Diseño e implementación de actualizaciones Over The Air (OTA) para redes de sensores inalámbricas

David García Fernández

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

Curso 2018-2019

Director:

Joaquín Recas

Resumen en castellano

El objeto de este trabajo es el estudio de la viabilidad de las actualizaciones de firmware Over The Air (OTA) utilizando una red LPWAN, en concreto LoRaWAN. Dar soporte para actualizaciones de firmware OTA a día de hoy resulta fundamental y más todavía cuando estos nodos se encuentran en lugares remotos o sitios donde el acceso es muy limitado o peligroso. Tener implementada la funcionalidad para actualizaciones vía OTA permite corregir errores o añadir nuevas funcionalidades a distancia. Uno de los objetivos prioritarios consiste en que la misma actualización pueda llegar a un conjunto de nodos a la vez evitando tener que actualizar los nodos uno por uno, ya que esto supondría el envío de gran cantidad de información duplicada. Para llevar a cabo esta tarea, se ha realizado un estudio en profundidad sobre las características y el funcionamiento de la red LoRaWAN teniendo en cuenta que las redes LPWAN en general dificultan llevar a cabo este tipo de actualizaciones debido a sus conocidas limitaciones en cuanto a tamaño de payload y elevada latencia. Finalmente, este trabajo demuestra la viabilidad de las OTA mediante la aplicación de parches de actualización de firmware a conjuntos de nodos.

Palabras clave

OTA

Over The Air

Pycom Lopy

STM32L475

LoRa

LoRaWAN

Bajo consumo

Firmware OTA

LoRa Server

Red de sensores

Abstract

The aim of this work is the study of the feasibility of firmware updates Over The Air (OTA) using an LPWAN network, specifically LoRaWAN. Supporting OTA firmware updates today is essential and especially when these nodes are located in remote places or places where access is very limited or dangerous. Having the functionality implemented for updates via OTA allows correcting errors or adding new functionalities remotely. One of the priority objectives is that the same update can reach a set of nodes at the same time, avoiding having to update the nodes one by one, since this would mean sending a large amount of duplicate information. To carry out this task, an in-depth study has been carried out on the characteristics and operation of the LoRaWAN network, considering that LPWAN networks in general are not the most suitable networks to carry out this kind of updates due to their known limitations in terms of payload size and high latency. Finally, this work demonstrates the viability of the OTA by applying firmware update patches to node sets.

Keywords

OTA

Over The Air

Pycom Lopy

STM32L475

LoRa

LoRaWAN

Low Power

Firmware OTA

LoRa Server

Sensor Network

Índice general

Agradecimientos

Al Departamento de Arquitectura de Computadores y Automática y en especial a mi tutor, Joaquín Recas, por ofrecerme la posibilidad de investigar en un campo tan interesante como son las redes LPWAN.

Capítulo 1

Introducción

En los últimos años hemos visto un crecimiento muy importante en cuanto al número de dispositivos IOT, la gran mayoría de las estadísticas[1] coinciden en que en un futuro no muy lejano llegaremos hasta los 75 mil millones de dispositivos conectados. Con este ritmo de crecimiento las redes tradicionales se han quedado obsoletas, por lo que estos dispositivos están demandando redes de bajo consumo y largo alcance. Este tipo de tecnologías se denominan LPWAN o Low Power Wide Area Network. Las redes LPWAN son redes utilizadas para cubrir un área muy extensa con un bajo consumo de energía; esto permite que los nodos puedan estar situados en lugares remotos alimentándose con una batería durante largos periodos de tiempo. A diferencia de las redes tradicionales que cuentan con un gran ancho de banda y bajas latencias, las redes LPWAN tienen un ancho de banda muy limitado y una alta latencia, es por esto que estas tecnologías no están orientadas a la transmisión de datos en tiempo real.

Las tecnologías LPWAN utilizan una banda denominada ISM (Industrial, Scientific and Medical); esta banda puede utilizarse de forma libre siempre y cuando se cumplan unos requisitos mínimos, estos requisitos consisten en respetar la potencia máxima de transmisión y el tiempo máximo que se puede ocupar el medio. En Europa esta banda corresponde con los rangos de frecuencia 861-870 MHz y los tiempos máximos que se puede ocupar el medio que van desde el 0,1 %, 1 % y el 10 % en función de la frecuencia concreta que se esté utilizando.

[1] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

Con este proyecto se pretende estudiar las distintas opciones y su viabilidad para que una red de sensores inalámbricos pueda recibir actualizaciones de firmware utilizando una red LoRaWAN. Una red de sensores inalámbricos es un conjunto de nodos que tienen instalados una serie de sensores para medir diferentes variables. También cuentan con al menos un módulo de radio que puede encontrarse ya instalado en la placa o acoplarse externamente; este módulo es el responsable de la comunicación por radio con un servidor de aplicación.

Habitualmente en los nodos IOT se pueden configurar parámetros que afectan al funcionamiento (periodos de muestreo, tipo de filtrado, etc.). Este tipo de nodos IOT además de soportar todo lo anterior pueden utilizar la radio para recibir actualizaciones de software. Esto nos ofrece la posibilidad de actualizar el firmware de uno o más nodos que podrían estar fácilmente a más de 20 kilómetros de distancia de la antena receptora más cercana.

Estos dispositivos IOT llevan dentro de su microcontrolador un programa informático llamado firmware, este programa, que es el encargado de la gestión completa del nodo, es el que pretendemos actualizar; habitualmente estos programas no tienen un gran tamaño y se programan en lenguajes como C y C++. El firmware habitualmente se encuentra en la memoria flash del propio microcontrolador y permanece en ejecución mientras el nodo se encuentra alimentado lo que dificulta enormemente su actualización.

Capítulo 2

Introduction

In recent years we have seen a very important growth in the number of IOT devices, the vast majority of statistics agree that in the not too distant future we will reach up to 75 billion connected devices. With this growth rate, traditional networks have become obsolete, so these devices are demanding networks with low power consumption and long range. These types of technologies are called LPWAN or Low Power Wide Area Network. LPWAN networks are used to cover a very large area with low energy consumption, allowing the nodes to be located in remote places feeding on a battery for long periods of time. Unlike traditional networks, that have high bandwidth and low latencies, LPWAN networks have a very limited bandwidth and high latency, which is why these technologies are not ideal for real-time data transmission.

LPWAN technologies use a band called ISM (Industrial, Scientific and Medical).This band can be used freely as long as minimum requirements are met. These requirements consist of respecting the maximum transmission power and the maximum time that the medium can be occupied. In Europe, this band corresponds to the frequency ranges 861-870 MHz and the maximum times that can be occupied by the medium, range from 0.1 %, 1 % and 10 % depending on the specific frequency that is being used.

This project aims to study the different options and their feasibility so that a wireless sensor network can receive firmware updates using a LoRaWAN network. A network of wireless sensors is a set of nodes that have installed a series of sensors to measure different

variables. They also have at least one radio module that can be found already installed on the board or externally coupled; This module is responsible for radio communication with an application server.

Normally in the IOT nodes, parameters that affect the operation can be configured (sampling periods, type of filtering, etc). This type of IOT nodes in addition to supporting all of the above, can use the radio to receive software updates. This offers the possibility of updating the firmware of one or more nodes that could easily be more than 20 kilometers away from the nearest receiving antenna.

These IOT devices carry within their microcontroller a computer program called firmware, this program, that is responsible for the complete management of the node, is the one we intend to update. Usually these programs do not have a large size and are programmed in languages such as C and C ++. The firmware is usually in the flash memory of the microcontroller itself and remains running while the node is powered making the update procedure difficult.

Capítulo 3

Estado del arte

El presente trabajo se enmarca dentro del proyecto Solar Node, en este proyecto se pretende colocar una serie de nodos en unos puntos estratégicos para medir la radiación solar y, en base a los datos enviados por los nodos calcular la cantidad de radiación solar futura que recibiremos. Este trabajo pretende dar respuesta a múltiples preguntas sobre la viabilidad de las actualizaciones OTA y en caso de demostrar su buen funcionamiento que el citado proyecto incorpore dichas actualizaciones.

Poder actualizar cualquier dispositivo casi siempre está asociado a tener un periodo de vida más largo, a día de hoy esto lo vemos con casi cualquier dispositivo, aunque probablemente donde más lo notemos es con los smartphone. Un smartphone con un sistema operativo antiguo no va a soportar muchas de las aplicaciones de última generación. En muchas ocasiones no recibir actualizaciones no guarda relación con que el hardware del dispositivo sea antiguo si no a cuestiones de marketing. Al final lo que se consigue es dejar obsoleto el dispositivo mucho antes de lo que le correspondería. Con una red de sensores puede ser que nos ocurra algo similar, tener un hardware potente y no poder sacarle más partido al resultar muy costosa la actualización individual de un conjunto de nodos, sobre todo si por la ubicación de los dispositivos resulta muy difícil acceder a ellos.

Durante el presente trabajo se van a implementar las OTA utilizando el hardware del que disponemos:

- Placa Pycom Lopy[1] con un procesador ESP32 fabricado por la empresa Espressif que dispone de 32Mbit (4MB) de memoria flash y 512 KB de memoria RAM, además lleva integrados en la placa módulos para conectividad WiFi, Bluetooth y LoRa. Esta placa también soporta programación a más alto nivel con MicroPython.

Figura 3.1: *Placa Pycom Lopy*

- Placa STM32[2]: Está placa contiene un procesador STM32L475VGT6[10] de bajo consumo con 1 MB de memoria flash, una memoria RAM de 128 KB, otra memoria de propósito general en la placa de 64 Mbit (8MB), un módulo WIFI integrado y sensores de temperatura, humedad y presión. Algunas otras características de esta placa y que no vamos a utilizar son: Bluetooth, NFC, SubGHZ (868MHz), micrófono, etc...

[1] https://pycom.io/
[2] https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html

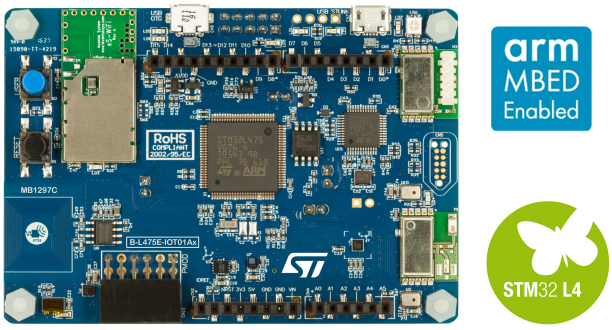

Figura 3.2: *Placa STM32 - B-L475E-IOT01A*

- Módulo LoRa RFM95W[9]/SX1276[3]

Figura 3.3: *Módulo LoRa RFM95W/SX1276*

Para seleccionar la placa con la que finalmente llevar a cabo el desarrollo de este proyecto se ha realizado un breve estudio de ambos datasheet, se han tenido en cuenta las capacidades de cada placa, su antigüedad, consumo y soporte. La placa PyCom de la que se dispone es un modelo ya antiguo y con poco soporte, cuenta con muchos bugs conocidos sin parchear. La placa B-L475E-IOT01A es una placa más moderna, con mayor soporte y

[3]https://www.hoperf.com/modules/lora/RFM95.html

múltiples herramientas como STM32CubeMX para una fácil y rápida configuración. Además está completamente integrada dentro del ecosistema mbed-os lo que proporciona facilidades extra gracias a sus múltiples librerías. Es cierto que para la placa B-L475E-IOT01A se necesita recurrir a un módulo externo de LoRa mientras que para la placa PyCom no es necesario pues lo lleva integrado. En realidad esto no es un problema debido a que por las necesidades del proyecto se debe fabricar una placa adicional donde perfectamente podría ir acoplado dicho módulo. Finalmente y valorando las mejores prestaciones ofrecidas por la placa B-L475E-IOT01A, su mayor soporte y su menor consumo teórico se opta por utilizar esta placa.

Existen múltiples tecnologías LPWAN para el envío y recepción de información con un bajo consumo de energía, en este proyecto se va a utilizar LoRaWAN [4] porque así ha sido impuesto por el proyecto solar node en el que participo. Existen otras alternativas para las redes LPWAN, como puede ser Sigfox, desarrollada por una empresa francesa del mismo nombre, creada en el año 2009 y que está desplegando su propia infraestructura de antenas por todo el mundo. La infraestructura de Sigfox es similar a la que utilizan los operadores de telefonía móvil pero orientada a la conectividad IOT. Sigfox, a diferencia de LoRaWAN es de pago y ofrece en la actualidad cobertura al 95 %[4] de la población en España.

Con LoRaWAN están surgiendo iniciativas para crear una red abierta a nivel mundial[5], donde cualquiera que tenga un gateway lo puede dar de alta y automáticamente los mensajes de cualquier dispositivo que esté registrado en dicha web y que sean capturados por alguno de estos gateways serán visibles para su propietario. Actualmente cuentan con 7639 gateways distribuidos por 138 países.

Las tecnologías tradicionales como el WIFI o el 2G/3G tienen otro tipo de ventajas e inconvenientes, en ambos casos no existen las limitaciones duty-cycle[4.11], el ancho de banda es superior y la latencia más baja, por el contrario el consumo de batería es mayor. En

[4]http://www.redestelecom.es/infraestructuras/noticias/1108555001803/cobertura-de-sigfox-alcanza-95-de-poblacion-espanola.1.html

[5]https://www.thethingsnetwork.org/

el caso concreto de las redes 2G/3G, además estamos hablando de tener que suscribirnos con un operador de telefonía e instalar alguno de los módulos 2G/3G disponibles en el mercado. En este sentido salvo una necesidad de gran ancho de banda y baja latencia no hay una gran diferencia con respecto a utilizar Sigfox.

Capítulo 4

LoRa y LoRaWAN

Es importante tener claras las diferencias existentes entre LoRa y LoRaWAN, a continuación se explica de forma detallada que es cada una.

4.1. LoRa

LoRa es el nombre de la modulación utilizada para transmitir datos a bajo nivel y que permite alcanzar grandes distancias con un bajo consumo de energía, su equivalente en el modelo OSI sería el modelo físico, o dicho de otra forma, es la forma en la que se codifica la información que se transmite de forma inalámbrica. Algunas de las características más importantes que nos ofrece LoRa son las siguientes:

- Muy bajo consumo, ideal para dispositivos con baterías.
- Largo alcance.
- Tamaño de payload limitado (hasta 255 Bytes).
- No se requiere licencia para utilizarlo en las bandas asignadas.
- Soporte para comunicación bidireccional.

Las frecuencias sobre las que trabaja LoRa varían en función de la zona del mundo en la que nos encontremos, a continuación una breve tabla que muestra las principales frecuencias LoRa[1] y la zona en la que se utilizan:

Zona	Frecuencia
Europa	863-870 MHz
Estados Unidos	902-928 MHz
Australia	915-928 MHz
China	470-510 & 779-787 MHZ

Cuadro 4.1: *Tabla de frecuencias LoRa*[1]

4.2. LoRaWAN

LoRaWAN[5] es una capa que trabaja sobre LoRa y que añade nuevas funcionalidades muy interesantes, algunas características son:

- Comunicaciones cifradas con AES-128.
- Diferentes clases para los nodos.
- El tamaño del payload se reduce debido a la información extra que añade LoRaWAN a la trama como porcentaje de batería, MIC (Message Integrity Code), etc...
- Posibilidad de tener redes públicas y privadas.
- Topología en estrella.

