete service chirpstack
microk8s kubectl delete service chirpstack-rest-api
microk8s kubectl delete service chirpstack-gateway-bridge

# Delete configmaps
microk8s kubectl delete configmaps config-mosquitto
microk8s kubectl delete configmaps config-postgres
microk8s kubectl delete configmaps config-chirpstack
microk8s kubectl delete configmaps config-chirpstack-gateway-bridge

# Delete persistent volume claims
microk8s kubectl delete pvc redisdata
microk8s kubectl delete pvc postgresqldata
microk8s kubectl delete pvc mosquittodata
```

4.5. Configuración de ChirpStack

En cuanto a la configuración del servidor LoRaWAN, se emplea ChirpStack, el cual se crea y despliega desde dentro del clúster contenido en la máquina virtual Ubuntu 22.04. Tal y como se explicó en el apartado 4.3.5, el servidor se expone mediante *LoadBalancer* a la IP 192.168.1.51 por el puerto TCP 8080.

A continuación se explican los pasos a seguir para configurar el servidor ChirpStack:

1. **Instalación y configuración de ChirpStack**: tal y como se ha comentado anteriormente, mediante Kubernetes se instala y configura ChirpStack junto a su bróker MQTT. Tras desplegar el servicio, se accede a la interfaz de la plataforma a través de la dirección web

http://192.168.1.51:8080 introduciendo el usuario *admin* y la contraseña *admin*.

2. **Creación de una organización**: en la interfaz de la plataforma existe el apartado *tenant* donde se incluyen los *gateways* y aplicaciones.

3. **Creación del gateway**: dentro de la organización se accede al apartado de *gateways* y se añade la pasarela que forma parte de la red. Como se muestra en la figura 4.6, se añade tanto el nombre como su identificador de dispositivo.

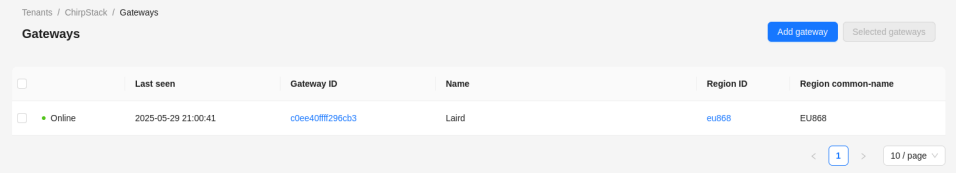

Figura 4.6: Creación del *gateway* en ChirpStack

4. **Creación de perfil de dispositivo**: en el apartado *Devices Profiles* se crea el perfil del dispositivo. En él se añade el nombre, la región y frecuencia en la que se encuentra (en este caso EU868), la MAC del dispositivo, algoritmo ADR (en este caso por defecto) y el intervalo de tiempo esperado de subida en segundos. La activación de la mota se hace mediante ABP, así que se desactiva la opción mediante OTAA (figura 4.7).

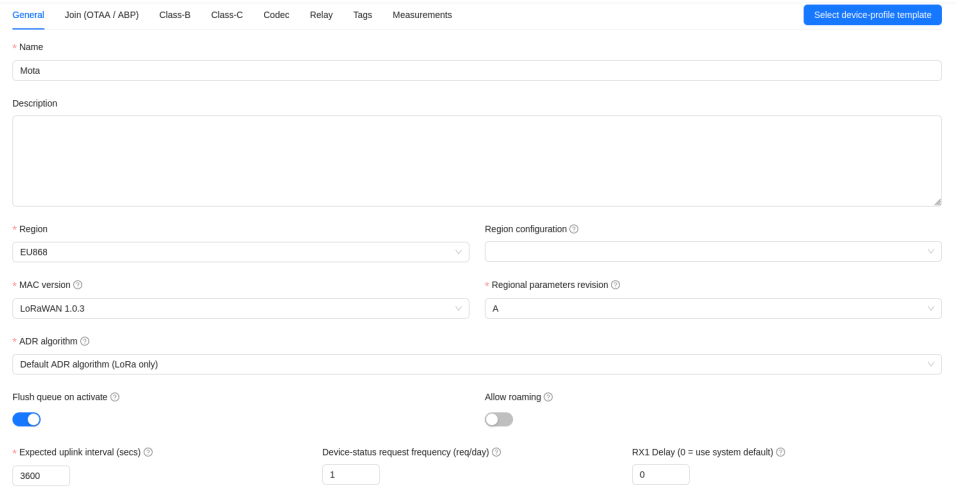

Figura 4.7: Creación del perfil de dispositivo en ChirpStack

5. **Creación de aplicación**: una vez creado el perfil del dispositivo, se crea la aplicación en el apartado *Applications* con el nombre que se desee. Ya dentro de la aplicación, se añade el equipo indicando su nombre, el *device EUI64*, el *join EUI64* y se elige el perfil del dispositivo (figura 4.8).

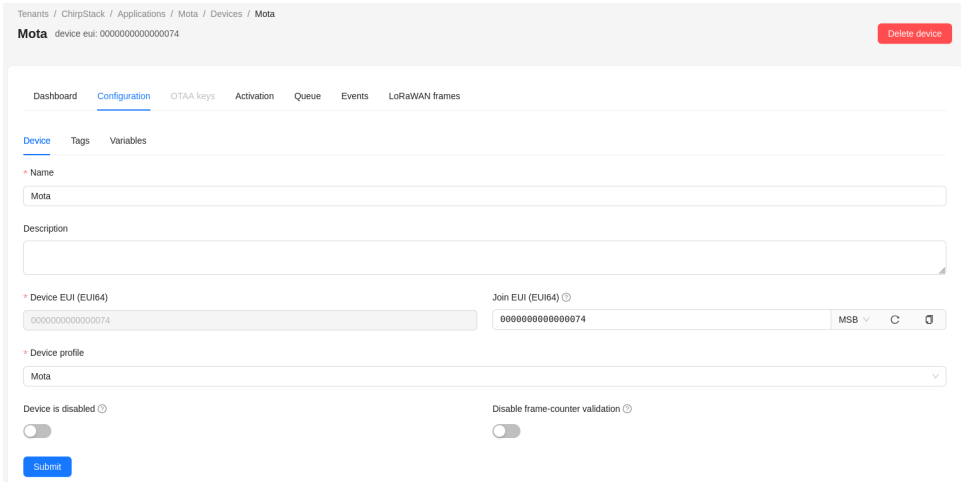

Figura 4.8: Creación de la aplicación en ChirpStack

Por otro lado, en el apartado de *Activation* del dispositivo añadido (figura 4.9), se indica su dirección obtenida mediante el uso de la herramienta PuTTY (figura 4.10), así como su *NwkSKey* y su *AppSKey* incluidos en el *firmware* de la mota.

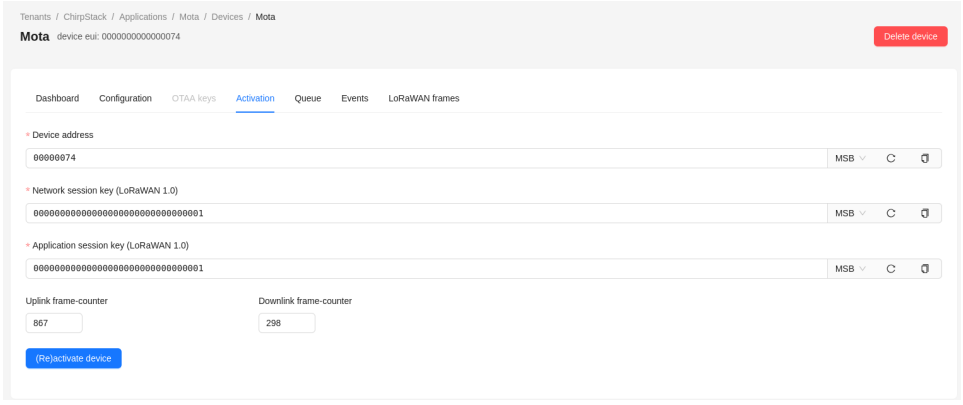

Figura 4.9: Activación de la aplicación en ChirpStack

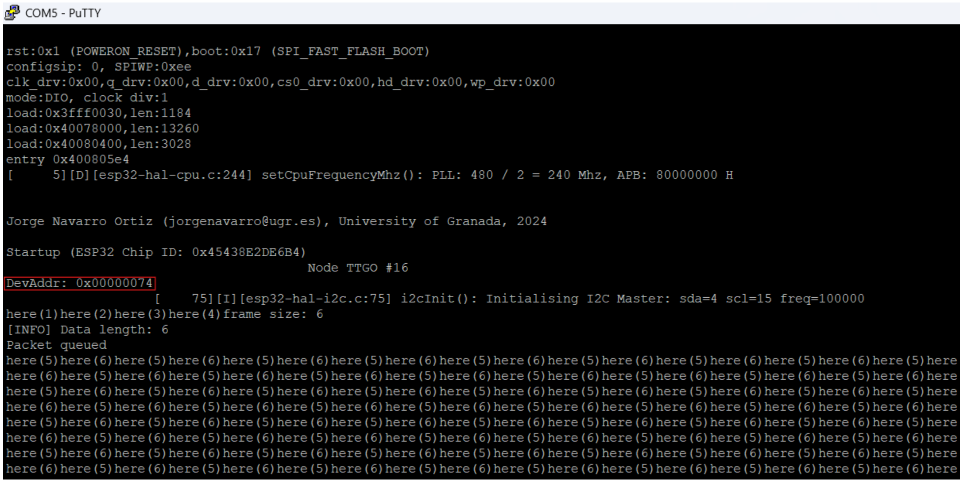

Figura 4.10: Acceso al puerto serial mediante PuTTY

Una vez realizados los pasos anteriores, en la figura 4.11 se comprueba si tanto el dispositivo como el *gateway* están activos en la red, así como los mensajes detallados en las figuras 4.12, 4.13 y 4.14.

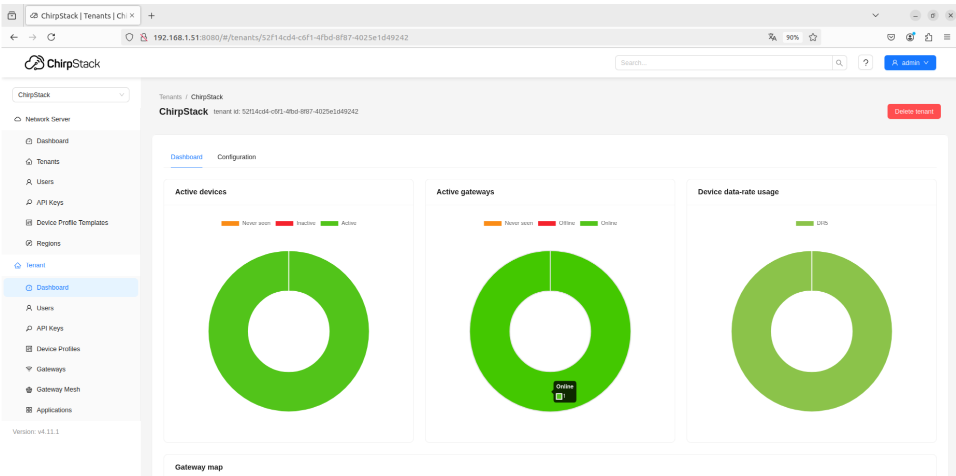

Figura 4.11: Comprobación del *dashboard* ChirpStack

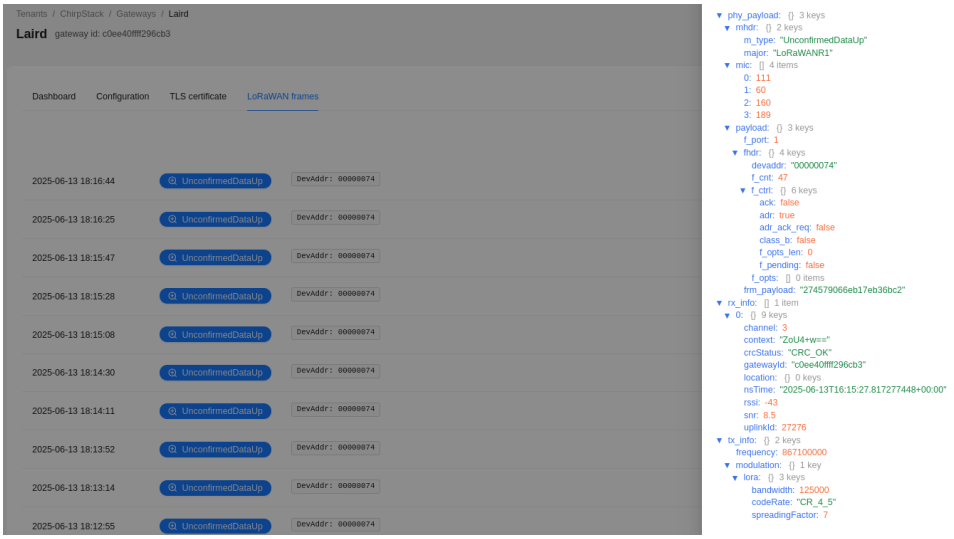

Figura 4.12: Mensajes recibidos por el *gateway* desde ChirpStack

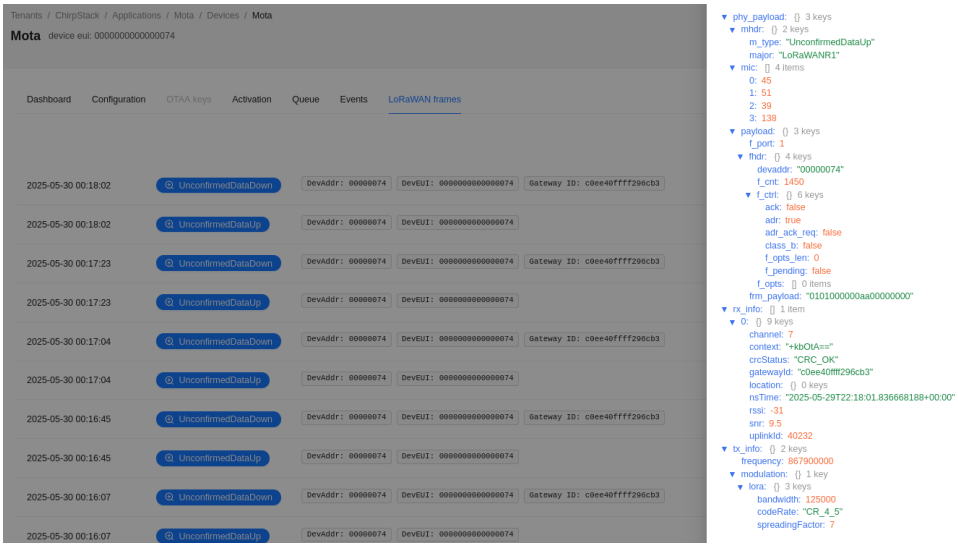

Figura 4.13: Mensajes recibidos por el servidor LoRaWAN desde ChirpStack

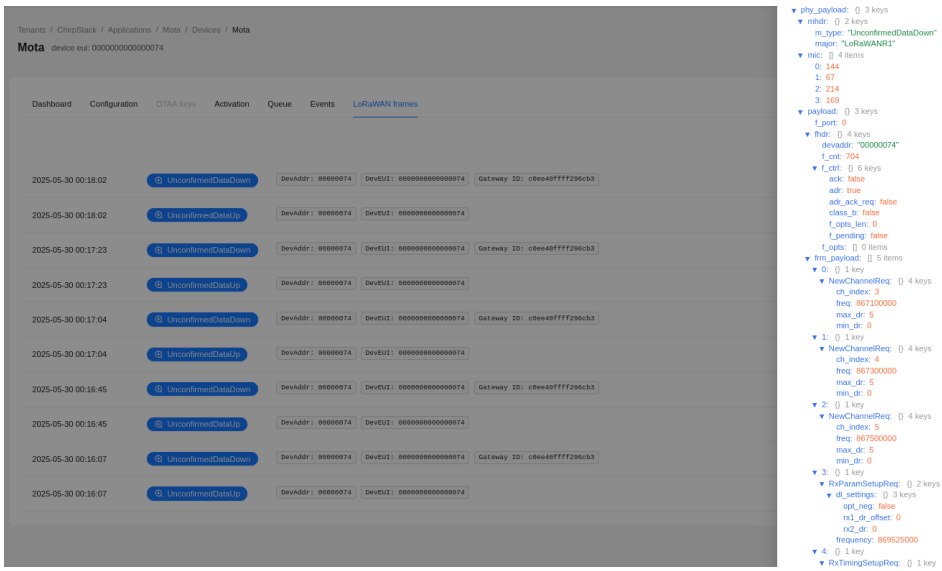

Figura 4.14: Mensajes enviados por el servidor LoRaWAN desde ChirpStack

Capítulo 5

Evaluación de rendimiento y conectividad

En este capítulo, se va a comentar el correcto funcionamiento del despliegue de red en el clúster Kubernetes, así como la evaluación de la conectividad de todos los nodos de red.

5.1. Rendimiento de los pods

Antes de abordar los registros de eventos de los diferentes pods que conforman la red, en la figura 5.1 se muestran los recursos de Kubernetes que están disponibles y su estado.