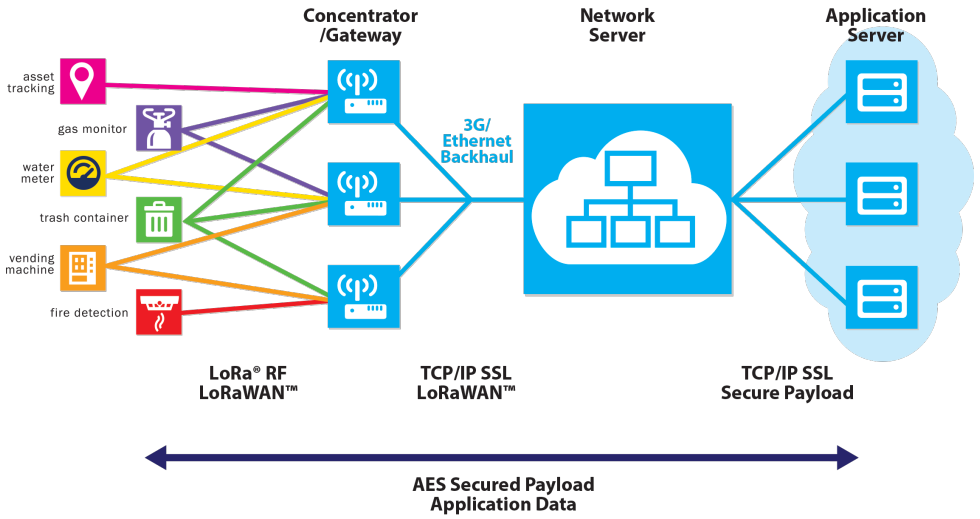

Figura 4.1: *Esquema LoRaWAN*

Fuente: https://lora-alliance.org

En la figura [4.1] podemos ver la estructura que sigue LoRaWAN, en la parte izquierda se encuentran los nodos o dispositivos finales, a continuación los gateways que son los responsables de recibir y transmitir mensajes desde/hacia los nodos, a su vez los gateways se comunican con el servidor responsable de la aplicación mediante un protocolo conocido por ambos, habitualmente MQTT [6.1] o UDP en ambos casos usando JSON pero con una estructura de mensaje diferente.

UDP es un protocolo de la capa de transporte muy rápido, no orientado a conexión y utilizado en aplicaciones donde la latencia es muy importante. En UDP no hay retransmisión de tramas perdidas.

JSON o JavaScript Object Notation, es un formato de texto utilizado para el intercambio de información entre procesos. JSON es ampliamente utilizado por lo que encontramos librerías para extraer la información en la mayoría de lenguajes de programación.

4.2.1. Autenticación con LoRaWAN

Con LoRaWAN es imprescindible que cada dispositivo esté dado de alta en una base de datos antes de comenzar a transmitir, si el dispositivo transmitiera antes de ser dado de alta

la aplicación automáticamente descartaría estos paquetes. Existen dos formas de realizar este proceso, a continuación se detallan las ventajas e inconvenientes de cada una de ellas.

Activation by Personalization (ABP)

La podríamos definir como una activación estática, cada nodo ya lleva configuradas las claves 'Network session key' y 'Application session key' de forma estática, por ejemplo, como constantes en el código fuente. Estas claves van a ser siempre las mismas y en caso de que otra persona se hiciera con ellas podría acceder a toda la información transmitida con destino/origen el nodo afectado.

Algunas de sus características son:

- Menor uso de la red, no se transmite el paquete JOIN REQUEST hacia el gateway ni la posible respuesta desde el gateway JOIN ACCEPT.
- Menor seguridad por usar claves estáticas.
- Parámetros de red que se pueden establecer de forma dinámica mediante el paquete de respuesta a un JOIN REQUEST quedan establecidos de forma estática.

Para que el nodo pueda funcionar en modo ABP debe disponer de al menos las claves 'Network session key', 'Application session key' y un 'Dev EUI'[4.2.1]. Además es necesario que el nodo esté dado de alta en la aplicación con las mismas claves.

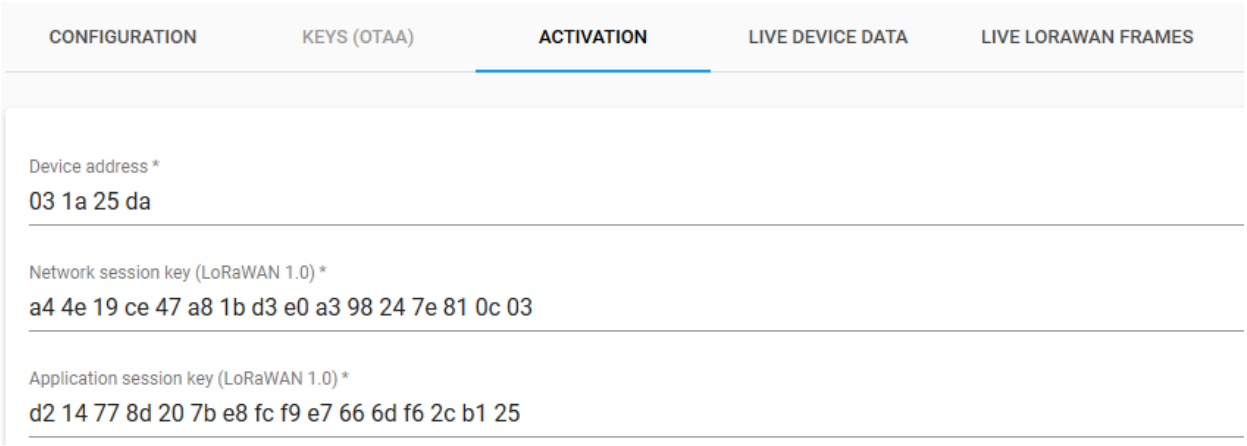

Figura 4.2: *Configuración de las claves de sesión de forma manual para el método de autenticación ABP.*

La figura [4.2] es una captura de pantalla de la aplicación Lora-App-Server instalada en nuestro servidor donde podemos ver los datos necesarios para dar de alta un nuevo nodo con autenticación ABP.

Over-the-Air Activation (OTAA)

Con este método de activación los nodos generan las claves de sesión 'Network session key' y 'Application session key' en el momento en el que quieren unirse a la red, en primer lugar envían un paquete JOIN_REQUEST el cual es verificado por la aplicación donde dichos nodos están registrados, si todos los datos son correctos la aplicación autoriza el acceso mediante el envío de un paquete JOIN_ACCEPT; el nodo utilizará los datos contenidos en este paquete para generar las claves de sesión. Algunas de sus características son:

- Configuración de parámetros de red como 'RxDelay' de forma dinámica mediante la respuesta JOIN_ ACCEPT.
- Mayor seguridad por usar claves dinámicas, las claves se generan de forma dinámica mediante los datos contenidos en el paquete JOIN_ACCEPT.

Hay que tener en cuenta que no todos los dispositivos soportan la activación OTAA. Para utilizar este método de autenticación es necesario configurar 'App EUI', 'App Key', 'Dev EUI':

- App EUI: Identifica la aplicación a la que van dirigidos los datos, es decir, que aplicación debe tratar los mensajes de los nodos.
- App Key: Es una clave 'maestra', se utiliza para establecer la conexión con la aplicación y generar las claves de sesión (NwkSKey y AppSKey).
- Dev EUI: Es un identificador único habitualmente establecido por el fabricante, es equivalente a por ejemplo una dirección MAC y sirve para identificar a un dispositivo de forma única.

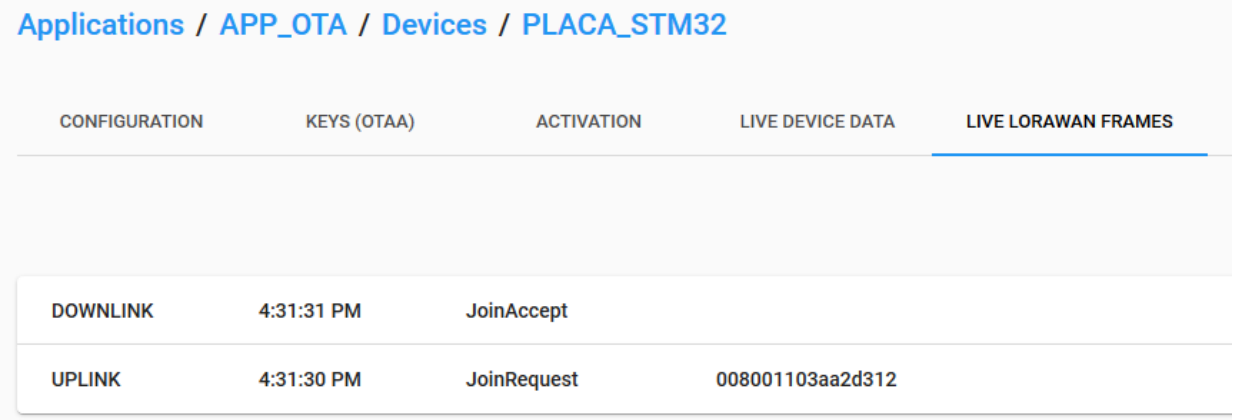

Figura 4.3: *Paquetes 'JOIN REQUEST' Y 'JOIN ACCEPT' en LoRa App Server (autenticación OTAA).*

En la siguiente figura [4.4] podemos observar como las claves han sido generadas de forma dinámica y por tanto no se nos permite modificarlas.

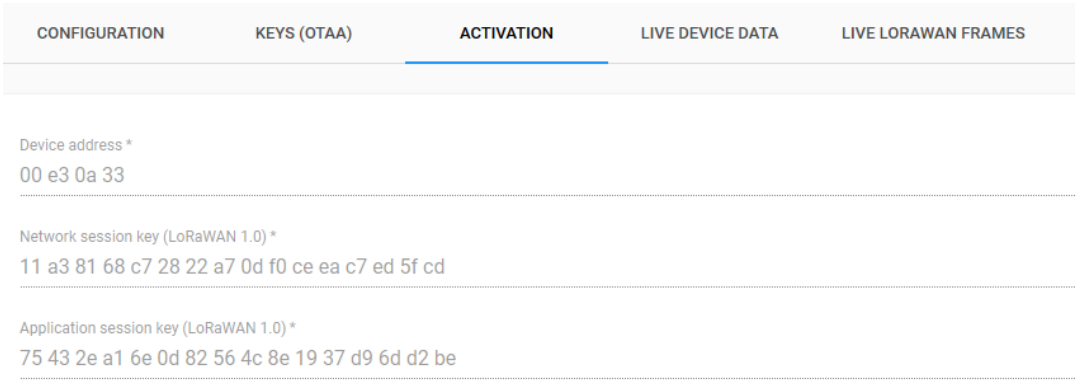

Figura 4.4: *Claves generadas de forma dinámica al unirse el nodo a la red utilizando OTAA como método para la autenticación.*

Es importante añadir que no debemos enviar el mensaje JOIN REQUEST por cualquier frecuencia, la documentación de LoRaWAN nos indica que tres canales podemos utilizar para enviar nuestro mensaje JOIN REQUEST. Ver figura [4.5]. Es obligatorio que estos tres canales estén soportados por todos los nodos, el resto de canales son de libre elección dentro de la banda asignada.

Modulation	Bandwidth [kHz]	Channel Frequency [MHz]	FSK Bitrate or LoRa DR / Bitrate	Nb Channels
LoRa	125	868.10 868.30 868.50	DR0 – DR5 / 0.3-5 kbps	3

Figura 4.5: *Frecuencias en las que podemos enviar un mensaje JOIN REQUEST.*
Fuente: https://lora-alliance.org LoRaWAN Regional Parameters

4.2.2. Clases y ventanas de recepción

En LoRaWAN existen tres modos de funcionamiento diferentes denominados clases, en función de cuál de ellas estemos utilizando podremos recibir datos desde un gateway en cualquier momento, en momentos programados o únicamente después de que el nodo transmita.

- Clase A: por lo general la más utilizada debido a su bajo consumo, el nodo únicamente

puede recibir datos durante un breve periodo de tiempo después de una transmisión. Después de realizar una transmisión el nodo abre la ventana de recepción RX1 en la misma frecuencia sobre la que ha realizado la transmisión. La ventana de recepción RX2 se abre en función de la configuración previa establecida mediante el comando `RXParamSetupReq`.

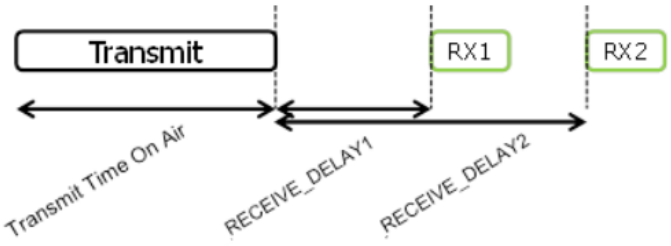

Figura 4.6: *Ventanas de recepción LoRaWAN para dispositivos de la clase A.*
Fuente: https://lora-alliance.org

- Clase B: el nodo puede recibir datos sin necesidad de tener que transmitir antes, el gateway está programado para emitir balizas de sincronización de forma periódica[1] y los nodos reciben estas balizas. Gracias a esto el gateway puede calcular en que momento el nodo estará a la escucha y realizar la transmisión en dicho momento.

- Clase C: el nodo está siempre a la escucha, mayor consumo y no recomendable para dispositivos que funcionen con baterías, puede recibir datos transmitidos por el gateway en cualquier momento excepto si en dicho momento el nodo estuviera transmitiendo.

[1]Habitualmente 128 segundos.

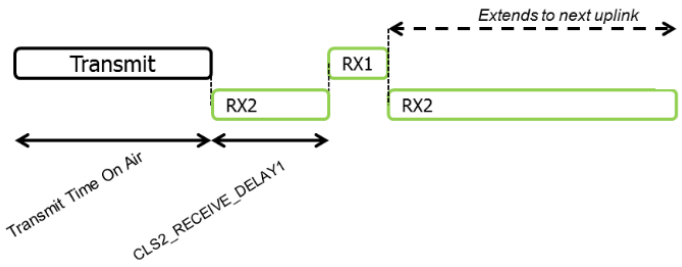

Figura 4.7: *Ventanas de recepción LoRaWAN para dispositivos de la clase C.*
Fuente: https://lora-alliance.org

4.2.3. Formato tramas de datos LoRaWAN

En la figura [4.8] podemos observar el formato que tiene una trama downlink, es decir, datos enviados por el servidor hacia los nodos, los campos PHDR y PHDR_CRC son campos que se rellenan por el transmisor, PHYPayload corresponde con los datos que añade nuestra aplicación y Preamble que se utiliza para sincronizar el receptor.

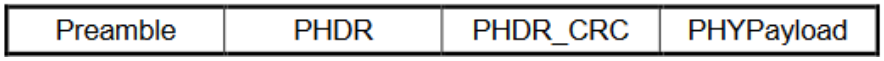

Figura 4.8: *Formato mensaje LoRaWAN enviado desde el servidor hacia los nodos.*
Fuente: https://lora-alliance.org

Por el contrario, la figura [4.9] muestra el formato que tiene una trama uplink, es decir, datos enviados por los nodos hacia el servidor, los campos PHDR, PHDR_CRC y CRC son campos que se rellenan por el transmisor, PHYPayload corresponde con los datos que añade nuestro nodo y Preamble al igual que en el caso anterior que se utiliza para sincronizar el receptor.

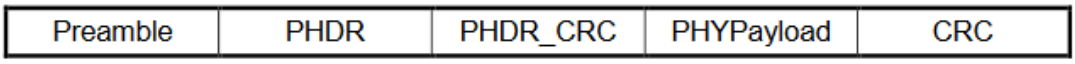

Figura 4.9: *Formato mensaje LoRaWAN enviado por los nodos hacia el servidor.*
Fuente: https://lora-alliance.org

Los tipos de mensajes que nos podemos encontrar en LoRaWAN son los siguientes [4.10]

MType	Description
000	Join-request
001	Join-accept
010	Unconfirmed Data Up
011	Unconfirmed Data Down
100	Confirmed Data Up
101	Confirmed Data Down
110	Rejoin-request
111	Proprietary

Figura 4.10: *Tipos de mensajes soportados por LoRaWAN.*
Fuente: https://lora-alliance.org

- Los mensajes Join-request y Join-accept corresponden a los mensajes intercambiados entre el nodo y la aplicación cuando un nodo solicita unirse mediante OTAA. Ver más en el apartado [4.2.1].

- Mensajes Unconfirmed Data Up y Unconfirmed Data Down corresponden a mensajes de datos enviados por los nodos y la aplicación respectivamente y que no tienen que ser confirmados, es decir, no hay que enviar un ACK como acuse de recibo.

- Mensajes Confirmed Data Up y Confirmed Data Down igual que los anteriores, corresponden a mensajes de datos enviados por los nodos y la aplicación respectivamente y que tienen que ser confirmados mediante el envío de un ACK. El mensaje de confirmación ACK se puede combinar con otros datos en el mismo mensaje.

- Rejoin-request este mensaje solo puede ser enviado por los nodos que se han unido utilizando como método de autenticación OTAA, sirve para re-inicializar una sesión, esto es útil si queremos cambiar algún parámetro que fue establecido en la antigua sesión o si por algún motivo perdemos el contexto de la sesión.

- El mensaje Proprietary nos permite definir dentro del protocolo nuestro propio formato de mensaje. Dado que este formato lo desarrollaríamos nosotros, sería responsabilidad

nuestra implementarlo tanto en los nodos como en la aplicación encargada de gestionar los mensajes.

4.2.4. Limitaciones Duty-cycle LoRaWAN

El tiempo que podemos transmitir con LoRaWAN está regulado, en nuestro caso por la regulación europea ETSI EN 300 220-1[7]. La figura [4.11] es un extracto de la parte que nos afecta para este proyecto.

Frequency Bands/frequencies	Applications	Maximum radiated power, e.r.p. / power spectral density	Channel spacing	Spectrum access and mitigation requirement (e.g. Duty cycle or LBT + AFA)
868,000 MHz to 868,600 MHz (see note 4)	Non-specific use	25 mW	No requirement (see note 6)	1 % or LBT + AFA (see note 3)
868,600 MHz to 868,700 MHz	Alarms	10 mW	25 kHz The whole stated frequency band may be used as 1 wideband channel for high speed data transmission	1 %
868,700 MHz to 869,200 MHz (see note 4)	Non-specific use	25 mW	No requirement (see note 6)	0,1 % or LBT + AFA (see note 3)
869,200 MHz to 869,250 MHz	Social alarms	10 mW	25 kHz	0,1 %
869,250 MHz to 869,300 MHz	Alarms	10 mW	25 kHz	0,1 %
869,300 MHz to 869,400 MHz	Alarms	10 mW	25 kHz	1 %
869,400 MHz to 869,650 MHz	Non-specific use	500 mW	≤25 kHz The whole stated frequency band may be used as 1 wideband channel for high speed data transmission	10 % or LBT + AFA (see note 3)
869,650 MHz to 869,700 MHz	Alarms	25 mW	25 kHz	10 %
869,700 MHz to 870,000 MHz (see note 5)	Non-specific use	25 mW	No requirement	1 % or LBT+AFA (see notes 2 and 3)
869,700 MHz to 870,000 MHz (see note 5)	Non-specific use	5 mW	No requirement	No restriction

Figura 4.11: *Porcentaje tiempo que podemos transmitir en cada banda.*

De la tabla podemos deducir que el máximo tiempo que vamos a poder transmitir con LoRaWAN va a estar entre un 0,1 %, 1 % y 10 % por nodo por canal; esto quiere decir que para una transmisión que dura 100 milisegundos con un duty cycle de un 1 % no podríamos

emitir en los siguientes 900 milisegundos. Otra forma de llevar un control podría ser cada 24 horas, de forma que cada 24 horas podríamos estar emitiendo un total de 864 segundos para un duty cycle del 1 %. La duración de una transmisión varía en función del factor de propagación (SF), el ancho de banda (BW), el coding rate (CR) que estemos utilizando y la cantidad de datos que contenga el payload.

- SF, Spreading Factor, a grandes rasgos indica la cantidad de tiempo que estará nuestra transmisión en el aire, a mayor tiempo más inmune a interferencias y menor RSSI[2] para recibir la señal. A menor SF obtendremos una mejor tasa de transmisión pero será necesario un RSSI mayor para recibir la señal correctamente.

- BW, Banwitdh, el ancho de banda utilizado en la transmisión, habitualmente 125 KHz.

- CR, Coding Rate, indica la cantidad de bits que se envían para corrección de errores, a mayor cantidad de bits enviados para corrección de errores más fácil es recuperar completamente un mensaje corrupto; hay que tener en cuenta que todo el espacio dedicado a los bits de corrección de errores es un espacio que no estamos utilizando para enviar nuestra información. LoRaWAN soporta CR 4/5, 4/6,4/7 y 4/8, siendo 4/5 el CR que menos información de recuperación envía y 4/8 el que más.

En la figura [4.11] podemos ver también que entre las frecuencias 869,400 MHz - 869,650 MHz tenemos un 10 % de duty cycle, esta es la razón por la que vamos a utilizar los canales correspondientes a esta banda para realizar el envío del nuevo firmware a los nodos.

En LoRa muchas veces en lugar de aparecer el factor de propagación y el ancho de banda únicamente nos van a dar el valor data rate, a continuación se muestra una tabla que relaciona todos estos valores.

[2]Valor que indica la fuerza que tiene la señal recibida

DataRate	Configuration	Indicative physical bit rate [bit/s]
0	LoRa: SF12 / 125 kHz	250
1	LoRa: SF11 / 125 kHz	440
2	LoRa: SF10 / 125 kHz	980
3	LoRa: SF9 / 125 kHz	1760
4	LoRa: SF8 / 125 kHz	3125
5	LoRa: SF7 / 125 kHz	5470
6	LoRa: SF7 / 250 kHz	11000
7	FSK: 50 kbps	50000
8..14	RFU	
15	Defined in LoRaWAN[1]	

Figura 4.12: *LoRa Data Rate*

Fuente: https://lora-alliance.org LoRaWAN Regional Parameters

4.3. Recapitulación

Este capítulo se ha dedicado a la obtención de información sobre el funcionamiento y características de LoRa y LoRaWAN, todo este trabajo sirve para poder comprender mejor el funcionamiento de esta tecnología y poder aplicarla correctamente a las actualizaciones OTA.

Se ha optado por utilizar LoRaWAN frente a LoRa en este proyecto, esta decisión se toma fundamentada principalmente por la mayor seguridad en las comunicaciones que ofrece LoRaWAN y el soporte para los diferentes modos de funcionamiento. Como método de autenticación vamos a utilizar OTAA ya que ofrece una mayor seguridad al renovar las claves de sesión en cada JOIN y la posibilidad de cambiar de forma dinámica algunos de los parámetros de red. Para el modo de funcionamiento o clase se ha optado por utilizar las clases A y C, el nodo va a permanecer la mayor parte del tiempo en la clase A para ahorrar batería y solo va a recibir datos en las ventanas de recepción abiertas después de una transmisión. El nodo va a cambiar a la clase C una vez ha completado todas las fases previas necesarias para recibir datos multicast, el nodo estará en la clase C el menor tiempo posible, volviendo a la clase A una vez termine de recibir los datos de la actualización o se alcance el tiempo máximo fijado para la sesión.

Capítulo 5

Gateway

Un gateway o puerta de enlace es un dispositivo hardware que hace de intermediario entre dos redes que utilizan protocolos de comunicación diferentes. El gateway reenvía los mensajes de los nodos hacia el servidor y viceversa. La ubicación de un gateway depende mucho del entorno, habitualmente se sitúa en un punto elevado con el objetivo de cubrir el máximo área posible, esto además permite recibir las transmisiones de los nodos con mejor calidad. En este capítulo se muestran los resultados de las diferentes pruebas realizadas a varios gateways de bajo coste.

5.1. Modos de funcionamiento

Los gateway disponen de varios modos de funcionamiento:

- Reenvío de los paquetes hacia un servidor externo, el gateway trabaja en modo packet forwarder y se limita a reenviar todo lo que recibe. Los gateway soportan habitualmente múltiples protocolos para reenviar esta información como por ejemplo MQTT o UDP.
- Network Server, el servidor se encuentra instalado en el propio gateway. Algunos modelos comerciales ya traen instaladas utilidades como la herramienta NODE-RED[1] que puede conectar con el servidor y administrar los nodos sin tener que desplegar una infraestructura.

[1] https://nodered.org/

Node-Red es una herramienta creada por la empresa IBM y escrita en NODEJS que permite conectar hardware con servicios de forma visual. Con Node-Red toda la programación se realiza de forma gráfica creando un flujo de datos entre los nodos, para crear un flujo de datos simplemente basta con insertar dos nodos y unirlos.

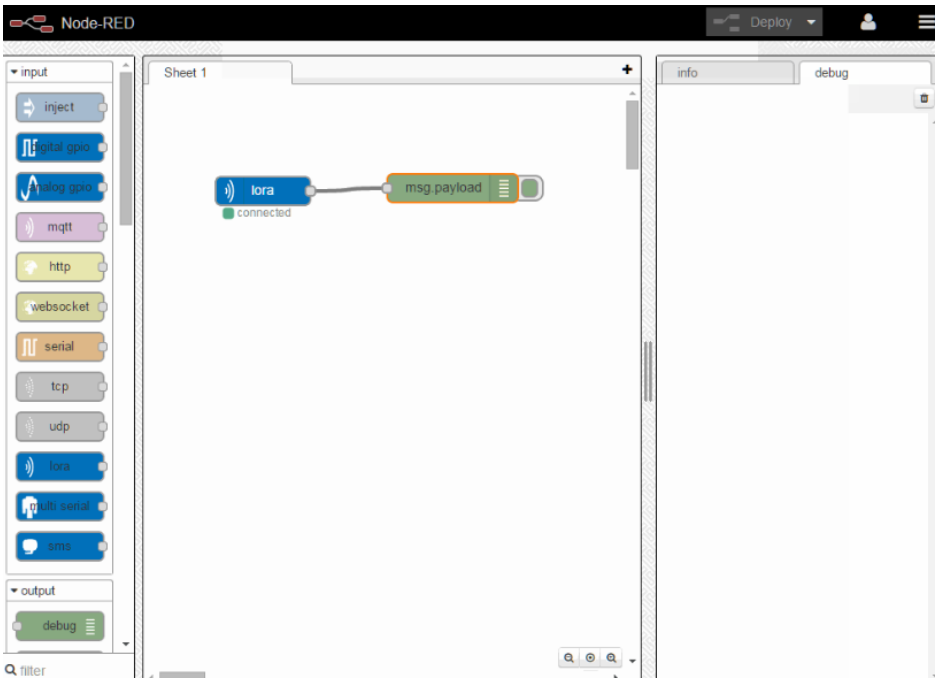

Figura 5.1: *Node-Red de un gateway MultiTech*

En todos los gateway se ha utilizado el modo reenvió de paquetes por ser el único modo soportado. Una diferencia destacable es que ambos gateway de bajo coste utilizan el protocolo UDP para la comunicación con el servidor, el modelo comercial además de UDP también soporta MQTT.

5.2. Pruebas realizadas

Durante el desarrollo de este trabajo se probaron múltiples soluciones de código libre para montar un gateway con soporte LoRaWAN sobre el módulo RFM95W/SX1276 obteniéndose los siguientes resultados:

1. En primer lugar se puso a prueba el proyecto https://github.com/hallard/single_chan_pkt_fwd, utilizando para ello una Raspberry PI y el módulo RFM95W [3.3]. Para conectarlo se ha utilizado el siguiente esquema:

RFM95W PIN	Raspberry PI 2/3
3.3V	1
GND	6
DIO0	7
RESET	11
NSS	22
MOSI	19
MISO	21
SCK	23

Cuadro 5.1: *Tabla de conexiones del módulo RFM95W con Raspberry PI*

Este gateway se prueba de forma inicial sabiendo que únicamente dispone de soporte para una comunicación unidireccional, en este caso comunicación desde los nodos hacia el servidor, con el objetivo de tener una primera toma de contacto con LoRaWAN y el tratamiento de la información recibida.

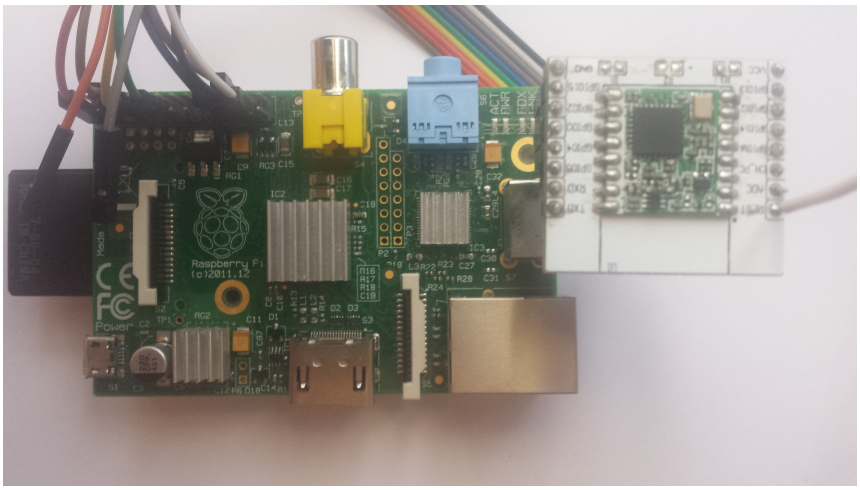

Figura 5.2: *Gateway con un módulo RFM95W y una Raspberry PI*

Es importante configurar en el fichero ”global_conf.json” los pines correspondientes

porque este software también funciona sobre otras placas.