```
vboxuser@Ubuntu:~$
vboxuser@Ubuntu:~$ kubectl get all
NAME                                            READY   STATUS    RESTARTS        AGE
pod/chirpstack-6778f859d8-7bmfh                 1/1     Running   14 (2m40s ago)  52d
pod/chirpstack-gateway-bridge-76f5585bf9-9kkfk  1/1     Running   4 (3m49s ago)   52d
pod/chirpstack-rest-api-656764888-pktr8         1/1     Running   4 (3m49s ago)   52d
pod/mosquitto-b9657555-6nzns                    1/1     Running   4 (3m49s ago)   52d
pod/postgres-86c9ff4bcd-h9z47                   1/1     Running   4 (3m49s ago)   52d
pod/redis-5dd644467b-v2xpm                      1/1     Running   4 (3m49s ago)   52d

NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP              PORT(S)          AGE
service/chirpstack                  LoadBalancer   10.152.183.84    192.168.1.51             8080:32269/TCP   52d
service/chirpstack-gateway-bridge   LoadBalancer   10.152.183.55    192.168.1.52,10.0.2.202  1700:31700/UDP   52d
service/chirpstack-rest-api         ClusterIP      10.152.183.188   <none>                   8090/TCP         52d
service/kubernetes                  ClusterIP      10.152.183.1     <none>                   443/TCP          208d
service/mosquitto                   LoadBalancer   10.152.183.34    192.168.1.50             1883:30971/TCP   52d
service/postgres                    ClusterIP      10.152.183.202   <none>                   5432/TCP         52d
service/redis                       ClusterIP      10.152.183.250   <none>                   6379/TCP         52d

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/chirpstack                 1/1     1            1           52d
deployment.apps/chirpstack-gateway-bridge  1/1     1            1           52d
deployment.apps/chirpstack-rest-api        1/1     1            1           52d
deployment.apps/mosquitto                  1/1     1            1           52d
deployment.apps/postgres                   1/1     1            1           52d
deployment.apps/redis                      1/1     1            1           52d