La prueba realizada ha consistido en el envío de paquetes de 1 byte desde una placa PyCom con autenticación ABP, factor de propagación 7 y un ancho de banda 125 KHz. Durante la prueba se ha podido comprobar que pese a las limitaciones que tiene la tasa de paquetes perdidos ha sido realmente baja. Este gateway se podría considerar una buena alternativa para los casos en los que no es necesaria la comunicación con los nodos. Para nuestro proyecto no es una opción a considerar porque la actualización de nodos requiere del envío de datos hacia los nodos. Otro inconveniente que surge es que al no contar con transmisión de paquetes no se puede utilizar el mecanismo de autenticación OTAA, estando limitado únicamente a ABP. Este gateway no soporta el protocolo MQTT por lo tanto para utilizarlo con LoRa Server debemos hacer uso de la herramienta LoRa Gateway Bridge.

```
root@raspberrypi:/home/pi/single_chan_pkt_fwd-master# ./single_chan_pkt_fwd
server: .address = 165.          ; .port = 1700; .enable = 1
server: .address = router.eu.thethings.network; .port = 1700; .enable = 0
Gateway Configuration
  SC Gateway (contact@whatever.com)
  Dragino Single Channel Gateway on RPI
  Latitude=0.00000000
  Longitude=0.00000000
  Altitude=10
Trying to detect module with NSS=6 DIO0=7 Reset=0 Led1=4
SX1276 detected, starting.
Gateway ID:                46:25:99
Listening at SF7 on 868.100000 Mhz.
-----------------------------------
```

Figura 5.3: *Arranque del programa single_chan_pkt_fwd*

Módulo	Precio
Raspberry PI 3 B+	40 €
RFM95W	7 €
Cables para conexión	2 €
Total: 49 €	

Cuadro 5.2: *Coste de los componentes para montar este proyecto.*

Para este gateway no se han medido los consumos porque al no permitir la transmisión de paquetes no resulta relevante.

2. En segundo lugar se pone a prueba el proyecto https://github.com/JaapBraam/LoRaWanGateway, este proyecto a diferencia del anterior si cuenta con la funcionalidad para transmitir datos hacia los nodos. Para poner en marcha este proyecto es necesario disponer de un módulo ESP8266[11] y un módulo RFM95W [3.3].

ESP PIN	RFM95W PIN
D1 (GPIO5)	DIO0
D2 (GPIO4)	DIO1
D5 (GPIO14)	SCK
D6 (GPIO12)	MISO
D7 (GPIO13)	MOSI
D0 (GPIO16)	NSS
GND	GND
3.3V	VCC

Cuadro 5.3: *Tabla de conexiones del módulo RFM95W con ESP8266.*

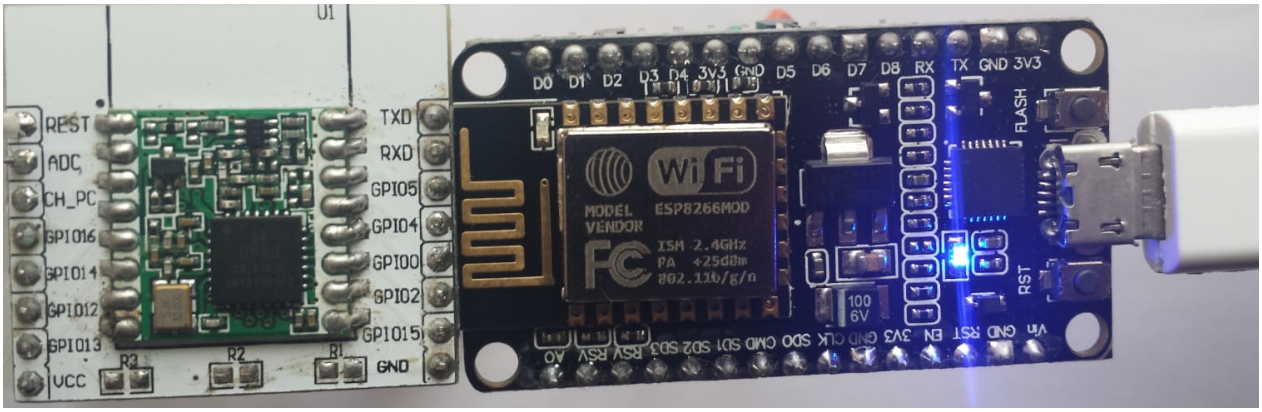

Figura 5.4: *Montaje de un gateway con un módulo ESP8266 y RFM95W*

Las pruebas realizadas consistieron en el envío de un determinado número de paquetes desde una placa STM32L4 [3.2]. Se enviaron un total de 50 paquetes con un payload de un byte, autenticación OTAA, un factor de propagación de 7 y un ancho de banda

de 125 KHz, el tiempo entre los envíos fue de 3 segundos. Como el gateway solo puede estar a la escucha en un único canal se optó por configurar en ambos dispositivos la frecuencia 868.100 MHz; es importante recordar que este gateway no soporta MQTT y en el caso utilizar LoRa Server debemos usar LoRa Gateway Bridge para realizar la conversión de UDP a MQTT.

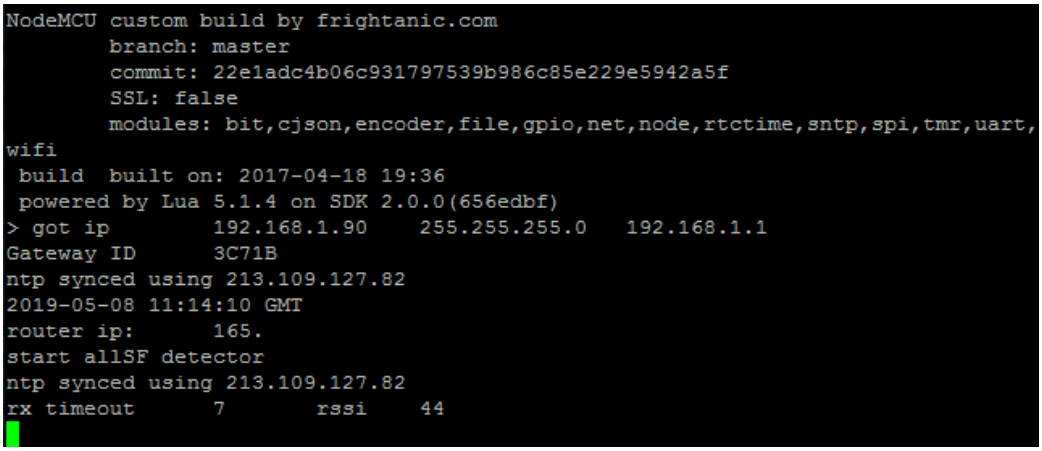

Figura 5.5: *Consola del gateway ESP8266*

La siguiente figura [5.6] es un recorte de la aplicación LoRa App Server donde podemos observar parte de los paquetes recibidos durante la prueba.

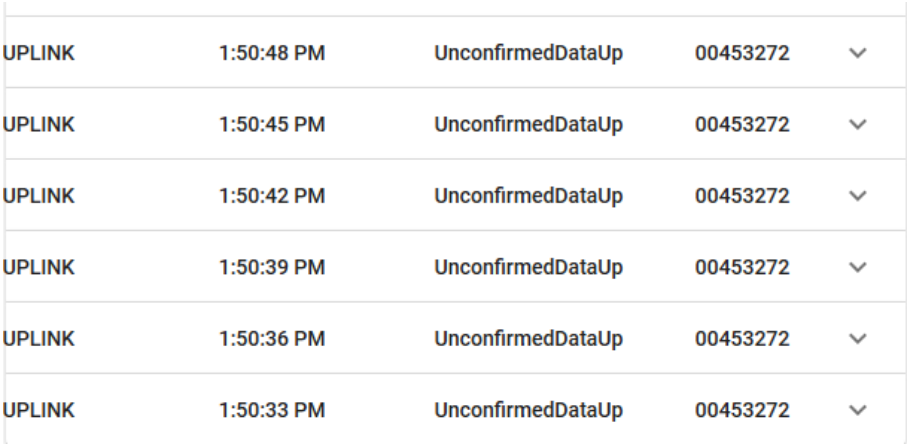

Figura 5.6: *Extracto donde podemos observar parte de los paquetes recibidos durante la prueba.*

Desplegando cualquiera de los paquetes de la figura anterior podemos observar la configuración de la modulación LoRa:

Figura 5.7: *Extracto de uno de los paquetes recibidos donde se ve la configuración utilizada para la transmisión.*

Después de realizar las pruebas anteriores junto con las pruebas de OTA se determina que este gateway no es válido para este proyecto por los siguientes motivos:

- Solo puede recibir datos de un único canal.
- El gateway se reinicia con errores PANIC aleatorios.
- El gateway no puede enviar datos en la frecuencia asignada al downlink del firmware.
- A pesar de que con paquetes de tamaño 1 byte el resultado ha sido muy bueno, durante las pruebas de OTA se observa que con paquetes de un tamaño más grande la sincronización no es buena y se pierden muchos paquetes.

Consumos ESP8266 + RFM95W	
Arranque	Máximo tras varios encendidos: 94 mA
Escuchando	Entre 40-50 mA
Transmitiendo	Entre 90-100 mA

Cuadro 5.4: *Tabla de consumo gateway módulo ESP8266*

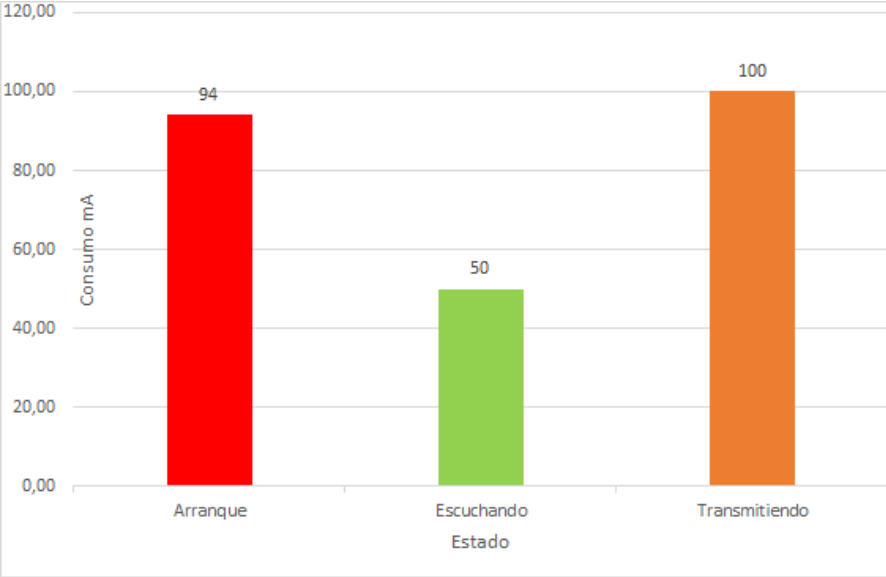

Figura 5.8: *Gráfico de consumo gateway módulo ESP8266*

Los consumos se han medido con un multímetro conectado en serie. En la siguiente tabla [5.5] se encuentra detallado el coste para montar este gateway.

Módulo	Precio
ESP8266	3 €
RFM95W	7 €
Cables para conexión	2 €
Total: 12 €	

Cuadro 5.5: *Coste de los componentes para montar este proyecto.*

3. Finalmente se optó por probar un gateway comercial, modelo 'Sentrius RG1xx'[2]. Este gateway comercial soporta tanto LoRa como LoRaWAN, dispone de conexión Ethernet, Wifi y Bluetooth, soporta y detecta todos los factores de propagación y puede tanto recibir como transmitir por cualquiera de los canales que componen la banda LoRa 868 MHz en Europa. A diferencia del fabricado por Multitech que viene con una

[2]https://www.lairdconnect.com/wireless-modules/lorawan-solutions/sentrius-rg1xx-lora-enabled-gateway-wi-fi-bluetooth-ethernet

versión de Node-Red instalada, este no permite la instalación de ningún paquete.

Figura 5.9: *Gateway Sentrius RG1xx*

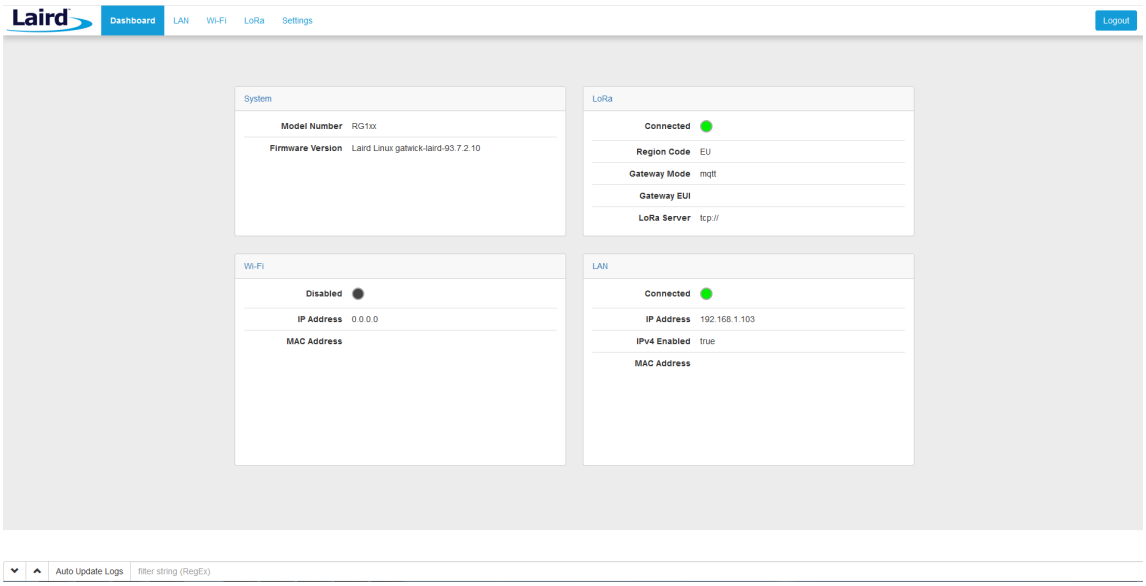

Figura 5.10: *Pantalla inicial del gateway Sentrius RG1xx*

Se realizaron las mismas pruebas que las realizadas a los gateways anteriores, se utilizó la placa STM32L4 con un factor de propagación de 7, un ancho de banda de 125 KHz, autenticación OTAA y sin ninguna restricción en cuanto al canal a utilizar. El porcentaje de paquetes recibidos fue del 100 %. Como la actualización de firmware es un proceso delicado y pretendemos tener la mayor eficiencia posible, utilizaremos este gateway para realizar todas las pruebas de OTA. Este gateway soporta el protocolo MQTT por lo que no necesitaremos ningún software adicional para que funcione con LoRa Server. El precio de este gateway es de 219,00€[3]

[3]https://es.farnell.com/laird-technologies/rg186/gateway-868mhz-wifi-blu-eth/dp/2802548

Capítulo 6

LoRa Server

LoRa Server[1] es un proyecto libre compuesto por múltiples aplicaciones para gestionar de forma sencilla la comunicación entre una aplicación en un servidor y los nodos. LoRa Server soporta la comunicación bidireccional, es decir, podemos tanto enviar mensajes hacia los nodos como recibirlos. A continuación se detallan las tres aplicaciones del proyecto que vamos a utilizar en este trabajo:

- LoRa Server es el corazón del proyecto, es el encargado de gestionar todas las tramas recibidas/enviadas desde/hacia los nodos, comprueba la integridad de los mensajes, gestiona las conexiones OTAA [4.2.1], envía las balizas en caso de que estemos trabajando con la clase B, en general gestiona toda la conexión inalámbrica. LoRa Server no tiene ningún tipo de interfaz gráfica, la única forma de comunicarse con el servidor es mediante los comandos gRPC.

  gRPC, Remote Procedure Call, es un framework desarrollado por la empresa Google que funciona sobre el nuevo protocolo HTTP/2.0; gRPC se utiliza para la conexión entre servicios. La ventaja que ofrece es que estos servicios pueden estar desarrollados en diferentes lenguajes de programación e incluso ejecutándose sobre plataformas diferentes.