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/chirpstack-6778f859d8                 1         1         1       52d
replicaset.apps/chirpstack-gateway-bridge-76f5585bf9  1         1         1       52d
replicaset.apps/chirpstack-rest-api-656764888         1         1         1       52d
replicaset.apps/mosquitto-b9657555                    1         1         1       52d
replicaset.apps/postgres-86c9ff4bcd                   1         1         1       52d
replicaset.apps/redis-5dd644467b                      1         1         1       52d
vboxuser@Ubuntu:~$
```

Figura 5.1: Contenido de los pods

Una vez que todos los pods se encuentran en ejecución, podemos introducir el comando *kubectl logs <nombre_del_pod>* para ver la información que contiene cada uno de ellos.

En la figura 5.2 se muestran los logs del pod mosquitto, donde se puede observar cómo se inicia correctamente el servicio MQTT y se encuentra escuchando en el puerto 1883. También pueden observarse conexiones exitosas de clientes MQTT, que se autentican con el usuario MQTTUSER y se proporcionan otros parámetros de conexión como QoS o *keep-alive time*.

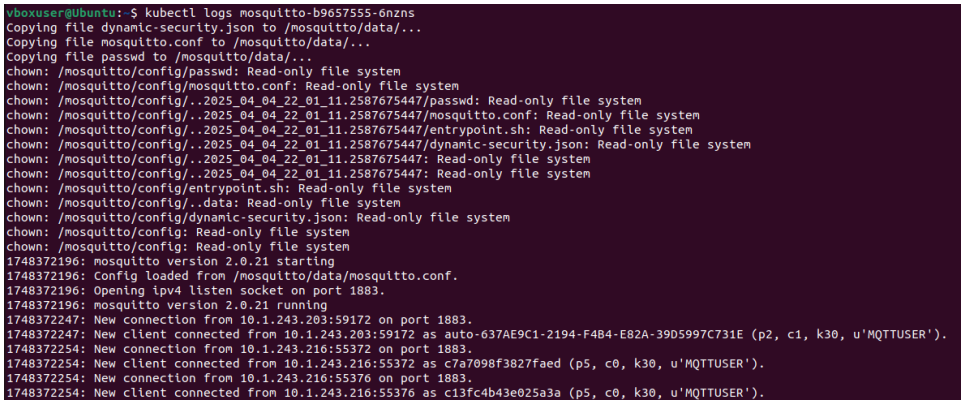

Figura 5.2: Logs pod mosquitto

En la figura 5.3 se muestran los logs del pod chirpstack-gateway-bridge, en el que se observa cómo ha conectado con el bróker MQTT y como la pasarela publica eventos de forma continua, luego funciona de manera correcta.

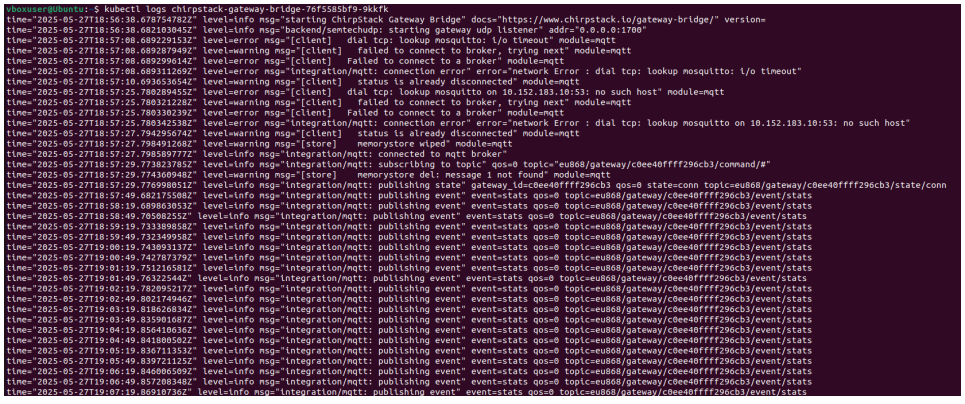

Figura 5.3: Logs pod chirpstack-gateway-bridge

En los logs del pod del servidor de chirpstack que se muestra en la figura

5.4, se puede observar cómo el servidor se inicializa de manera correcta, conectando con redis y postgresql, aplicando las migraciones correspondientes y conectando con el bróker MQTT. También se observa cómo se configura la región y los *backends*, se confirman suscripciones exitosas a MQTT y recibe datos desde el *gateway*.

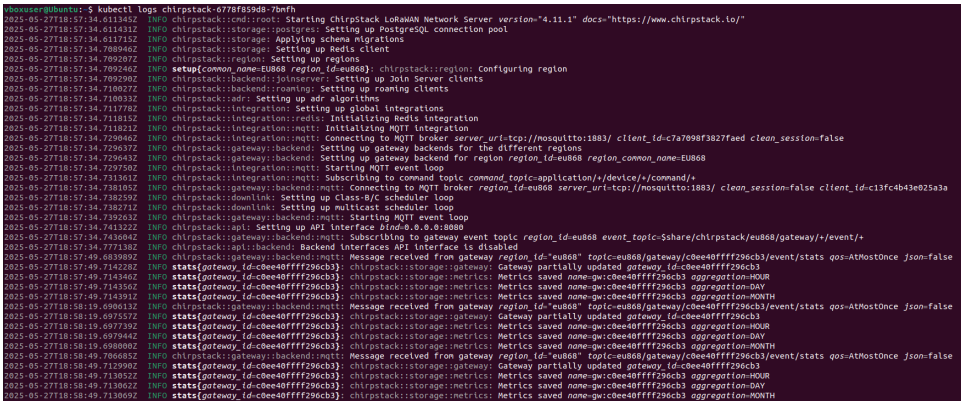

Figura 5.4: Logs pod servidor chirpstack

En lo que respecta a los logs del pod chirpstack-rest-api, en la figura 5.5 se muestra cómo el servicio REST API arranca correctamente.

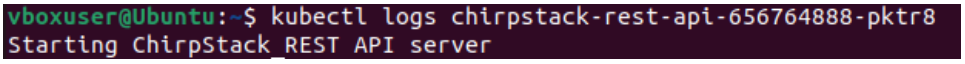

Figura 5.5: Logs pod chirpstack-rest-api

Como puede observarse en la figura 5.6, perteneciente a los logs de redis, el servidor redis muestra la versión y el puerto en el que inicia. Carga datos previos desde un archivo RDB y realiza guardados en segundo plano con éxito cada cierto tiempo, asegurando su persistencia. También mantiene y actualiza la base de datos en memoria, gestionando claves y cambios.

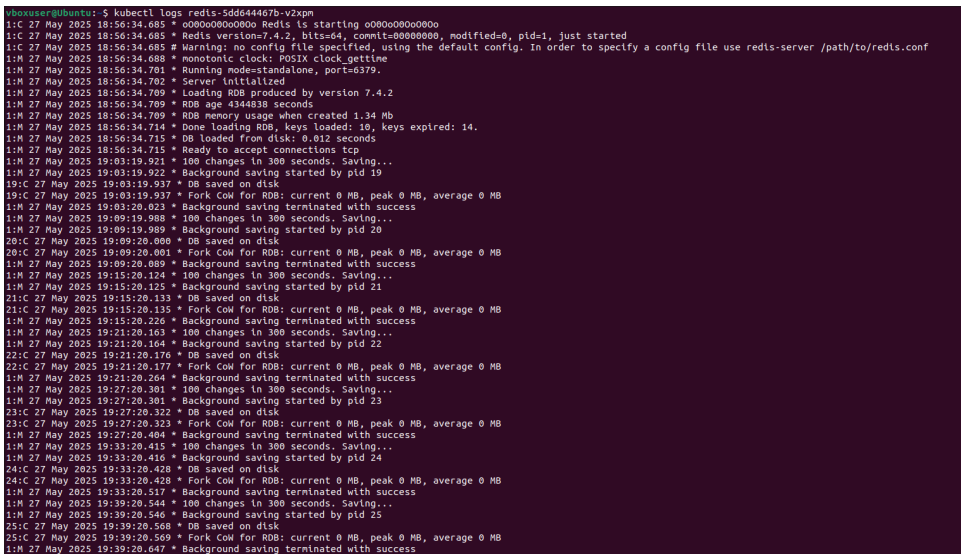

Figura 5.6: Logs pod redis

Por último, en la figura 5.7 se muestran los logs del pod postgresql, donde se observa que se inicializa la base de datos detectando ya una base de datos existente, lo que es normal en volúmenes persistentes. Además, se inicia correctamente el servicio postgresql que está listo para aceptar conexiones, que escucha en todas las interfaces de red y cuyo puerto es 5432.

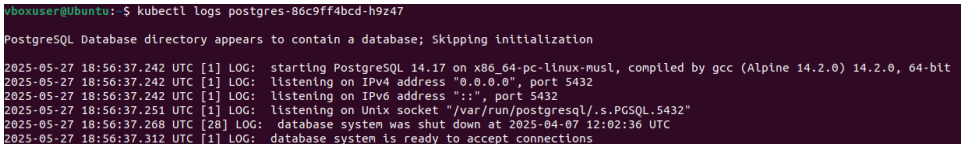

Figura 5.7: Logs pod postgresql

5.2. Conectividad entre dispositivos

Tal y como se comentó en el apartado 4.1, se ha creado una red entre el gateway, el PC local y la máquina virtual, asignándoles las IPs 192.168.111.1, 192.168.111.2 y 192.168.111.3 respectivamente. A continuación, se muestra en las figuras 5.8, 5.9 y 5.10, la correcta comunicación entre equipos de red mediante la herramienta *ping*.

```
vboxuser@Ubuntu:~$ ping 192.168.111.1
PING 192.168.111.1 (192.168.111.1) 56(84) bytes of data.
64 bytes from 192.168.111.1: icmp_seq=1 ttl=64 time=1.15 ms
64 bytes from 192.168.111.1: icmp_seq=2 ttl=64 time=1.24 ms
64 bytes from 192.168.111.1: icmp_seq=3 ttl=64 time=1.69 ms
64 bytes from 192.168.111.1: icmp_seq=4 ttl=64 time=1.51 ms
64 bytes from 192.168.111.1: icmp_seq=5 ttl=64 time=1.22 ms
64 bytes from 192.168.111.1: icmp_seq=6 ttl=64 time=1.17 ms
^C
--- 192.168.111.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5015ms
rtt min/avg/max/mdev = 1.146/1.327/1.685/0.199 ms
vboxuser@Ubuntu:~$
```

Figura 5.8: Comunicación entre clúster y *gateway*

```
vboxuser@Ubuntu:~$ ping 192.168.111.2
PING 192.168.111.2 (192.168.111.2) 56(84) bytes of data.
64 bytes from 192.168.111.2: icmp_seq=1 ttl=128 time=0.777 ms
64 bytes from 192.168.111.2: icmp_seq=2 ttl=128 time=0.473 ms
64 bytes from 192.168.111.2: icmp_seq=3 ttl=128 time=0.531 ms
64 bytes from 192.168.111.2: icmp_seq=4 ttl=128 time=0.885 ms
64 bytes from 192.168.111.2: icmp_seq=5 ttl=128 time=0.551 ms
64 bytes from 192.168.111.2: icmp_seq=6 ttl=128 time=0.439 ms
^C
--- 192.168.111.2 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5138ms
rtt min/avg/max/mdev = 0.439/0.609/0.885/0.163 ms
vboxuser@Ubuntu:~$
```

Figura 5.9: Comunicación entre clúster y PC cliente