- LoRa App Server es una interfaz gráfica para LoRa Server, se comunica con este

[1] https://www.loraserver.io

mediante el protocolo gRPC. LoRa App Server ofrece una API muy completa para comunicarnos con LoRa Server y poder tratar los datos que recibimos de nuestros nodos directamente con una aplicación creada por nosotros para su gestión. La API de LoRa App Server soporta los protocolos gRPC y REST.

- LoRa Gateway Bridge es un software que convierte los datos de los paquetes del protocolo UDP en paquetes del protocolo MQTT y viceversa, esto es necesario con algunos gateways que no soportan el protocolo MQTT y se vuelve necesario realizar la conversión para que sea compatible con LoRa Server.

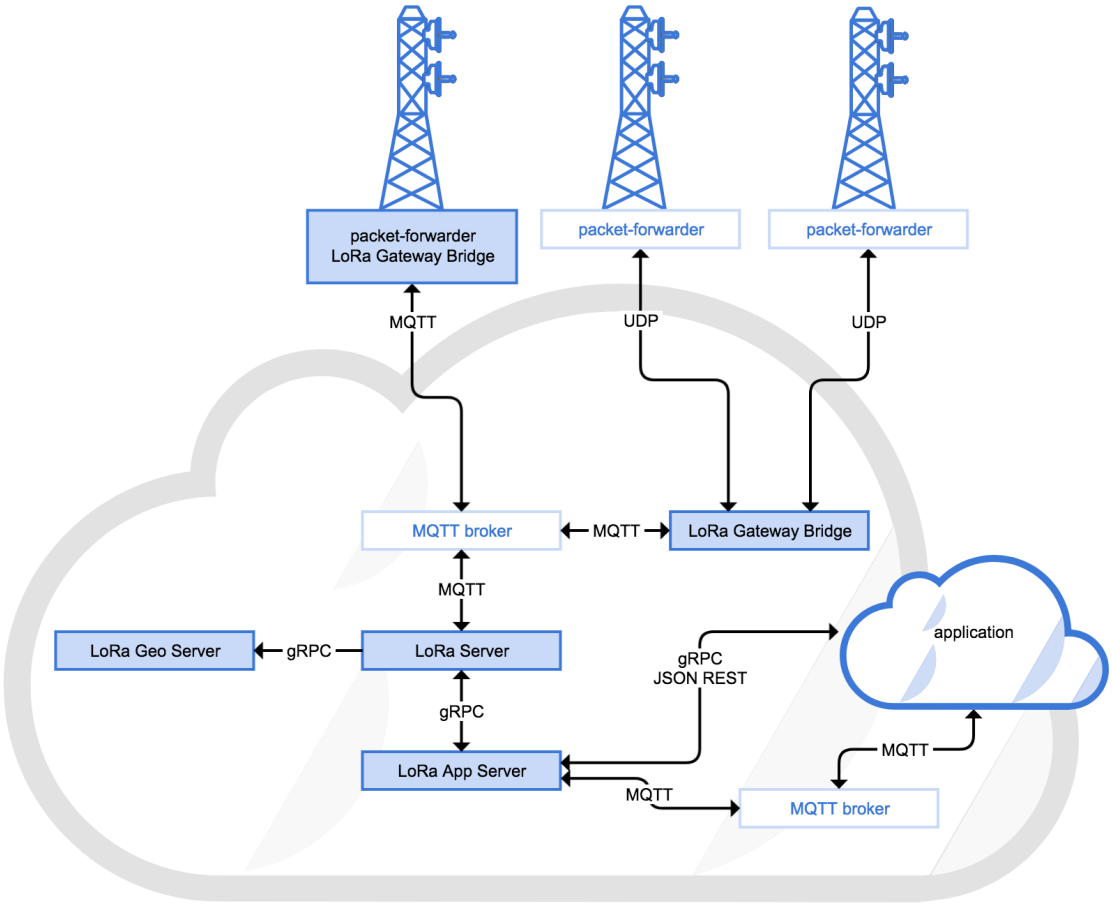

Figura 6.1: *Arquitectura del proyecto LoRa Server con sus múltiples aplicaciones.*
Fuente: https://www.loraserver.io/overview/architecture/

En la figura [6.1] podemos observar cómo se comunican las diferentes aplicaciones que componen LoRa Server, en el centro nos encontramos con el broker MQTT, tanto los gateway como LoRa Server están subscritos a una serie de canales para recibir los mensajes, por ejemplo, para los gateway estos canales tienen el formato gateway/IDENTIFICADOR_GATEWAY/. Continuando por la parte superior vemos que el siguiente salto sería o bien el propio gateway en el caso de soportar el protocolo MQTT o bien el software encargado de realizar la conversión de MQTT a UDP. De la mitad de la figura hacia abajo se encuentran todos los servicios que funcionan con LoRa Server y nuestra aplicación.

6.1. Dependencias

Para que LoRa Server funcione correctamente debemos instalar antes las siguientes dependencias:

- Servidor con soporte MQTT 3.1.1 o superior, por ejemplo, el servidor Mosquitto.[2] MQTT[3] es un protocolo de comunicación máquina a máquina muy ligero y útil en entornos donde contamos con ancho de banda muy limitado. Su funcionamiento se basa en una jerarquía de 'topics' a los que nos podemos subscribir para recibir mensajes. Por ejemplo, vamos a suponer que tenemos colocados nodos a ambos lados de una calle, los nodos del lado izquierdo enviarán sus mensajes al topic `/nodos/calle/izquierda`, mientras que los del lado derecho lo harían a `/nodos/calle/derecha`, por otro lado, una aplicación que tuviera que recibir los mensajes de todos los nodos, independientemente del lado de la calle en el que se encuentren estaría subscrita a `/nodos/calle/#`.

  El comodín `#` sustituye todos los niveles que queramos hacia abajo.

  El comodín `+` únicamente lo utilizamos para un único nivel, por ejemplo, `/nodos/+/temperatura`, que nos daría la temperatura de `/nodos/1/temperatura`, `/nodos/2/temperatura`, etc...

[2] Mosquitto es un servidor de código libre que implementa los protocolos MQTT 3.1, 3.1.1 y 5.0, es un servidor muy ligero y de código libre que se encuentra disponible para la mayoría de las plataformas actuales.

[3] Message Queue Telemetry Transport

- PostgreSQL 9.5 o superior. PostgreSQL es un motor de base de datos relacional de código libre y orientado a objetos. LoRa Server lo utiliza para guardar la información que debe ser persistente como por ejemplo los identificadores de los gateway.

- Redis es un motor de base de datos muy rápido, utiliza la memoria RAM para almacenar la información. Como ya sabemos la memoria RAM es una memoria volátil por lo que los datos no son persistentes, si aun así queremos que sean persistentes a costa de perder rendimiento Redis ofrece algunas posibilidades; en este caso tal y como indica la documentación de LoRa Server, Redis se utiliza para guardar información no persistente por lo que no es necesaria ninguna configuración adicional.

Para desplegar este servicio se opta por alquilar un servidor en la empresa DigitalOcean. DigitalOcean es una empresa estadounidense dedicada entre otras cosas al alquiler de servidores virtuales (VPS) por horas, el más básico desde $0.007 por hora ($5 al mes) que dispone de:

SO	Ubuntu/Debian/FreeBSD/FEDORA/CentOS/*[4]
CPU	1 CPU
RAM	1 GB
Almacenamiento	25 GB SSD
Transferencia[5]	1000 GB

Cuadro 6.1: *Características del servidor más barato de la compañía DigitalOcean.*

Un servidor VPS o Virtual Private Server no es más que una máquina virtual a la cual se le asignan una serie de recursos de la máquina anfitriona en función del precio del servidor virtual, por ejemplo, a mayor precio mayor tamaño de disco duro, mayor número de procesadores, etc..).

[4]También nos permite importar nuestras imágenes personalizadas.

[5]Corresponde con la cantidad de datos incluidos en nuestro paquete que podemos enviar/recibir desde internet.

Capítulo 7

Actualización de nodos vía OTA con LoRaWAN

Cuando tenemos una red de nodos grande resulta muy costoso en tiempo actualizar cada nodo de forma individual. Una transmisión de un firmware completo de aproximadamente 120 KiB puede tardar hasta diez minutos utilizando LoRaWAN, esto asumiendo unas condiciones favorables donde los nodos reciben la señal con una buena calidad y se puede usar un data rate elevado. También debido a las limitaciones duty cycle, si suponemos un tiempo medio de actualización por nodo de 10 minutos, con un duty cycle del 10 % podríamos actualizar aproximadamente 12 nodos al día cumpliendo con la normativa. Resulta mucho más conveniente enviar la actualización a todos los nodos al mismo tiempo, de esta forma el tiempo que ahorramos con respecto al envío individual lo podemos aprovechar con aquellos nodos que por alguna razón no han podido recibir todos los fragmentos correctamente. Es de esperar que alguno/s de los nodos puedan perder una pequeña parte de los paquetes, esto es algo que también podría ocurrir cuando la actualización se hace de forma individual, en el apartado de redundancia se tratará esta parte.

En LoRaWAN el principal problema a la hora de enviar datos a múltiples nodos al mismo tiempo es que cada nodo tiene sus propias claves sesión, esto implica que cada nodo únicamente puede descifrar aquellos mensajes que van exclusivamente dirigidos a el. A finales del año 2018 la asociación LoRa Alliance[1] publicó un estándar acerca de cómo

[1] https://lora-alliance.org

realizar multicast[4] utilizando LoRaWAN, dicho protocolo será el que utilizaremos para la actualización de los nodos.

7.1. Multicast con LoRaWAN

Antes de enviar datos multicast debemos seguir una serie de pasos para configurar nuestros nodos, a continuación y de forma más breve se resumen todos estos pasos. Todos los mensajes intercambiados durante este proceso y sus campos pueden verse en el apéndice [A].

El primer paso es sincronizar el reloj[2] de todos los nodos, este paso previo es fundamental para que todos los nodos puedan cambiar a la clase B o C al mismo tiempo y no perder ningún paquete. En segundo lugar se crea un grupo denominado de multicast, al crear este grupo se consigue que todos los nodos generen las claves de sesión compartidas que se van a utilizar para cifrar/descifrar la información. Una vez los nodos han confirmado su unión al grupo es momento de enviarles los datos de la sesión de fragmentación, en este punto entre otras cosas se envía el tamaño del payload y el número de fragmentos en los que se han dividido los datos. Por último, nos queda indicar al nodo si queremos que abra una ventana de recepción de la clase B (sincronización de ventanas de recepción mediante balizas) o de la clase C (ventana de recepción siempre abierta). En nuestro caso nos interesa que el firmware se descargue en los nodos lo antes posible para que estos puedan seguir funcionando con normalidad; por lo tanto la clase C es la más acertada.

7.2. Actualizando un nodo vía OTA

Una vez vistos en el apartado anterior de forma breve todos los pasos previos necesarios para enviar datos multicast con LoRaWAN es hora de implementarlo en nuestra placa STM32 [3.2]. Para ello vamos a utilizar la aplicación desarrollada por Jan Jongboom que está disponible en https://github.com/ARMmbed/mbed-os-example-lorawan-fuota, este software soporta todo el protocolo establecido por la asociación LoRa-Alliance[6] incluyendo las

actualizaciones delta, es decir, actualizar un nodo enviando solo una lista con los cambios en el firmware, verificación del nuevo firmware mediante una firma criptográfica y un bootloader, que se encargará de gestionar la actualización de firmware en la placa una vez se han recibido todos los fragmentos del nuevo firmware y se ha verificado que la firma es correcta.

Este software funciona sobre la plataforma Mbed. Mbed-OS es un sistema operativo desarrollado por la empresa ARM[2] orientado a IOT para microcontroladores de 32 bits de la familia ARM CORTEX-M. Al añadir una capa sobre el hardware facilita la programación del microcontrolador.

En primer lugar vamos a conectar el módulo RFM95W con la placa siguiendo este esquema:

RFM95W PIN	PLACA STM32L4 PIN
MOSI	D11
MISO	D12
SCK	D13
CS	D10
RESET	A0
DIO0	D2
DIO1	D3
DIO2	D4
DIO3	D5
DIO4	D8
DIO5	D9
3.3V	3V3
GND	GND

Cuadro 7.1: *Tabla de conexiones de la placa STM32L475VGT6 - B-L475E-IOT01A2 con el módulo RFM95W.*

[2]https://os.mbed.com/

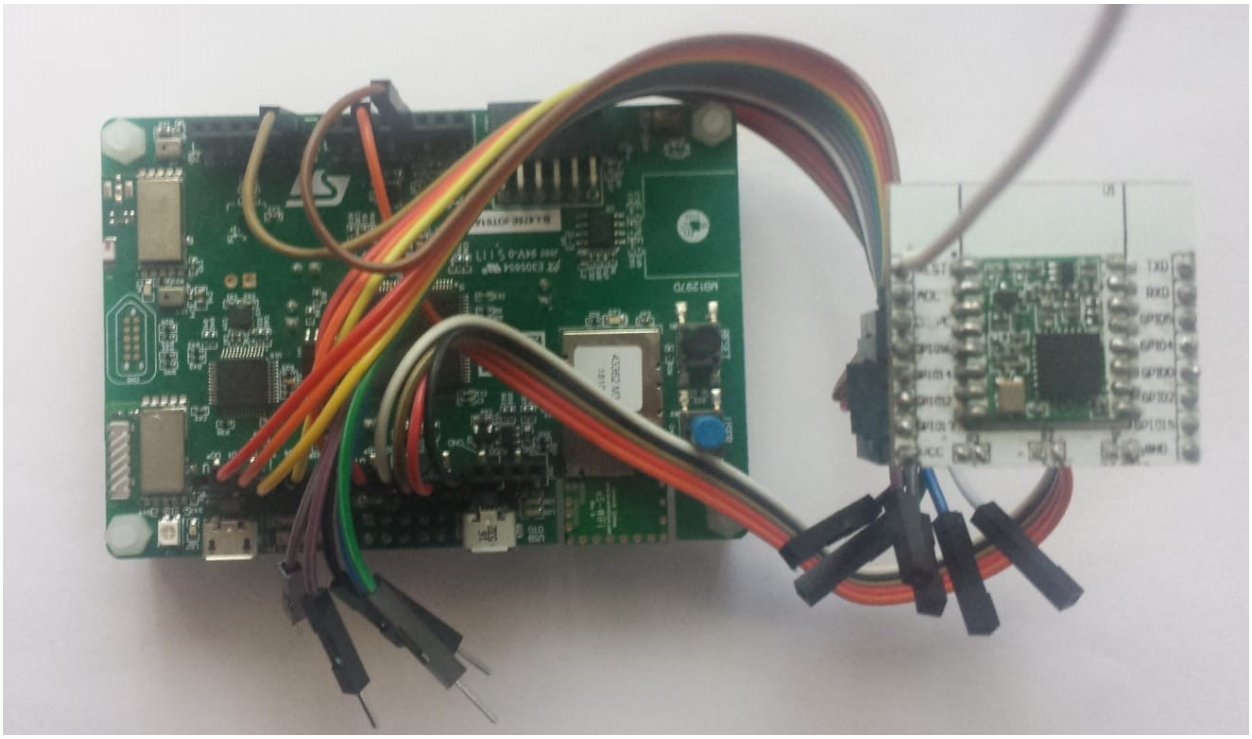

Figura 7.1: *Placa STM32L475VGT6 - B-L475E-IOT01A2 con el módulo RFM95W.*

El módulo RFM95W se vende sin ningún tipo de soporte por lo que es necesario soldarlo primero, en la figura [7.1] se puede observar que el módulo RFM95W ha sido soldado sobre una placa adaptadora.

Con todo el software ya instalado (mbed-cli, nodejs, etc...), en el fichero main.cpp debemos establecer los datos para la conexión OTAA [4.2.1]. Ver figura [7.2].

```
static uint8_t DEV_EUI[] = { 0x00, 0x80, 0x01, 0xA0, 0x00, 0x1D, 0x00, 0x4E };
static uint8_t APP_EUI[] = { 0x70, 0xB3, 0xD5, 0x7E, 0xD0, 0x00, 0xC1, 0x84 };
static uint8_t APP_KEY[] = { 0xB8, 0xB4, 0x33, 0x0D, 0xFD, 0xD5, 0xD8, 0x61, 0xE7, 0x37,
```

Figura 7.2: *Claves para la conexión OTAA.*

Una vez hemos rellenado todas las constantes debemos registrar una aplicación en LoRa Server y dentro de ella dar de alta nuestro nodo, en la figura [7.3] puede verse además que la clave OTAA [4.2.1] coincide con la establecida en el nodo [7.2].

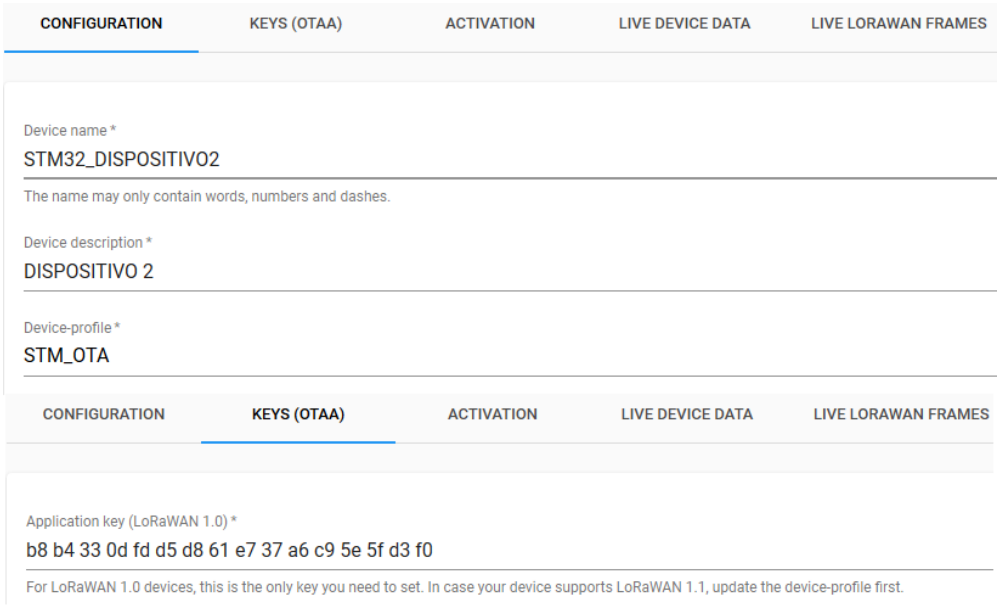

Figura 7.3: *Alta de un nuevo dispositivo OTAA.*

En segundo lugar debemos crear un dispositivo ABP, este dispositivo va a ser el utilizado para el broadcast del firmware. El id generado de forma aleatoria para esta aplicación será de especial interés más adelante. Para obtener las claves que se utilizarán durante la sesión basta con cargar el firmware y ver los mensajes de depuración que aparecen por consola, estas claves serán las mismas para todos los nodos. Para que LoRa-Server envíe los datos de broadcast es necesario que el dispositivo aparezca con una última vez en la aplicación; esto se consigue enviando al menos un paquete de datos desde cualquier dispositivo con dichas claves. En la figura [7.4] podemos ver todos los datos que van a compartir los nodos en el grupo multicast.

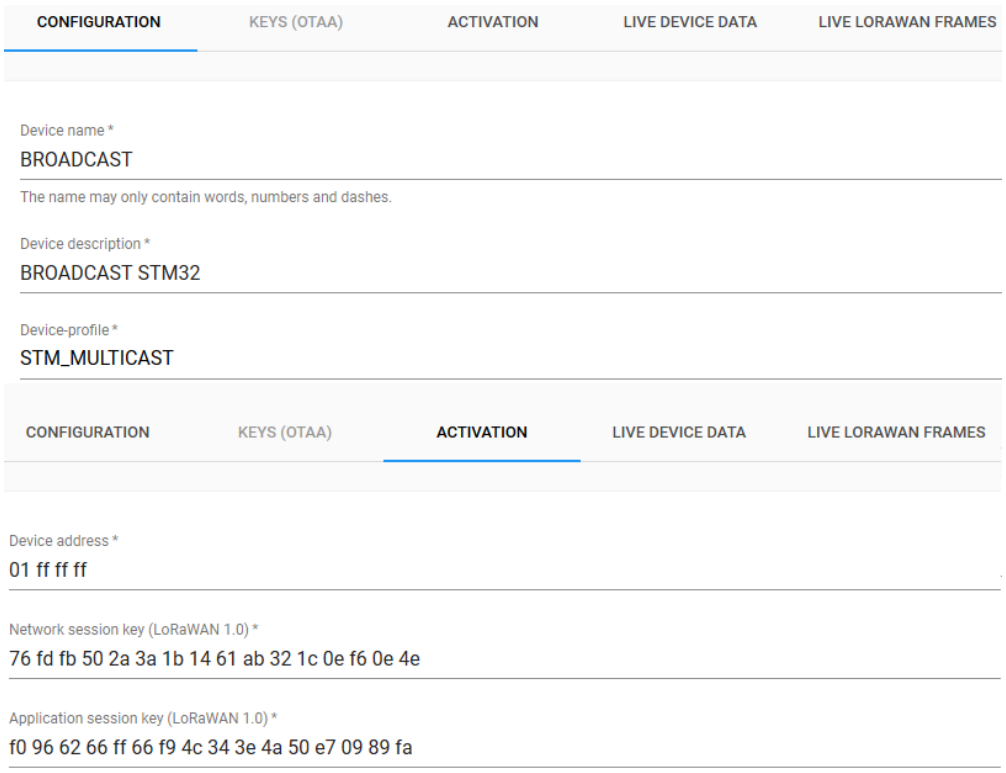

Figura 7.4: *Alta de un nuevo dispositivo ABP.*

Una vez hemos configurado nuestra aplicación en `LoRa-App-Server` es momento de trabajar con el nodo. En este ejemplo se pretende actualizar el firmware mediante una actualización de tipo delta (ver apartado [7.7]). Este tipo de actualización se limita a enviar los cambios entre el firmware guardado en los nodos y el nuevo firmware. En la siguiente figura se puede observar el fragmento de código que ha sido insertado.

```
static void lora_event_handler(lorawan_event_t event) {
    switch (event) {
        case CONNECTED:
            printf("Connection - Successful\n");

            uc.printHeapStats("CONNECTED OTA V10");

            uc.printHeapStats("AGREGAMOS TEXTO PARA AUMENTAR EL TAMAÑO DEL FIRMWARE");
            uc.printHeapStats("AGREGAMOS TEXTO PARA AUMENTAR EL TAMAÑO DEL FIRMWARE");
```

Figura 7.5: *Nuevo código insertado en el firmware.*

Antes de compilar el nuevo firmware debemos hacer una copia del firmware actual, esto nos va a permitir crear la actualización delta comparando ambos firmware. Una vez tenemos una copia del firmware podemos compilar el nuevo código:

```
$ mbed compile -m DISCO_L475VG_IOT01A -t GCC_ARM --profile=./profiles/tiny.json
```

El siguiente paso es generar y firmar el parche, para esto se utiliza el siguiente comando:

```
$ lorawan-fota-signing-tool sign-delta --old old.bin --new new.bin
--output-format bin -o signed-diff.bin
```

Por último dividimos el parche en fragmentos de hasta 204 bytes, en este punto debemos decidir cuantos paquetes de redundancia queremos añadir. Para la placa que estamos utilizando y debido a las limitaciones de memoria, el máximo es de 39 paquetes de redundancia.

```
$ lorawan-fota-signing-tool create-frag-packets -i signed-diff.bin
--output-format plain --frag-size 204 --redundancy-packets 32 -o fragmentos
```

Con esto daríamos por concluida la parte de preparación de la actualización. Es momento de, sabiendo que nodos son los que vamos a actualizar, darlos de alta en el script encargado de tal gestión. Para ello modificamos el fichero `loraserver.js` dentro de la carpeta fuota-server. En la siguiente figura podemos observar algunos nodos de pruebas comentados y un único nodo para actualizar (el que no está comentado). En la parte inferior aparece el identificador del dispositivo encargado del broadcast, este id corresponde con el generado al dar de alta el dispositivo ABP [7.2].

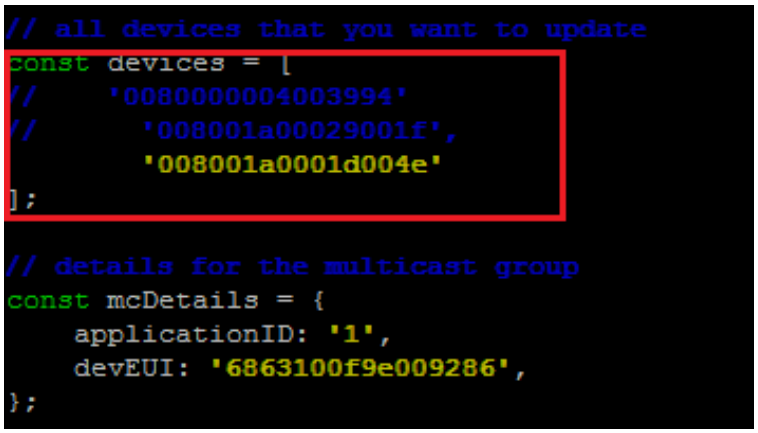

Figura 7.6: *Identificadores de los nodos que se van a actualizar.*

Llegados a este punto no queda más que lanzar el software encargado de las actualizaciones en el servidor e ir viendo como es este proceso. Para ello ejecutamos el siguiente comando:

```
$ node loraserver.js fragmentos
Subscribed to all application events
```

A partir de este momento comienzan a intercambiarse entre el servidor y los nodos una serie de mensajes para preparar la descarga multicast, a continuación se muestra por diferentes secciones cada uno de los pasos. Recordar también que en el Apéndice [A] se encuentra un listado con todo detalle de cada uno de los mensajes intercambiados y sus campos.

7.2.1. Sincronización del reloj

El primer paso para crear un grupo multicast es sincronizar el reloj de todos los nodos, es muy importante tener la hora correctamente sincronizada en los nodos para que estos puedan abrir la ventana de recepción en el momento justo y no pierdan ningún paquete, esto se puede realizar de dos maneras, el nodo puede enviar un mensaje `AppTimeReq` como en este caso o el servidor puede forzar la actualización mediante un mensaje `ForceDeviceResyncReq`. Lo

normal es que sea el nodo, que al entrar en la rutina de actualización sincronice la hora de forma automática. En la siguiente figura podemos observar la respuesta del servidor en el nodo.

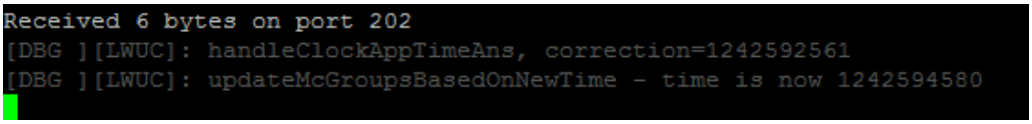

Figura 7.7: *Sincronización de hora en un nodo.*

7.2.2. Configuración del grupo multicast

Una vez que todos los nodos han sincronizado su reloj se procede a enviar un mensaje `McGroupStatusReq` para crear un grupo multicast. En este momento los nodos utilizando la clave `APP_KEY` más una clave enviada desde el servidor pueden calcular las claves de la sesión. En la siguiente figura puede observarse la consola del nodo y las claves de la sesión multicast.

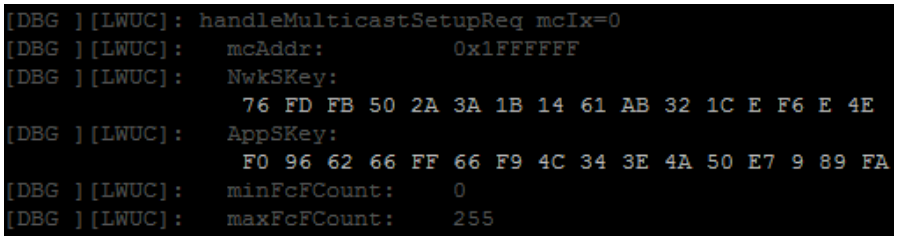

Figura 7.8: *Configuración del grupo multicast en un nodo.*

7.2.3. Fragmentación de paquetes

Dadas las limitaciones en el tamaño del payload que tenemos, debemos dividir el firmware en pequeños fragmentos de hasta 204 bytes por paquete. Esto lo hacemos enviando desde el servidor el mensaje `FragSessionSetupReq`. En la figura se observa entre otras cosas el tamaño de la actualización (32 fragmentos), el tamaño de cada fragmento (204 bytes) y el padding (cuando el último fragmento no está completo, cuanta información hay que

descartar contando por la derecha).

```
Received 11 bytes on port 201
[DBG ][LWUC]: handleFragmentationSetup fragIx=0
[DBG ][LWUC]: FragmentationSessionSetupReq
[DBG ][LWUC]:   Index:            0
[DBG ][LWUC]:   McGroupBitMask:   0
[DBG ][LWUC]:   NbFrag:           32
[DBG ][LWUC]:   FragSize:         204
[DBG ][LWUC]:   FragAlgo:         0
[DBG ][LWUC]:   BlockAckDelay:    0
[DBG ][LWUC]:   Padding:          202
[DBG ][LWUC]:   Descriptor:       0
[DBG ][FSES]: FragmentationSession starting:
[DBG ][FSES]:   NumberOfFragments:   32
[DBG ][FSES]:   FragmentSize:        204
[DBG ][FSES]:   Padding:             202
[DBG ][FSES]:   MaxRedundancy:       39
[DBG ][FSES]:   FlashOffset:         0x1128
```

Figura 7.9: *Configuración de la sesión de fragmentación.*

7.2.4. Ventana de recepción

Por último, nos queda indicar al nodo si queremos que el nodo abra una ventana de recepción de la clase B (sincronización de ventanas de recepción mediante balizas) o de la clase C (ventana de recepción siempre abierta). En nuestro caso nos interesa que el firmware se descargue en los nodos lo antes posible para que estos puedan seguir funcionando con normalidad. Para abrir una ventana de recepción de la clase C el servidor envía el mensaje `McClassCSessionReq`.

```
Received 11 bytes on port 200
[DBG ][LWUC]: handleMulticastClassCSessionReq mcIx=0
[DBG ][LWUC]:   timeToStart:      10
[DBG ][LWUC]:   timeOut:          128
[DBG ][LWUC]:   dlFreq:           869525000
[DBG ][LWUC]:   dataRate:         5
[DBG ][LWUC]: updateClassCSessionAns, originalTimeToStart=10, delta=5, newTimeToStart=5
```

Figura 7.10: *Configuración de la ventana de recepción.*

En la figura podemos ver el tiempo restante para que el nodo abra la ventana de recepción (10 segundos), el timeout (128 segundos), la frecuencia de descarga del firmware, que

pertenece al rango de frecuencias con un duty cycle del 10 % y el data rate [4.12].