```
C:\Users\USUARIO>ping 192.168.111.1

Haciendo ping a 192.168.111.1 con 32 bytes de datos:
Respuesta desde 192.168.111.1: bytes=32 tiempo<1m TTL=64
Respuesta desde 192.168.111.1: bytes=32 tiempo=1ms TTL=64
Respuesta desde 192.168.111.1: bytes=32 tiempo=1ms TTL=64
Respuesta desde 192.168.111.1: bytes=32 tiempo=1ms TTL=64

Estadísticas de ping para 192.168.111.1:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 0ms, Máximo = 1ms, Media = 0ms

C:\Users\USUARIO>
```

Figura 5.10: Comunicación entre PC cliente y *gateway*

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo, se presentan las conclusiones obtenidas en la realización del proyecto y se plantean propuestas de trabajo futuro.

6.1. Conclusiones

En este proyecto se ha realizado la automatización del despliegue de una red LoRaWAN mediante un orquestador de contenedores, que ha permitido gestionar de manera eficiente las aplicaciones y cuyo proceso se ha visto simplificado.

Primeramente, se estudiaron tanto las redes LoRaWAN como la orquestación de contenedores y se contemplaron varias alternativas para desplegar su red y automatizarla. Se eligieron las herramientas que mejor encajaban, tal y como se explica en el apartado 2.3.

El siguiente paso fue diseñar la arquitectura de la red para la integración de la red LoRaWAN mediante Kubernetes. Los dispositivos físicos que se requerían (mota y *gateway*), así como los componentes del clúster necesarios. Seguidamente, se realizaron las implementaciones y configuraciones correspondientes de cada elemento de red, tal y como se explica en el apartado 4.3.

Tras todo ello, se analizó el correcto funcionamiento de la red y se hicieron pruebas de conectividad entre los distintos nodos que la conforman, como se muestra en el capítulo 5.

6.2. Propuestas de trabajo futuro

Partiendo de la red diseñada y que funciona de manera correcta, se pueden proponer líneas de trabajo futuro como:

- Implementar copias de seguridad automatizadas, como pueden ser *backups* de bases de datos tanto para redis como para postgres y recuperación ante desastres (DRP).
- Monitorizar en tiempo real mediante el despliegue de InfluxDB y Grafana como pods en Kubernetes. InfluxDB para almacenar las métricas y Grafana como plataforma de visualización del estado de las motas, utilización de recursos y tráfico MQTT.
- Dotar a la red de un autoescalado según los recursos que se estén consumiendo y proporcionarle una alta disponibilidad.

De esta manera se prepara para un posible crecimiento de la red y se detectan de forma rápida los fallos con el empleo de métricas.

Apéndice A

Manual de usuario

En este apéndice se va a explicar cómo se utiliza el entorno del proyecto sobre orquestación de servicios LoRaWAN, para que sea utilizado en un futuro y a partir de él se hagan desarrollos y pruebas.

En primer lugar, hay que conectar todos los dispositivos que conforman la red, tanto la mota como el *gateway* se conectan al portátil vía USB y RJ-45 respectivamente. El *gateway* debe estar ya configurado correctamente en la red doméstica, tal y como se explica en el apartado 4.2. Una vez conectado, se accede a la máquina virtual donde está contenido el clúster Kubernetes y se arranca mediante el siguiente comando:

```
$ microk8s start
```

Una vez esté el clúster activo, debemos introducir los siguientes comandos para acceder al directorio donde se encuentra el *script deploy.sh* y desplegar los archivos YAML.

```
$ cd chirpstack
$ ./deploy.sh
```

En la figura A.1 se puede observar cómo se muestran todos los elementos que componen el clúster y se verifica que están corriendo los pods correctamente, mediante el comando:

```
$ kubectl get all
```

```
vboxuser@Ubuntu:~$
vboxuser@Ubuntu:~$ kubectl get all
NAME                                        READY   STATUS    RESTARTS        AGE
pod/chirpstack-6778f859d8-7bmfh             1/1     Running   14 (2m40s ago)  52d
pod/chirpstack-gateway-bridge-76f5585bf9-9kkfk  1/1     Running   4 (3m49s ago)   52d
pod/chirpstack-rest-api-656764888-pktr8     1/1     Running   4 (3m49s ago)   52d
pod/mosquitto-b9657555-6nzns                1/1     Running   4 (3m49s ago)   52d
pod/postgres-86c9ff4bcd-h9z47               1/1     Running   4 (3m49s ago)   52d
pod/redis-5dd644467b-v2xpm                  1/1     Running   4 (3m49s ago)   52d

NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP                PORT(S)          AGE
service/chirpstack                  LoadBalancer   10.152.183.84    192.168.1.51               8080:32269/TCP   52d
service/chirpstack-gateway-bridge   LoadBalancer   10.152.183.55    192.168.1.52,10.0.2.202    1700:31700/UDP   52d
service/chirpstack-rest-api         ClusterIP      10.152.183.188   <none>                     8090/TCP         52d
service/kubernetes                  ClusterIP      10.152.183.1     <none>                     443/TCP          208d
service/mosquitto                   LoadBalancer   10.152.183.34    192.168.1.50               1883:30971/TCP   52d
service/postgres                    ClusterIP      10.152.183.202   <none>                     5432/TCP         52d
service/redis                       ClusterIP      10.152.183.250   <none>                     6379/TCP         52d

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/chirpstack          1/1     1            1           52d
deployment.apps/chirpstack-gateway-bridge   1/1     1            1           52d
deployment.apps/chirpstack-rest-api 1/1     1            1           52d
deployment.apps/mosquitto           1/1     1            1           52d
deployment.apps/postgres            1/1     1            1           52d
deployment.apps/redis               1/1     1            1           52d