7.2.5. Recepción del nuevo firmware

Pasados los 10 segundos indicados en el campo timeToStart [7.10], el servidor comienza a enviar los fragmentos del firmware. En la siguiente figura se ve como el nodo cambia a la clase C y comienza a recibir los fragmentos:

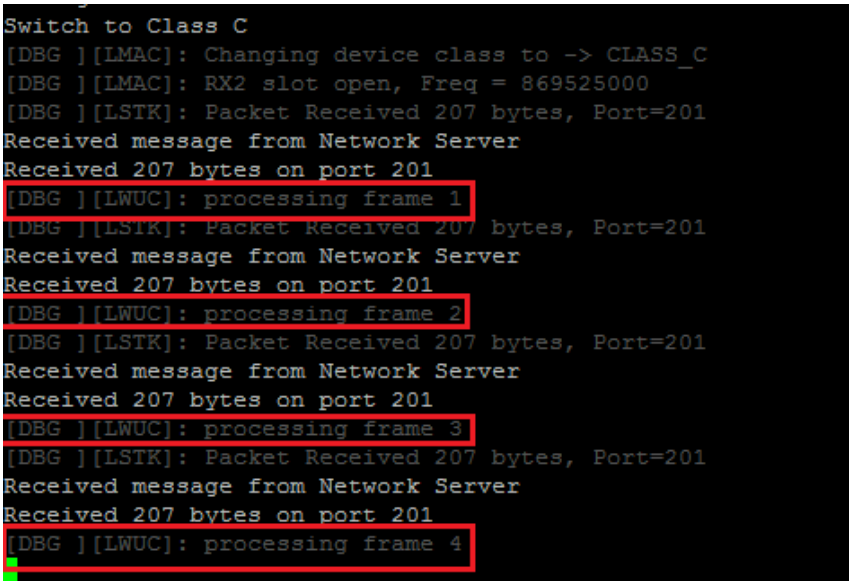

Figura 7.11: *Recepción de los paquetes fragmentados.*

En este caso concreto la actualización está fragmentada en 32 paquetes. Cuando el nodo reciba todos los paquetes comenzará a parchear el firmware. Esto puede verse en la siguiente figura:

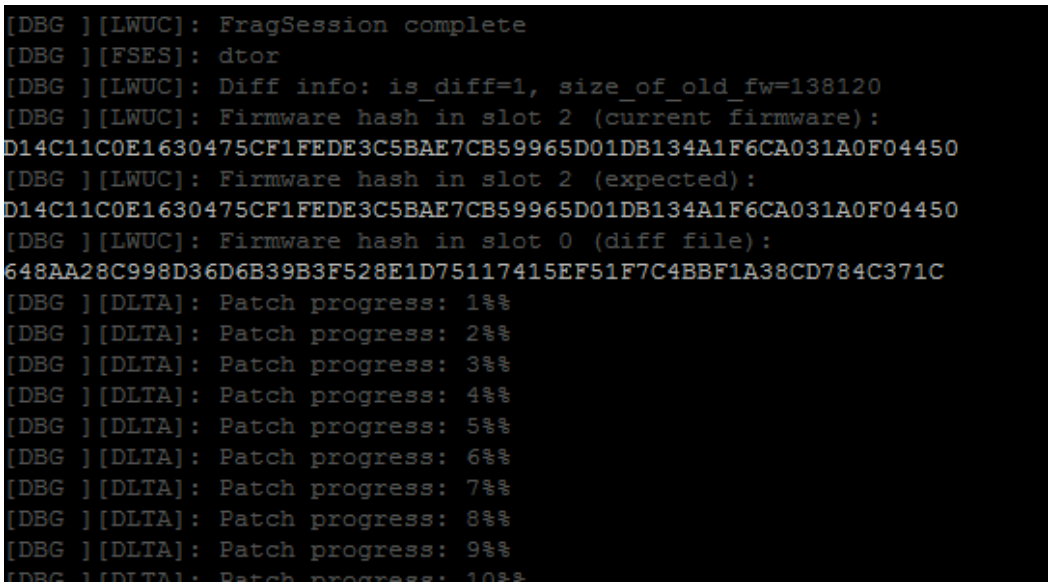

Figura 7.12: *Parcheo del firmware.*

Una vez el nodo termina de parchear el firmware, tiene que verificar utilizando la clave pública que tiene guardada si la firma es correcta, en caso de que la firma sea correcta se escribe en la cabecera del slot 1 (ver apartado [7.8]) la información de versión y la firma. Todo esto puede verse en la siguiente figura:

Figura 7.13: *Verificación de la firma.*

Por último, escrita la información de la cabecera del nuevo firmware se da por concluida la sesión de fragmentación. Ahora el nodo tiene que reiniciarse, esto fuerza la ejecución del

bootloader que se encargará de ir comprobando si hay alguna versión más reciente en alguno de los slot, en caso de encontrarla la copiará donde corresponda.

En la siguiente figura vemos precisamente como ocurre esto, la placa se reinicia automáticamente y el bootloader detecta una versión más reciente en el slot 1[7.8] (el firmware actualizado mediante una actualización delta), entonces el bootloader copia esta versión a la flash del microcontrolador, a continuación detecta que el backup (slot 2) no coincide con el firmware más reciente por lo que copia a la memoria externa, en concreto al slot 2 que sirve como backup del firmware más reciente, el firmware que se encuentra en la memoria flash del microcontrolador. Finalmente se ejecuta el nuevo firmware, podemos ver al final de la figura el texto 'AGREGAMOS TEXTO PARA ...' que fue la modificación realizada en el código fuente.

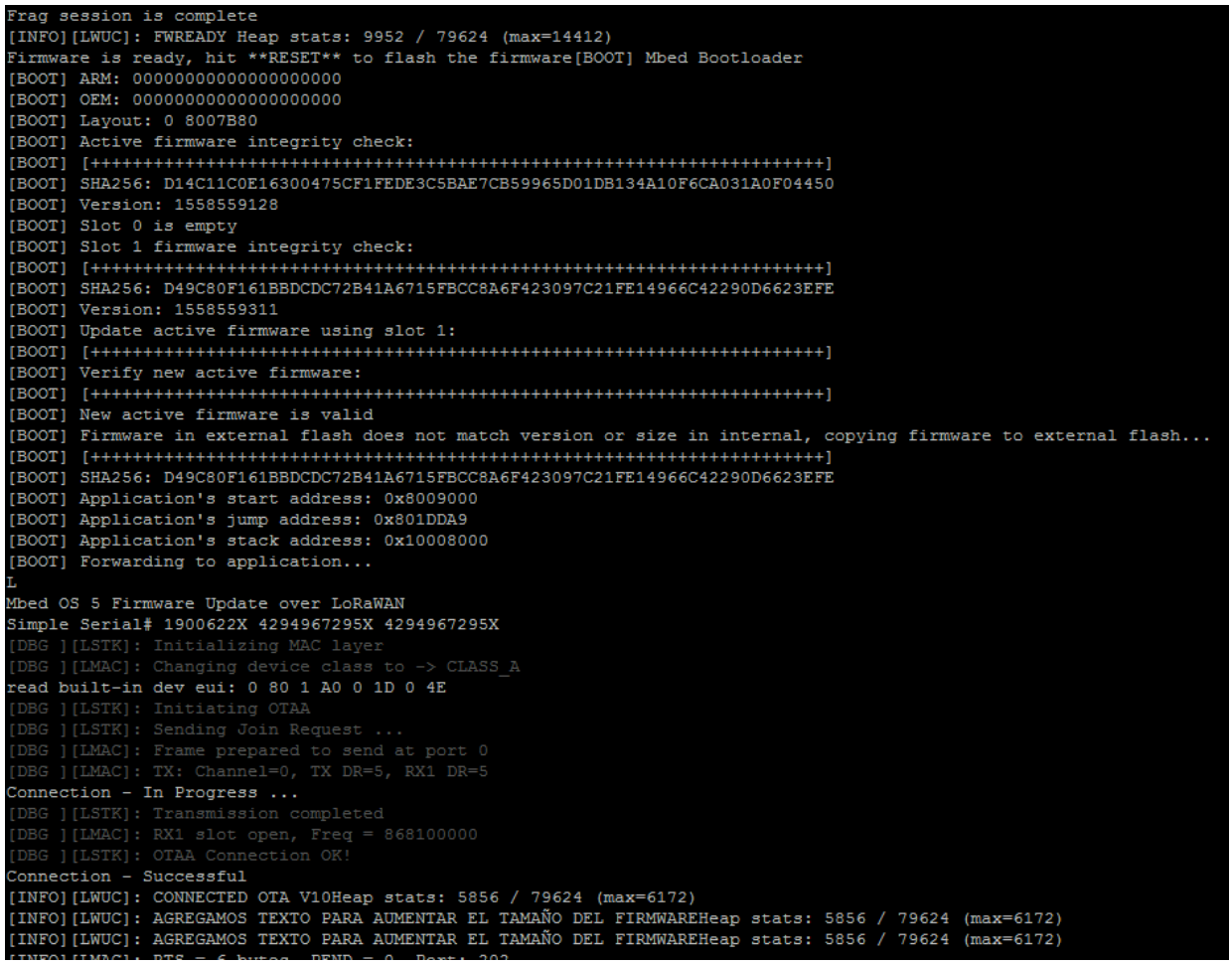

Figura 7.14: *Fin la sesión de fragmentación y reinicio del nodo.*

Con el objetivo de mostrar toda la información disponible, a continuación se muestra el proceso anterior visto desde la aplicación de consola que se ejecuta en el lado del servidor. En la figura pueden verse cada una de las etapas descritas anteriormente.

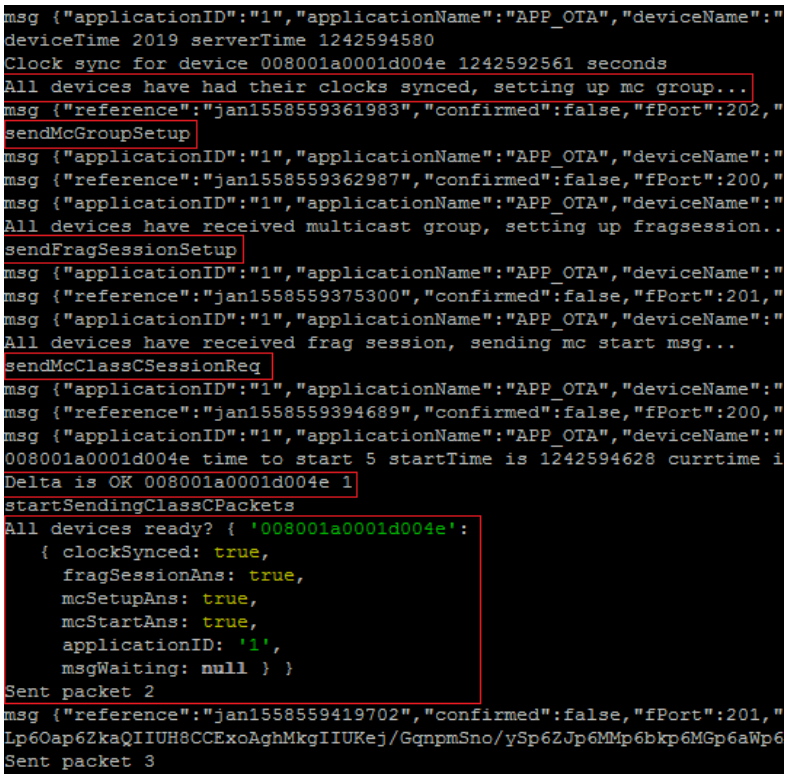

Figura 7.15: *Proceso de actualización visto desde el servidor.*

7.3. Redundancia

La pérdida de fragmentos por parte de los nodos es algo real, una breve interferencia externa en la frecuencia de descarga de datos puede fácilmente provocar la pérdida de algunos fragmentos. No puede ser aceptable que un nodo que ha perdido un pequeño porcentaje de paquetes no pueda actualizarse correctamente y tenga que esperar a que se envíe la actualización completa de nuevo. Durante la sesión de broadcast (en la actualización) los nodos no envían ACK, esto quiere decir que el servidor desconoce que paquetes no han recibido para poder reenviarlos. La redundancia se consigue con la aplicación de los algoritmos denominados LDPC o Low Density Parity Check, estos algoritmos fueron propuestos en el año 1960 por Robert G. Gallager[8] aunque no se empezaron a utilizar hasta los años 90.

Para este proceso se crea una matriz de NxM, donde N corresponde al parámetro de

redundancia y M al tamaño de fragmentación. Con esta matriz se realizan una serie de operaciones matemáticas de forma que la información se dispersa. El algoritmo concreto utilizado en este caso puede verse en el documento que especifica el protocolo para el envió de datos fragmentados[3].

A continuación ponemos a prueba uno de los nodos para ver cómo se recupera en caso de pérdida de paquetes. La pérdida de paquetes se va a forzar desde el servidor de aplicación insertando un bloque condicional de forma que uno o varios fragmentos no se envíen. Para esta demostración se va a utilizar una actualización delta compuesta por 32 fragmentos de 204 bytes y una redundancia de 32 paquetes.

Resultados pruebas de redundancia	
Paquetes perdidos	Paquetes enviados hasta recuperarse
Paquete 2	1 único paquete
Paquete 24	1 único paquete
Paquete 32	1 único paquete
Paquetes 2 y 3	2 paquetes
Paquetes 24 y 30	10 paquetes
Paquetes 2, 24 y 30	10 paquetes
Paquetes 2, 10, 24 y 30	10 paquetes
Paquetes 2, 10, 15, 20, 24 y 30	11 paquetes
Paquetes 2, 10, 15, 18, 20, 22, 24 y 30	11 paquetes
Paquetes 2, 10, 15, 18, 20, 22, 24 y 30	11 paquetes
Paquetes 2, 4, 7, 10, 15, 18, 20, 22, 24 y 30	13 paquetes
Paquetes 2, 4, 7, 10, 15, 18, 20, 22, 24, 25, 26, 27, 28 y 30	14 paquetes

Cuadro 7.2: *Resultados de las pruebas de redundancia.*

En la figura [7.2] pueden observarse los resultados de las pruebas. Es muy importante darse cuenta de que el número de paquetes enviados hasta recuperarse puede estar suponiendo el envío de información de la que ya dispone el nodo, por ejemplo, enviando el paquete número uno de redundancia puede ayudar al nodo a reconstruir información que ya tiene y, aunque este paquete se cuenta en la tabla anterior, no está siendo de utilidad para dicho nodo. El número de los paquetes que no se han enviado se ha escogido de forma aleatoria. Según se estima en la documentación oficial añadir un 10 % de redundancia supone poder

perder aproximadamente un 10 % de los paquetes. Añadiendo más redundancia se envía más información adicional pero esta información solo será utilizada por aquellos nodos que la necesiten. En todos los casos el nodo pudo recuperar todos los fragmentos perdidos y realizar la actualización.

7.4. Seguridad en las actualizaciones

El procedimiento para actualizar los nodos vía LoRaWAN puede ser considerado seguro, todos los datos que se intercambian durante la creación del grupo multicast se envían sobre una sesión LoRaWAN unicast establecida previamente por el nodo, esto significa que todos estos datos van cifrados desde el primer momento.