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/chirpstack-6778f859d8          1         1         1       52d
replicaset.apps/chirpstack-gateway-bridge-76f5585bf9   1         1         1       52d
replicaset.apps/chirpstack-rest-api-656764888  1         1         1       52d
replicaset.apps/mosquitto-b9657555             1         1         1       52d
replicaset.apps/postgres-86c9ff4bcd            1         1         1       52d
replicaset.apps/redis-5dd644467b               1         1         1       52d
vboxuser@Ubuntu:~$
```

Figura A.1: Estado del clúster mediante *kubectl get all*

Con la información obtenida, podemos acceder al *gateway* y configurar tanto su IP como el *forwarder* para el reenvío de paquetes de datos hacia el servidor de red (figura A.2). Para ello, se accede a la IP estática del *gateway* y se introducen las credenciales de usuario *sentrius* y contraseña *RG1xx*.

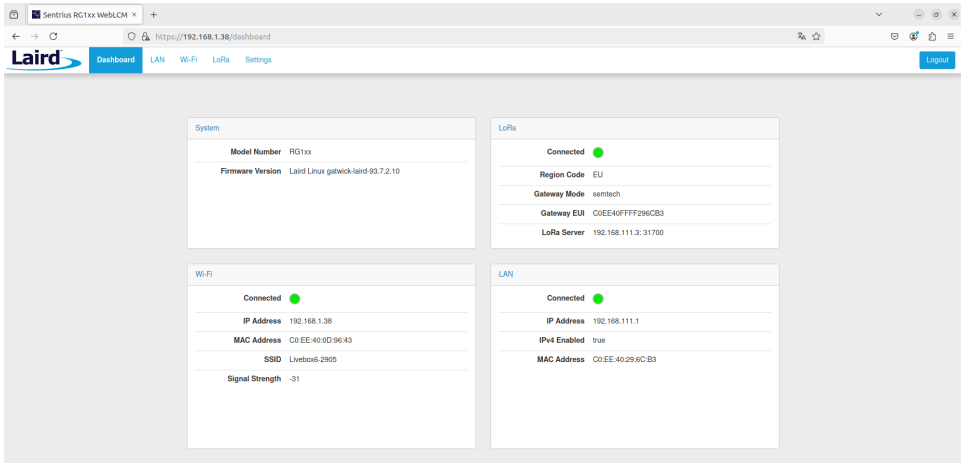

Figura A.2: *Dashboard* del *gateway*

Ahora, se accede a la interfaz web del servidor ChirpStack en la dirección *http://192.168.1.51:8080* e introducimos las credenciales de usuario *admin* y contraseña *admin*.

Una vez dentro de la interfaz de ChirpStack, se realizan una serie de pasos para configurar la red con los dispositivos que forman parte de ella:

1. **Creación de una organización**: en el apartado *tenant*, donde se incluyen los *gateways* y aplicaciones.

2. **Creación de gateways**: dentro de la organización se accede al apartado de *gateways* y se añaden las pasarelas que conforman la red. Como se muestra en la figura A.3, se añade tanto el nombre como su identificador de dispositivo.

Figura A.3: Creación del gateway en ChirpStack

3. **Creación de perfil de dispositivo**: en el apartado *Devices Profiles* se crean los perfiles de los dispositivos. En él se añade el nombre, la región y frecuencia en la que se encuentra (en nuestro caso EU868), la MAC del dispositivo, algoritmo ADR (en este caso por defecto) y el intervalo de tiempo esperado de subida en segundos (figura A.4). La activación de la mota se hace mediante ABP, así que se desactiva la opción mediante OTAA.

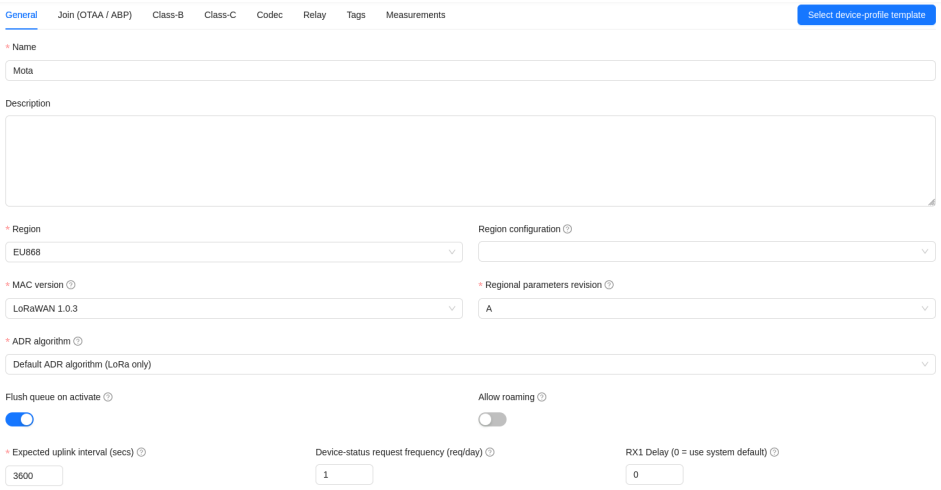

Figura A.4: Creación del perfil de dispositivo en ChirpStack

4. **Creación de aplicación**: se crea la aplicación en el apartado *Applications* con el nombre que se desee. Ya dentro de la aplicación, se añade el equipo indicando su nombre, el *device EUI64*, el *join EUI64* y se elige el perfil del dispositivo (figura A.5). Por otro lado, en el apartado de *Activation* del dispositivo añadido, se indica su dirección, su *NwkSKey* y su *AppSKey* incluidos en el *firmware* de la mota (figura A.6).

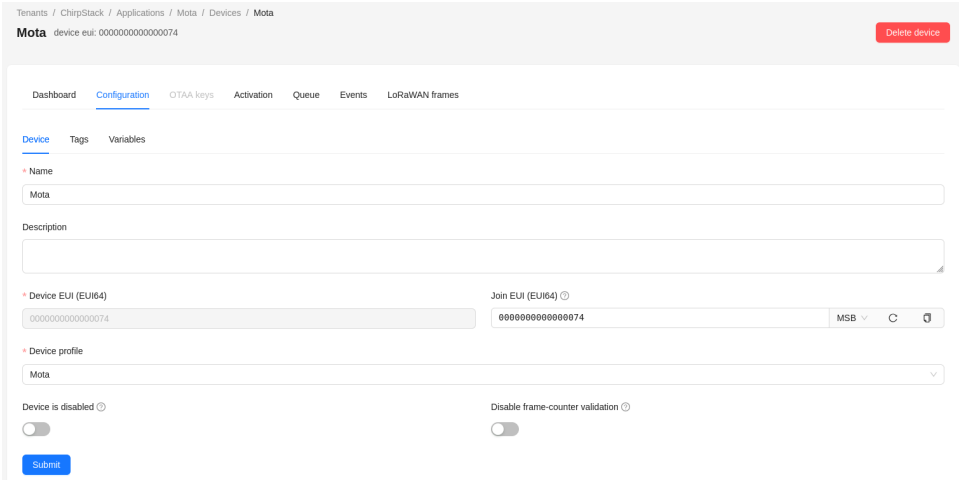

Figura A.5: Creación de la aplicación en ChirpStack

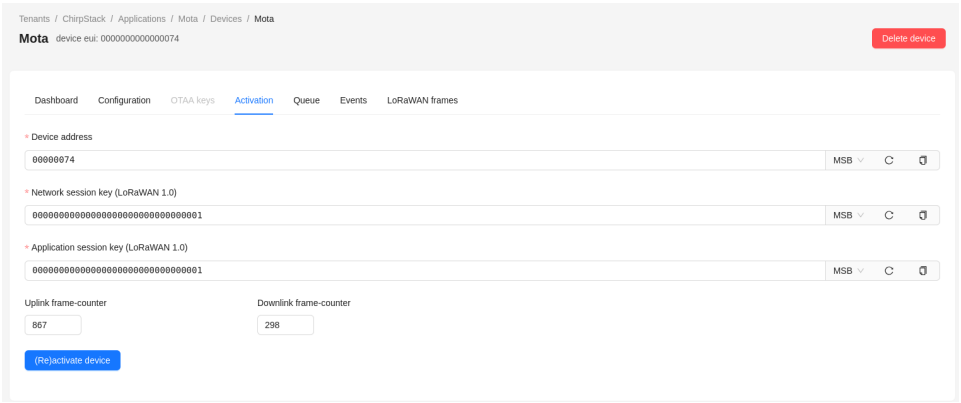

Figura A.6: Activación de la aplicación en ChirpStack

Una vez realizados los pasos anteriores, en la figura A.7 se puede ver que se han detectado los dispositivos en la organización creada, así como los

mensajes que llegan al *gateway* encriptados y los que se reciben y envían en la aplicación desencriptados (figura A.8, figura A.9 y figura A.10).

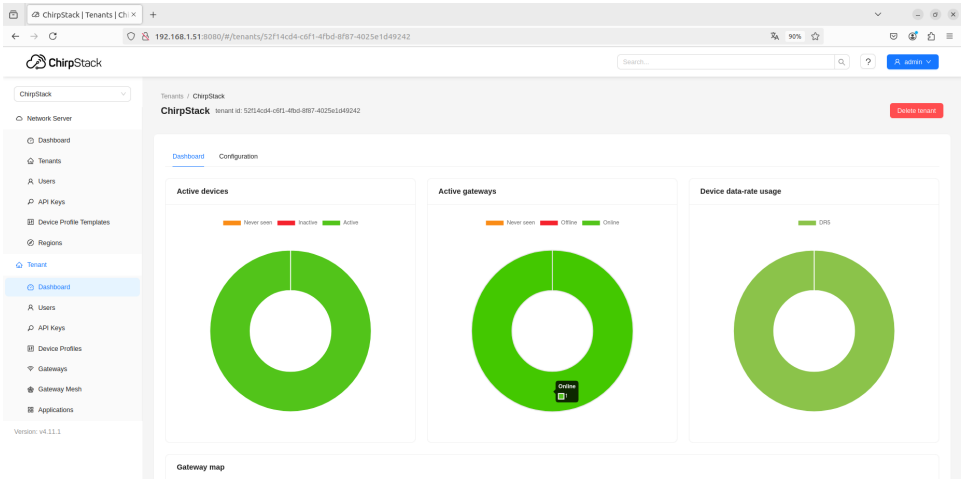

Figura A.7: Detección de dispositivos en ChirpStack

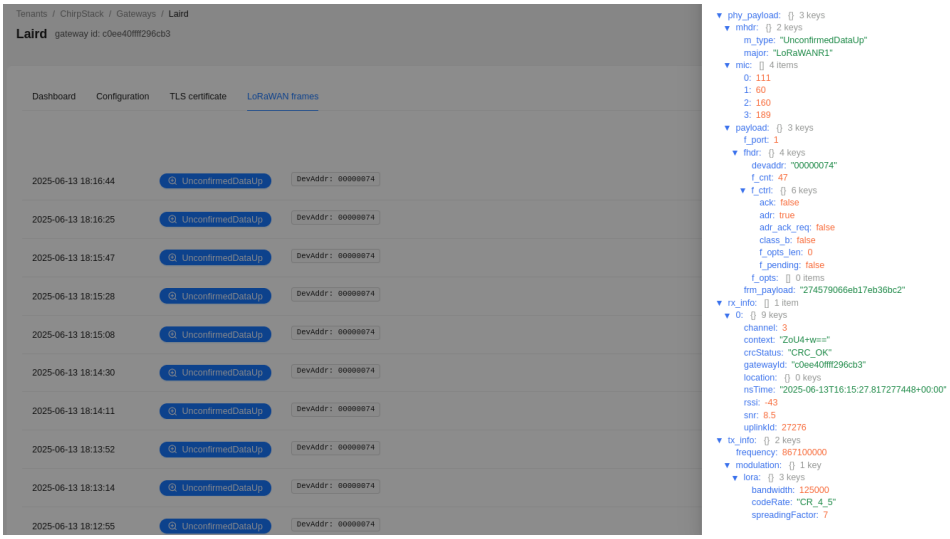

Figura A.8: Mensaje recibido por el *gateway* desde ChirpStack

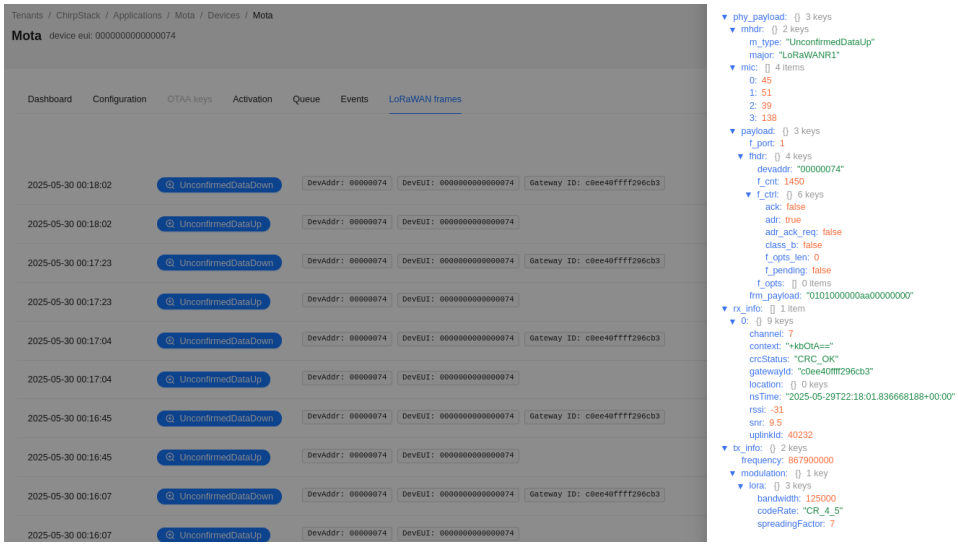

Figura A.9: Mensaje recibido por el servidor LoRaWAN desde ChirpStack

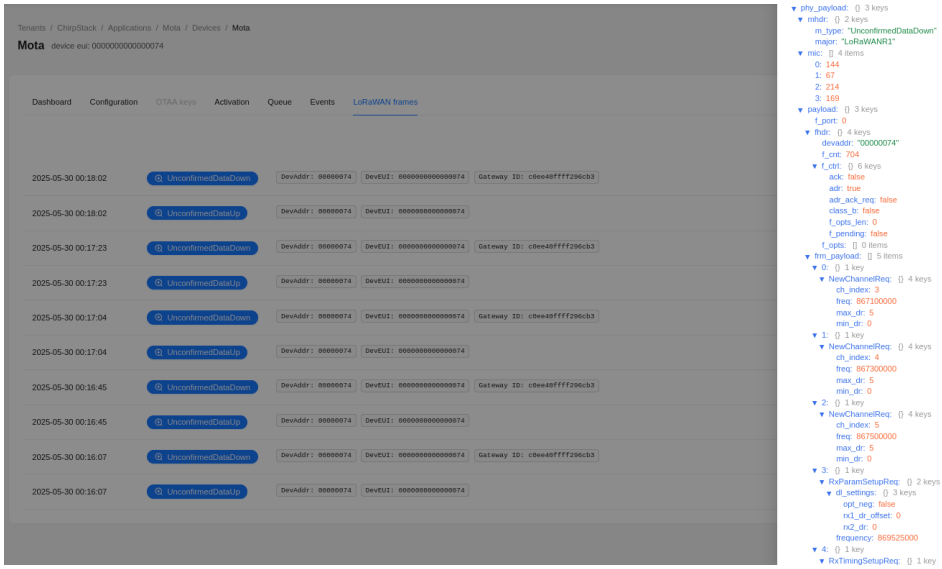

Figura A.10: Mensaje enviado por el servidor LoRaWAN desde ChirpStack

Por último, si queremos detener los servicios del clúster, hacemos uso del comando:

```
$ microk8s stop
```

Apéndice B

Anexos

B.1. Archivos YAML

B.1.1. Redis

```
\begin{verbatim}
# Redis deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: redis
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: redis
  strategy:
    type: Recreate
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
```

```
        io.kompose.service: redis
    spec:
      containers:
        - image: redis:7-alpine
          name: redis
          resources: {}
          volumeMounts:
            - mountPath: /data
              name: redisdata
      restartPolicy: Always
      volumes:
        - name: redisdata
          persistentVolumeClaim:
            claimName: redisdata
status: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# Redis Service
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: redis
  name: redis
spec:
  ports:
    - name: "6379"
      port: 6379
      targetPort: 6379
  selector:
    io.kompose.service: redis
status:
  loadBalancer: {}
  \end{verbatim}
```

.....

```
\begin{verbatim}
# Redis PersistentVolumeClaim
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: redisdata
  name: redisdata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
status: {}
\end{verbatim}
```

B.1.2. PostgreSQL

```
\begin{verbatim}
# Postgres Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: postgres
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: postgres
  strategy:
    type: Recreate
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
```

```
        io.kompose.service: postgres
    spec:
      containers:
        - env:
            - name: POSTGRES_USER
              value: postgres
            - name: POSTGRES_PASSWORD
              value: root
          image: postgres:14-alpine
          name: postgres
          resources: {}
          volumeMounts:
#            - mountPath: /docker-entrypoint-initdb.d
#              name: postgres-claim0
            - name: config-postgres-cm
              mountPath: /docker-entrypoint-initdb.d
            - mountPath: /var/lib/postgresql/data
              name: postgresqldata
      restartPolicy: Always
      volumes:
#        - name: postgres-claim0
#          persistentVolumeClaim:
#            claimName: postgres-claim0
        - name: config-postgres-cm
          configMap:
            name: config-postgres
            defaultMode: 0777
        - name: postgresqldata
          persistentVolumeClaim:
            claimName: postgresqldata
status: {}
\end{verbatim}

.....

\begin{verbatim}
# Postgres Service
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
```

```
    io.kompose.service: postgres
  name: postgres
spec:
  ports:
    - name: "5432"
      port: 5432
      targetPort: 5432
  selector:
    io.kompose.service: postgres
status:
  loadBalancer: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# Postgres PersistentVolumeClaim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: postgresqldata
  name: postgresqldata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
status: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# Postgres Configmap
apiVersion: v1
data:
  001-init-chirpstack.sh: |
    #!/bin/bash
    set -e
    echo "Executing 001-init-chirpstack.sh"

    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER"
    ↪  <<-EOSQL
```

```
        create role chirpstack with login password
        ↪  'chirpstack';
        create database chirpstack with owner chirpstack;
    EOSQL
  002-chirpstack_extensions.sh: |
    #!/bin/bash
    set -e
    echo "Executing 002-chirpstack_extensions.sh"

    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER"
    ↪  --dbname="chirpstack" <<-EOSQL
        create extension pg_trgm;
        create extension hstore;
    EOSQL
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-10T21:02:37Z"
  name: config-postgres
  namespace: default
  resourceVersion: "267884"
  uid: 3d0eec77-bfc8-4f72-8e41-0d85e2c16df2
\end{verbatim}
```

B.1.3. Mosquitto

```
\begin{verbatim}
# Mosquitto Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: mosquitto
  name: mosquitto
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: mosquitto
  strategy:
    type: Recreate
```

```
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
        io.kompose.service: mosquitto
    spec:
      containers:
        - image: eclipse-mosquitto:2
          name: mosquitto
          command: ["/bin/sh", "-c"]
          args: ["cp /mosquitto/config/entrypoint.sh
           ↪  /mosquitto/data/entrypoint.sh; chmod -R 777
           ↪  /mosquitto/data/entrypoint.sh;
           ↪  /mosquitto/data/entrypoint.sh"]
          ports:
            - containerPort: 1883
          resources: {}
          volumeMounts:
#             - mountPath: /mosquitto/config/mosquitto.conf
#               name: mosquitto-claim0
            - name: config-mosquitto-cm
              mountPath: /mosquitto/config
            - name: mosquittodata
              mountPath: /mosquitto/data
      restartPolicy: Always
      volumes:
        - name: mosquittodata
          persistentVolumeClaim:
            claimName: mosquittodata
        - name: config-mosquitto-cm
          configMap:
            name: config-mosquitto
status: {}
\end{verbatim}

.....

\begin{verbatim}
# Mosquitto Service
apiVersion: v1
kind: Service
metadata:
```

```
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: mosquitto
  name: mosquitto
spec:
  ports:
    - name: "1883"
      protocol: TCP
      port: 1883
      targetPort: 1883
  selector:
    io.kompose.service: mosquitto
  type: LoadBalancer
status:
  loadBalancer: {}
  \end{verbatim}
```

.....

```
\begin{verbatim}
# Mosquitto PersistentVolumeClaim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: mosquittodata
  name: mosquittodata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
status: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# Mosquitto ConfigMap
apiVersion: v1
```

```
data:
  mosquitto.conf: |
    listener 1883 0.0.0.0
    allow_anonymous false
    password_file /mosquitto/data/passwd
#    tls_version tlsv1.2
#    plugin /usr/lib/mosquitto_dynamic_security.so
#    plugin_opt_config_file
↪  /mosquitto/data/dynamic-security.json
  passwd: |
    # Put your <user>:<password hash> list here. You can add
    ↪  more users to connect to the MQTT broker.
    # User MQTTUSER will be use for the ChirpStack platform.
    ↪  Use the script 'python3 mosquitto_passwd.py
    ↪  <password>' to get the hash of the password, since it
    ↪  appears below.

    ↪  MQTTUSER:$6$VRQC7qoQmP1HdhEv$EqtXc44BOgNM+xFbxFpS1Go9A2O5iPRGJrRcMbn6c3e/ruYOs8C
  entrypoint.sh: |
    #!/bin/ash
    SRCDIRECTORY="/mosquitto/config/"
    DSTDIRECTORY="/mosquitto/data/"

    SRCFILE="dynamic-security.json"
    if test -f "$FILEDST"; then
      echo "File $SRCFILE exists in $DSTDIRECTORY of the
      ↪  persistent volume, keeping that file...";
    else
      echo "Copying file $SRCFILE to $DSTDIRECTORY...";
      cp $SRCDIRECTORY/$SRCFILE $DSTDIRECTORY
      chown mosquitto:mosquitto $DSTDIRECTORY/$SRCFILE
      chmod 0700 $DSTDIRECTORY/$SRCFILE
    fi
    SRCFILE="mosquitto.conf"
    DSTDIRECTORY="/mosquitto/data/"
    if test -f "$FILEDST"; then
      echo "File $SRCFILE exists in $DSTDIRECTORY of the
      ↪  persistent volume, keeping that file...";
    else
      echo "Copying file $SRCFILE to $DSTDIRECTORY...";
      chown mosquitto:mosquitto $DSTDIRECTORY/$SRCFILE
      cp $SRCDIRECTORY/$SRCFILE $DSTDIRECTORY
      chmod 0700 $DSTDIRECTORY/$SRCFILE
    fi
```

```
    SRCFILE="passwd"
    DSTDIRECTORY="/mosquitto/data/"
    if test -f "$FILEDST"; then
      echo "File $SRCFILE exists in $DSTDIRECTORY of the
      ↪  persistent volume, keeping that file...";
    else
      echo "Copying file $SRCFILE to $DSTDIRECTORY...";
      chown mosquitto:mosquitto $DSTDIRECTORY/$SRCFILE
      cp $SRCDIRECTORY/$SRCFILE $DSTDIRECTORY
      chmod 0700 $DSTDIRECTORY/$SRCFILE
    fi

    /docker-entrypoint.sh /usr/sbin/mosquitto -c
    ↪  /mosquitto/data/mosquitto.conf
  dynamic-security.json: |
    {
      "clients":        [{
          "username":        "MQTTUSER",
          "textName":        "Dynsec admin user",

          ↪  "password":        "KaHUYcXeECe+dkTyEQLgr6+o9thzKwlzNoJ08l0Ace90B
          "salt":        "lru2Pbmjjl5/XGMN",
          "iterations":        101,
          "roles":        [{
              "rolename":        "admin"
            }]
        }],
      "roles":        [{
          "rolename":        "admin",
          "acls":        [{
              "acltype":        "publishClientSend",
              "topic":        "$CONTROL/dynamic-security/#",
              "allow":        true
            }, {
              "acltype":        "publishClientReceive",
              "topic":        "$CONTROL/dynamic-security/#",
              "allow":        true
            }, {
              "acltype":        "subscribePattern",
              "topic":        "$CONTROL/dynamic-security/#",
              "allow":        true
            }, {
              "acltype":        "publishClientReceive",
              "topic":        "$SYS/#",
```

```
            "allow":        true
          }, {
            "acltype":        "subscribePattern",
            "topic":        "$SYS/#",
            "allow":        true
          }, {
            "acltype":        "publishClientSend",
            "topic":        "#",
            "allow":        true
          }, {
            "acltype":        "publishClientReceive",
            "topic":        "#",
            "allow":        true
          }, {
            "acltype":        "subscribePattern",
            "topic":        "#",
            "allow":        true
          }, {
            "acltype":        "unsubscribePattern",
            "topic":        "#",
            "allow":        true
          }]
      }],
    "defaultACLAccess":        {
      "publishClientSend":        false,
      "publishClientReceive":        true,
      "subscribe":        false,
      "unsubscribe":        true
    }
  }
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-10T10:37:45Z"
  name: config-mosquitto
  namespace: default
  resourceVersion: "184700"
  uid: 481f1c1d-4d6a-4973-a986-4020b062b131
\end{verbatim}
```

B.1.4. ChirpStack Gateway

```
\begin{verbatim}
# Gateway Deployment
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: chirpstack-gateway-bridge
  name: chirpstack-gateway-bridge
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: chirpstack-gateway-bridge
  strategy:
    type: Recreate
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
        io.kompose.service: chirpstack-gateway-bridge
    spec:
      containers:
        - env:
            - name: INTEGRATION__MQTT__COMMAND_TOPIC_TEMPLATE
              value: eu868/gateway/{{ .GatewayID }}/command/#
            - name: INTEGRATION__MQTT__EVENT_TOPIC_TEMPLATE
              value: eu868/gateway/{{ .GatewayID }}/event/{{
              ↪  .EventType }}
            - name: INTEGRATION__MQTT__STATE_TOPIC_TEMPLATE
              value: eu868/gateway/{{ .GatewayID }}/state/{{
              ↪  .StateType }}
          image: chirpstack/chirpstack-gateway-bridge:4
          name: chirpstack-gateway-bridge
          ports:
            - containerPort: 1700
              protocol: UDP
# JNa: In order to be able to ping other pods...
#          securityContext:
#            runAsUser: 1001
#            rusAsGroup: 0
```

```
#            capabilities:
#              add: ['NET_RAW']
# End JNa
          resources: {}
          volumeMounts:
#            - mountPath: /etc/chirpstack-gateway-bridge
#              name: chirpstack-gateway-bridge-claim0
            - name: config-chirpstack-gateway-bridge-cm
              mountPath: /etc/chirpstack-gateway-bridge
      restartPolicy: Always
      volumes:
#        - name: chirpstack-gateway-bridge-claim0
#          persistentVolumeClaim:
#            claimName: chirpstack-gateway-bridge-claim0
        - name: config-chirpstack-gateway-bridge-cm
          configMap:
            name: config-chirpstack-gateway-bridge
status: {}
\end{verbatim}

.....

\begin{verbatim}
# Gateway Service
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: chirpstack-gateway-bridge
  name: chirpstack-gateway-bridge
spec:
  ports:
    - name: "1700"
      port: 1700
      protocol: UDP
      targetPort: 1700
      nodePort: 31700
  externalIPs:
    - 10.0.2.202
  selector:
    io.kompose.service: chirpstack-gateway-bridge
```

```
  type: LoadBalancer
status:
  loadBalancer: {}
\end{verbatim}

.....

\begin{verbatim}
# Gateway ConfigMap
apiVersion: v1
data:
  chirpstack-gateway-bridge.toml: |
    # See
    ↪ https://www.chirpstack.io/gateway-bridge/install/config/
    ↪ for a full
    # configuration example and documentation.

    [integration.mqtt.auth.generic]
    servers=["tcp://mosquitto:1883"]
    username="MQTTUSER"
    password="MQTTPASSWORD"
    ca_cert=""
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-10T22:47:51Z"
  name: config-chirpstack-gateway-bridge
  namespace: default
  resourceVersion: "282724"
  uid: e7ea0a1a-e94a-4f61-accd-bcd7c41ae2a0
\end{verbatim}
```

B.1.5. ChirpStack

```
\begin{verbatim}
# ChirpStack Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: chirpstack
  name: chirpstack
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: chirpstack
  strategy:
    type: Recreate
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
        io.kompose.service: chirpstack
    spec:
      containers:
        - args:
            - -c
            - /etc/chirpstack
          env:
            - name: MQTT_BROKER_HOST
              value: mosquitto
            - name: POSTGRESQL_HOST
              value: postgres
            - name: REDIS_HOST
              value: redis
          image: chirpstack/chirpstack:4
          name: chirpstack
          ports:
            - containerPort: 8080
          resources: {}
          volumeMounts:
#             - mountPath: /etc/chirpstack
#               name: chirpstack-claim0
            - name: config-chirpstack-cm
              mountPath: /etc/chirpstack
      restartPolicy: Always
      volumes:
#        - name: chirpstack-claim0
#          persistentVolumeClaim:
#            claimName: chirpstack-claim0
        - name: config-chirpstack-cm
          configMap:
```

```
            name: config-chirpstack

status: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# ChirpStack Service
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
  labels:
    io.kompose.service: chirpstack
  name: chirpstack
spec:
  ports:
    - name: "8080"
      port: 8080
      targetPort: 8080
  selector:
    io.kompose.service: chirpstack
  type: LoadBalancer
status:
  loadBalancer: {}
\end{verbatim}
```

.....

```
\begin{verbatim}
# ChirpStack ConfigMap
apiVersion: v1
data:
  chirpstack.toml: |
    # Logging.
    [logging]

      # Log level.
      #
      # Options are: trace, debug, info, warn error.
      level="info"
```

```
# PostgreSQL configuration.
[postgresql]

  # PostgreSQL DSN.
  #
  # Format example:
  ↪  postgres://<USERNAME>:<PASSWORD>@<HOSTNAME>/<DATABASE>?sslmode=<SSLMODE>.
  #
  # SSL mode options:
  #  * disable - no SSL
  #  * require - Always SSL (skip verification)
  #  * verify-ca - Always SSL (verify that the certificate
  ↪  presented by the server was signed by a trusted CA)
  #  * verify-full - Always SSL (verify that the
  ↪  certification presented by the server was signed by
  ↪  a trusted CA and the server host name matches the
  ↪  one in the certificate)

  ↪  dsn="postgres://chirpstack:chirpstack@$POSTGRESQL_HOST/chirpstack?sslmode=disa

  # Max open connections.
  #
  # This sets the max. number of open connections that are
  ↪  allowed in the
  # PostgreSQL connection pool.
  max_open_connections=10

  # Min idle connections.
  #
  # This sets the min. number of idle connections in the
  ↪  PostgreSQL connection
  # pool (0 = equal to max_open_connections).
  min_idle_connections=0


# Redis configuration.
[redis]

  # Server address or addresses.
  #
  # Set multiple addresses when connecting to a cluster.
  servers=[
```

```
    "redis://$REDIS_HOST/",
  ]

  # TLS enabled.
  tls_enabled=false

  # Redis Cluster.
  #
  # Set this to true when the provided URLs are pointing
  ↪  to a Redis Cluster
  # instance.
  cluster=false


# Network related configuration.
[network]

  # Network identifier (NetID, 3 bytes) encoded as HEX
  ↪  (e.g. 010203).
  net_id="000000"

  # Enabled regions.
  #
  # Multiple regions can be enabled simultaneously. Each
  ↪  region must match
  # the 'name' parameter of the region configuration in
  ↪  '[[regions]]'.
  enabled_regions=[
    "eu868",
  ]


# API interface configuration.
[api]

  # interface:port to bind the API interface to.
  bind="0.0.0.0:8080"

  # Secret.
  #
  # This secret is used for generating login and API
  ↪  tokens, make sure this
  # is never exposed. Changing this secret will invalidate
  ↪  all login and API
```

```
      # tokens. The following command can be used to generate
      ↪  a random secret:
      #   openssl rand -base64 32
      secret="WHw0SdwftZyXJ3SnrQXc2tMKMdurxMiM7gnE62vnGwM="


    [integration]
      enabled=["mqtt"]

      [integration.mqtt]
        server="tcp://$MQTT_BROKER_HOST:1883/"
        json=true
        username="MQTTUSER"
        password="MQTTPASSWORD"
        ca_cert=""
  region_eu868.toml: |
    # This file contains an example EU868 configuration.
    [[regions]]

      # ID is an user-defined identifier for this region.
      id="eu868"

      # Description is a short description for this region.
      description="EU868"

      # Common-name refers to the common-name of this region
      ↪  as defined by
      # the LoRa Alliance.
      common_name="EU868"


      # Gateway configuration.
      [regions.gateway]

        # Force gateways as private.
        #
        # If enabled, gateways can only be used by devices
        ↪  under the same tenant.
        force_gws_private=false


        # Gateway backend configuration.
        [regions.gateway.backend]
```

```
# The enabled backend type.
enabled="mqtt"

# MQTT configuration.
[regions.gateway.backend.mqtt]

  # Topic prefix.
  #
  # The topic prefix can be used to define the
  ↪  region of the gateway.
  # Note, there is no need to add a trailing '/' to
  ↪  the prefix. The trailing
  # '/' is automatically added to the prefix if it
  ↪  is configured.
  topic_prefix="eu868"

  # MQTT server (e.g. scheme://host:port where
  ↪  scheme is tcp, ssl or ws)
  server="tcp://$MQTT_BROKER_HOST:1883/"

  # Connect with the given username (optional)
  username="MQTTUSER"

  # Connect with the given password (optional)
  password="MQTTPASSWORD"

  # Quality of service level
  #
  # 0: at most once
  # 1: at least once
  # 2: exactly once
  #
  # Note: an increase of this value will decrease
  ↪  the performance.
  # For more information:
  ↪  https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality
  qos=0

  # Clean session
  #
  # Set the "clean session" flag in the connect
  ↪  message when this client
  # connects to an MQTT broker. By setting this flag
  ↪  you are indicating
```

```
      # that no messages saved by the broker for this
      ↪  client should be delivered.
      clean_session=false

      # Client ID
      #
      # Set the client id to be used by this client when
      ↪  connecting to the MQTT
      # broker. A client id must be no longer than 23
      ↪  characters. If left blank,
      # a random id will be generated by ChirpStack.
      client_id=""

      # Keep alive interval.
      #
      # This defines the maximum time that that should
      ↪  pass without communication
      # between the client and server.
      keep_alive_interval="30s"

      # CA certificate file (optional)
      #
      # Use this when setting up a secure connection
      ↪  (when server uses ssl://...)
      # but the certificate used by the server is not
      ↪  trusted by any CA certificate
      # on the server (e.g. when self generated).
      ca_cert=""

      # TLS certificate file (optional)
      tls_cert=""

      # TLS key file (optional)
      tls_key=""


  # Gateway channel configuration.
  #
  # Note: this configuration is only used in case the
  ↪  gateway is using the
  # ChirpStack Concentratord daemon. In any other case,
  ↪  this configuration
  # is ignored.
  [[regions.gateway.channels]]
```

```
  frequency=868100000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=868300000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=868500000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=867100000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=867300000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=867500000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=867700000
  bandwidth=125000
  modulation="LORA"
  spreading_factors=[7, 8, 9, 10, 11, 12]

[[regions.gateway.channels]]
  frequency=867900000
  bandwidth=125000
```

```
    modulation="LORA"
    spreading_factors=[7, 8, 9, 10, 11, 12]

  [[regions.gateway.channels]]
    frequency=868300000
    bandwidth=250000
    modulation="LORA"
    spreading_factors=[7]

  [[regions.gateway.channels]]
    frequency=868800000
    bandwidth=125000
    modulation="FSK"
    datarate=50000


# Region specific network configuration.
[regions.network]

  # Installation margin (dB) used by the ADR engine.
  #
  # A higher number means that the network-server will
  ↪  keep more margin,
  # resulting in a lower data-rate but decreasing the
  ↪  chance that the
  # device gets disconnected because it is unable to
  ↪  reach one of the
  # surrounded gateways.
  installation_margin=10

  # RX window (Class-A).
  #
  # Set this to:
  # 0: RX1 / RX2
  # 1: RX1 only
  # 2: RX2 only
  rx_window=0

  # RX1 delay (1 - 15 seconds).
  rx1_delay=1

  # RX1 data-rate offset
  rx1_dr_offset=0
```

```
# RX2 data-rate
rx2_dr=0

# RX2 frequency (Hz)
rx2_frequency=869525000

# Prefer RX2 on RX1 data-rate less than.
#
# Prefer RX2 over RX1 based on the RX1 data-rate. When
↪  the RX1 data-rate
# is smaller than the configured value, then the
↪  Network Server will
# first try to schedule the downlink for RX2, failing
↪  that (e.g. the gateway
# has already a payload scheduled at the RX2 timing)
↪  it will try RX1.
rx2_prefer_on_rx1_dr_lt=0

# Prefer RX2 on link budget.
#
# When the link-budget is better for RX2 than for RX1,
↪  the Network Server will first
# try to schedule the downlink in RX2, failing that it
↪  will try RX1.
rx2_prefer_on_link_budget=false

# Downlink TX Power (dBm)
#
# When set to -1, the downlink TX Power from the
↪  configured band will
# be used.
#
# Please consult the LoRaWAN Regional Parameters and
↪  local regulations
# for valid and legal options. Note that the
↪  configured TX Power must be
# supported by your gateway(s).
downlink_tx_power=-1

# ADR is disabled.
adr_disabled=false

# Minimum data-rate.
min_dr=0
```

```
# Maximum data-rate.
max_dr=5


# Rejoin-request configuration (LoRaWAN 1.1)
[regions.network.rejoin_request]

  # Request devices to periodically send
  ↪  rejoin-requests.
  enabled=false

  # The device must send a rejoin-request type 0 at
  ↪  least every 2^(max_count_n + 4)
  # uplink messages. Valid values are 0 to 15.
  max_count_n=0

  # The device must send a rejoin-request type 0 at
  ↪  least every 2^(max_time_n + 10)
  # seconds. Valid values are 0 to 15.
  #
  # 0  = roughly 17 minutes
  # 15 = about 1 year
  max_time_n=0


# Class-B configuration.
[regions.network.class_b]

  # Ping-slot data-rate.
  ping_slot_dr=3

  # Ping-slot frequency (Hz)
  #
  # set this to 0 to use the default frequency plan
  ↪  for the configured region
  # (which could be frequency hopping).
  ping_slot_frequency=0


# Below is the common set of extra channels. Please
↪  make sure that these
# channels are also supported by the gateways.
[[regions.network.extra_channels]]
```

```
        frequency=867100000
        min_dr=0
        max_dr=5

        [[regions.network.extra_channels]]
        frequency=867300000
        min_dr=0
        max_dr=5

        [[regions.network.extra_channels]]
        frequency=867500000
        min_dr=0
        max_dr=5

        [[regions.network.extra_channels]]
        frequency=867700000
        min_dr=0
        max_dr=5

        [[regions.network.extra_channels]]
        frequency=867900000
        min_dr=0
        max_dr=5

kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-10T22:46:44Z"
  name: config-chirpstack
  namespace: default
  resourceVersion: "282577"
  uid: 19570314-e329-4d45-ac05-8ee2d8550ce1
\end{verbatim}
```

B.1.6. ChirpStack Rest API

```
\begin{verbatim}
# ChirpStack-Rest-API Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
```

```
  labels:
    io.kompose.service: chirpstack-rest-api
  name: chirpstack-rest-api
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: chirpstack-rest-api
  strategy: {}
  template:
    metadata:
      annotations:
        kompose.cmd: kompose --file docker-compose.yml convert
        kompose.version: 1.26.0 (40646f47)
      creationTimestamp: null
      labels:
        io.kompose.service: chirpstack-rest-api
    spec:
      containers:
        - args:
            - --server
            - chirpstack:8080
            - --bind
            - 0.0.0.0:8090
            - --insecure
          image: chirpstack/chirpstack-rest-api:4
          name: chirpstack-rest-api
          ports:
            - containerPort: 8090
          resources: {}
      restartPolicy: Always
status: {}
\end{verbatim}

.....

\begin{verbatim}
# ChirpStack-Rest-Api Service
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose --file docker-compose.yml convert
    kompose.version: 1.26.0 (40646f47)
  creationTimestamp: null
```

```
  labels:
    io.kompose.service: chirpstack-rest-api
  name: chirpstack-rest-api
spec:
  ports:
    - name: "8090"
      port: 8090
      targetPort: 8090
  selector:
    io.kompose.service: chirpstack-rest-api
status:
  loadBalancer: {}
\end{verbatim}
```

Bibliografía

[1] LoRa Alliance (2021). *LoRaWAN 1.1 Specification.* https://lora-alliance.org/about-lorawan/ (último acceso 03/04/2025).

[2] Documentation for Kubernetes. *Kubernetes Documentation.* https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/ (último acceso 05/04/2025).

[3] Loriot. *Loriot Official Web Page.* https://loriot.io/ (último acceso 10/04/2025).

[4] Mioty Alliance. *Mioty Alliance Official Web Page.* https://mioty-alliance.com/ (último acceso 10/04/2025).

[5] Loriot. *Loriot LoRaWAN Network Server.* https://loriot.io/lorawan-network-server.html (último acceso 10/04/2025).

[6] Loriot. *Loriot Mioty Service Server.* https://loriot.io/mioty-service-center.html (último acceso 10/04/2025).

[7] Loriot. *Loriot Mioty Service Server.* https://loriot.io/assets/images/star-topology-mioty.svg (último acceso 10/04/2025).

[8] Chirpstack. *The ChirpStack project.* https://www.chirpstack.io/docs/index.html (último acceso 11/04/2025).

[9] ChirpStack. *ChirpStack Architecture.* https://www.chirpstack.io/docs/architecture.html (último acceso 11/04/2025).

[10] The Things Network. *The Things Network Official Web Page.* https://www.thethingsnetwork.org/ (último acceso 15/04/2025).

[11] The Things Network. *The Things Stack Architecture.* https://www.thethingsindustries.com/docs/the-things-stack/concepts/architecture/ (último acceso 15/04/2025).

[12] Docker Swarm. *Docker Swarm Rocks Official Web Page.* https://dockerswarm.rocks/ (último acceso 15/04/2025).

[13] Docker Swarm. *Docker Swarm Architecture.* https://www.geeksforgeeks.org/introduction-to-docker-swarm-mode/ (último acceso 15/04/2025).

[14] Mesos: A platform for fine-grained resource sharing in the data center. *Proceedings of the 8th USENIX conference on Networked systems design and implementation.* https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Hindman.pdf (último acceso 16/04/2025).

[15] Documentation for Apache Mesos. *Apache Mesos Documentation.* https://mesos.apache.org/documentation/latest/architecture/ (último acceso 16/04/2025).

[16] Documentation for Kubernetes. *Kubernetes Concepts.* https://kubernetes.io/es/docs/concepts/ (último acceso 26/04/2025).

[17] Documentation for Kubernetes. *Kubernetes Cluster Architecture.* https://kubernetes.io/docs/concepts/architecture/ (último acceso 26/04/2025).

[18] LoRa Alliance (2021). *¿Qué es LoRaWAN?* https://lora-alliance.org/about-lorawan/ (último acceso 29/04/2025).

[19] LoRa Alliance (2021). *LoRaWAN Architecture.* https://lora-alliance.org/about-lorawan-old/ (último acceso 29/04/2025).

[20] LoRa Alliance security whitepaper (2017). *LoRaWAN Security.* https://lora-alliance.org/wp-content/uploads/2020/11/lorawan_security_whitepaper.pdf (último acceso 30/04/2025).

[21] J. Navarro-Ortiz, S. Sendra, P. Ameigeiras, and J. M. Lopez-Soler, "Integration of LoRaWAN and 4G/5G for the Industrial Internet of Things", *IEEE Communications Magazine*, vol. 56, no. 2, pp. 60–67, 2018.

[22] Sylvain Montagny, "LoRa-LoRaWAN and Internet of Things", *Savoie Mont Blanc University*, pp. 44-48, 2022.

[23] Luis Llamas. *¿Qué es MQTT? Su importancia como protocolo IoT.* https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/ (último acceso 11/06/2025).

[24] Message Queing Telemetry Transport. *Arquitectura de los mensajes.* https://www.paessler.com/es/it-explained/mqtt (último acceso 11/06/2025).

[25] Alon Ronder. *The 4-way handshake WPA/WPA2 encryption protocol.* `https://medium.com/@alonr110/the-4-way-handshake-wpa-wpa2-encryption-protocol-65779a315a64` (último acceso 11/06/2025).

[26] Nigel Poulton, Pushkar Joglekar, "The Kubernetes Book", *Leanpub*, edición 2022.

[27] Documentation for Kubernetes. *Kubernetes Components.* `https://kubernetes.io/docs/concepts/overview/components/` (último acceso 19/05/2025).

[28] Herramienta utilizada para diseñar el esquema de red. *Draw.io.* `https://app.diagrams.net/` (último acceso 03/06/2025).