Por el contrario debemos prestar más atención al propio nodo porque en caso de que uno de ellos desapareciera las claves podrían verse comprometidas, para esto existen diversos niveles de protección de código. En el caso de los microcontroladores STM32 la gran mayoría de ellos, incluyendo el que estamos utilizando, un STM32 de la serie L4, tienen 3 niveles de protección. Con el nivel cero cualquiera puede conectarse al microcontrolador y leer el firmware para obtener las claves, bien usando un JTAG, mediante SWD o en el caso de mantener el bootloader por defecto, utilizando la propia USART. Con el nivel de protección uno todavía es posible conectar con el microcontrolador mediante JTAG/SWD pero no nos permitirá la lectura de la memoria, para poder leer la memoria será preciso borrarla en primer lugar, después pasará a tener el nivel cero de nuevo. El último nivel de protección es el nivel dos, este nivel impide toda conexión con el chip mediante JTAG/SWD, una vez asignado este nivel no se puede volver a cambiar de nivel.

Para garantizar que la red siga siendo segura incluso aunque desapareciera alguno de los nodos deberíamos establecer un nivel alto, establecer un nivel dos no implica que el microcontrolador no pueda actualizarse el mismo, únicamente que no podremos volver a leer la memoria de forma externa. Este tipo de protecciones no tienen ningún efecto si las claves se intercambian con el módulo de radio mediante una comunicación no cifrada SPI,

I2C, USART, etc... Para evitar esto, algunas empresas que fabrican este tipo de módulos de radio pueden incorporar directamente las claves al chip de radio para que de esta forma no se tengan que enviar por ningún bus de comunicaciones.

Para aumentar la seguridad en las actualizaciones todos los nodos llevan hardcodeada una clave pública generada previamente en el servidor, antes de enviar un nuevo firmware a los nodos es imprescindible firmarlo, este proceso es parecido a la firma de un mensaje de correo electrónico o de un documento cualquiera y consiste en utilizar la clave privada para firmar un hash del fichero, posteriormente, una vez el nodo recibe todos los fragmentos del firmware y, utilizando la clave pública, puede comprobar que se trata de un firmware autentico y no de un intento de manipulación por parte de terceros. La clave pública utilizada en los nodos es la misma para todos ellos, que un tercero pueda conseguir de alguna forma dicha clave no implicaría una vulnerabilidad, dicha clave únicamente se utiliza para verificar la firma. El proceso de firma consiste en calcular un hash del fichero a firmar y cifrarlo con nuestra clave privada, entonces aquella persona que tenga nuestra clave pública podrá descifrarlo, una vez descifrado solo tiene que calcular el hash de los datos que ha recibido (sin la parte extra de la firma) y compararlos, si coinciden se puede actualizar el nodo.

7.5. Consumo

Con el objetivo de comprobar si el consumo indicado en el datasheet del módulo de radio por el fabricante coincide con el nuestro se decide realizar una prueba de consumo colocando un multímetro en serie, en este punto debemos concretar algunas cosas, la medición realizada mientras la ventana de recepción estaba abierta es muy fiable porque el consumo se mantenía estable en el tiempo, las mediciones realizadas mientras el nodo estaba transmitiendo son poco fiables, una transmisión de unos pocos bytes dura apenas unos cuantos milisegundos y la tasa de muestreo de un multímetro no es tan baja como para ser capaz de mostrar dicha información, por esto y aunque se medirá tanto en recepción como en transmisión es importante tener en cuenta que los valores de transmisión medidos pueden estar lejos de los

reales.

Consumos RFM95W	
Estado	Consumo
Inactivo	0.07mA
Recibiendo datos	11.2mA
Transmitiendo	18-50mA

Cuadro 7.3: *Resultados de la medición con un multímetro en serie.*

El consumo mientras la ventana de recepción estaba abierta, es decir, mientras el nodo estaba recibiendo el nuevo firmware ha sido de 11.2 miliamperios, este valor se ha mantenido estable durante todo el tiempo que ha estado la ventana de recepción abierta. Para las transmisiones hay que recordar que el tiempo que dura una transmisión viene dado por el parámetro SF, con un SF más bajo el tiempo de transmisión es más corto, en este caso se ha medido con un DR=5 (ver figura [4.12] para más información), equivalente a un SF 7 y un ancho de banda de 125 KHz. Tras varios reinicios consecutivos se han obtenido valores muy variados, siendo el más alto de 50mA y el más bajo de aproximadamente 18mA. El consumo por tener conectado el módulo mientras este está inactivo es de aproximadamente 7μA. Los valores observados son muy cercanos e incluso algunos similares a los proporcionados por el fabricante en el datasheet.

7.6. Actualizando múltiples nodos

Durante el desarrollo de este trabajo se llevaron a cabo múltiples pruebas en el laboratorio para comprobar el comportamiento durante una actualización OTA con múltiples nodos. Los resultados han sido muy favorables, los nodos se han comportado de la forma esperada, es decir, igual que si fueran un único nodo. El procedimiento es el mismo con la excepción de que para pasar a la siguiente fase primero todos los nodos deben confirmar la fase anterior, es decir, si nos encontramos en la fase uno donde los nodos sincronizan el reloj, hasta que todos los nodos no sincronicen el reloj no se crea el grupo multicast. El motivo por el que no es recomendable continuar con la creación del grupo es que existe un parámetro que

establece el tiempo máximo que puede durar una sesión de fragmentación, por lo que si unos pocos nodos se adelantan y otros tardan mucho, los que han empezado antes podrían verse forzados a tener que cerrar la sesión por agotarse el tiempo máximo.

Para poder realizar la prueba con múltiples nodos fue necesario modificar el código fuente. El autor del programa no ha implementado soporte para obtener un ID único en la placa STM32L4, es decir, todos los dispositivos con el mismo firmware tienen el mismo dev-eui. Para solucionar esto y que cada dispositivo tenga un único dev-eui se ha programado una pequeña función que lee el ID único de 96 bits del que disponen los procesadores STM32 y lo utiliza para generar una parte del dev-eui.

7.7. Actualizaciones delta

Las actualizaciones delta o actualizaciones parciales consisten en enviar únicamente los cambios que hay entre el firmware que actualmente tienen los nodos y el nuevo firmware. Para realizar la comparación se utiliza la herramienta jdiff, esta herramienta dados dos ficheros binarios es capaz de crear un parche para transformar el primer fichero en el segundo; para hacer esto en primer lugar se comparan ambos ficheros binarios para agrupar todas las partes que son iguales, después se crea una estructura de control donde se añade en que partes se ha insertado/borrado información, en cuales simplemente se ha modificado y donde se ha mantenido igual. Utilizando la estructura de control los nodos son capaces de parchear su firmware actual.

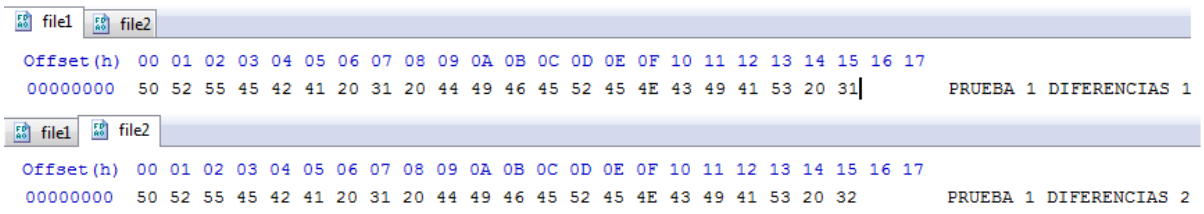

Figura 7.16: *Dos ficheros binarios en el editor HxD.*

En la figura anterior podemos ver dos ficheros binarios cargados en un editor hexadecimal.

Estos ficheros se han creado utilizando bytes con representación ASCII para que puedan verse de una forma más clara los resultados.

A continuación ejecutamos la utilidad jdiff con la opción verbose para que nos muestre un resumen:

```
$ node jdiff-js -v file1 file2  parche
...
Prescanning:
.
Equal     bytes          = 21
Data      bytes written = 1
Overhead  bytes written = 5
```

Podemos observar que 21 bytes son idénticos en ambos ficheros, 1 byte cambia y escribe su valor y 5 bytes son la sobrecarga que añade la estructura de control. Por tanto el tamaño del parche es de 6 bytes, lo podemos confirmar con el comando stat:

```
$ stat parche

File: parche
Size: 6       Blocks: 8       IO Block: 4096   regular file
```

Si este fichero fuera una actualización de firmware, en lugar de tener que enviar 22 bytes solo sería necesario enviar 6 bytes, es decir, un ahorro del 72,73 %.

A continuación se muestra el formato de este parche:

```
$ xxd parche

00000000: a7a3 14a7 a632
```

Observamos cómo se indica con el comando a7a3 la posición 14 (para escribir en la siguiente posición) y con el comando a7 a6 establece nuevo valor.

Veamos ahora un ejemplo más complejo:

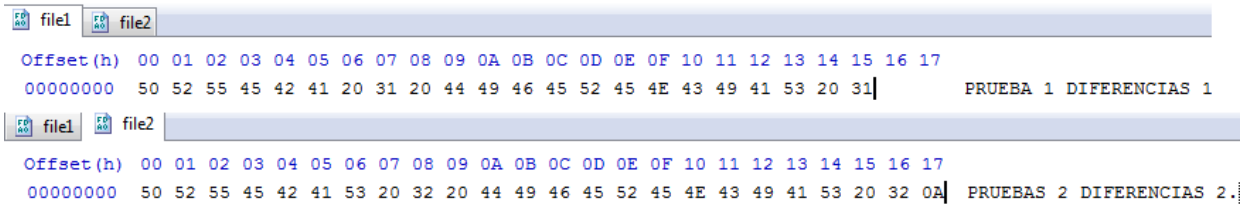

Figura 7.17: *Dos nuevos ficheros binarios en el editor HxD.*

```
$ node jdiff -js -v file1 file2 parche
...
Prescanning:
.
Equal     bytes         = 6
Data      bytes written = 18
Overhead  bytes written = 7

$ xxd parche

00000000: a7a3 05a7 a653 2032 2044 4946 4552 454e  .....S 2 DIFEREN
00000010: 4349 4153 20a7 a532 0a                   CIAS ..2.
```

a7a3 ->05 situarse en la posición 05 (para escribir en la siguiente posición).

a7a6 ->escribir en el fichero 53 2032 2044 4952 454e 4349 4153 20 que corresponde con 'S 2 DIFERENCIAS '

a7a5 ->situarse al final del fichero y escribir 32 0a que corresponde con '2.'

Tamaño del nuevo posible firmware:

```
$ stat file2
```

```
File: file2
Size: 24      Blocks: 8      IO Block: 4096   regular file
```

Tamaño del parche:

```
$ stat parche

File: parche
Size: 25      Blocks: 8      IO Block: 4096   regular file
```

Tal y como podemos ver en el último ejemplo, no siempre es mejor opción una actualización delta frente a una actualización completa. Habitualmente sí lo será debido a que el firmware no se genera de forma aleatoria, si no que se compone de diferentes secciones que se ubican en las mismas zonas del binario. En cualquier caso se debe valorar cada actualización de forma individual.

Veamos ahora que ocurre con un ejemplo real, el tamaño del firmware cargado en el nodo es de 160 KiB, dado que el bootloader ya se encuentra cargado en el nodo no es necesario adjuntarlo en la actualización, esto quiere decir que el fichero de actualización tiene 32KiB menos. La actualización va a consistir en añadir un for con un printf, este printf va a contener un string de 33 bytes.

Tamaño firmware cargado en el nodo (sin bootloader):

```
$ stat old.bin

File: old.bin
Size: 138120       Blocks: 272      IO Block: 4096   regular file
```

Nuevo código:

```
for(int i=0;i<20;i++)
```

```
{
    printf("AGREGAR NUEVOS BYTES AL CODIGO");
}
```

Tamaño nuevo firmware (sin bootloader):

```
$ stat new.bin


File: new.bin
Size: 138168      Blocks: 272      IO Block: 4096   regular file
```

Generar la actualización delta y firmarla:

```
$ lorawan-fota-signing-tool sign-delta --old old.bin --new new.bin
--output-format bin -o signed-diff.bin

$ stat signed-diff.bin
File: signed-diff.bin
Size: 6493       Blocks: 16       IO Block: 4096   regular file
```

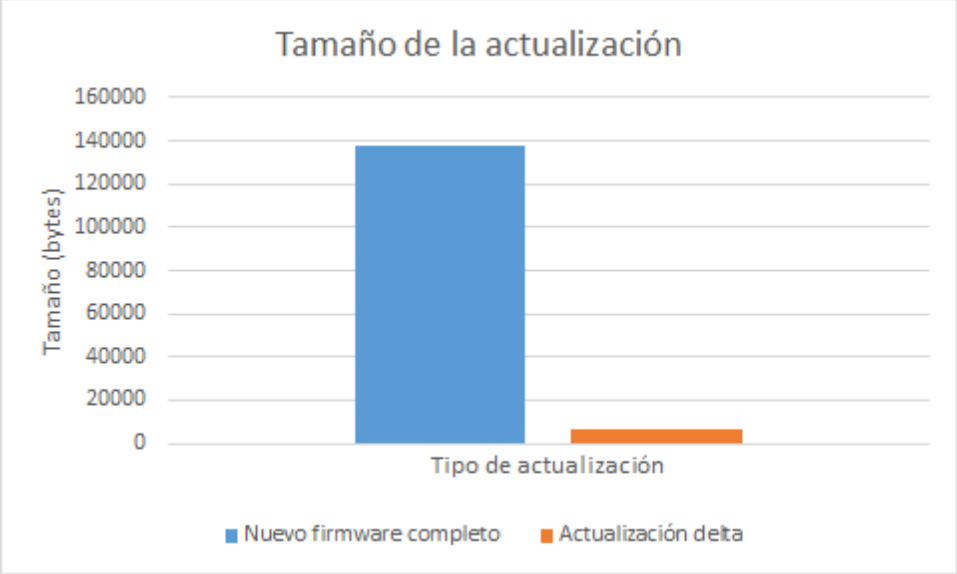

Figura 7.18: *Comparativa entre el tamaño de los dos tipos de actualización.*

En la figura [7.18] podemos observar una diferencia de tamaños muy importante, una reducción en el tamaño de la actualización de un 95,3 % utilizando una actualización delta frente a una completa. Al reducir el tamaño de la actualización se consigue un mejor resultado. Los nodos necesitan menos espacio para almacenar la información, el porcentaje de paquetes perdidos se reduce y el tiempo necesario para la descarga es muy inferior.

7.8. Almacenamiento

En esta implementación de OTA se utilizan las dos memorias disponibles en la placa, la primera es una memoria SPI de 8MB y la segunda es la memoria flash del microcontrolador de 1 MB. El mapa de la memoria flash del microcontrolador es el siguiente:

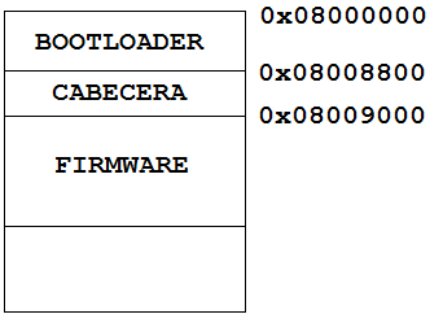

Figura 7.19: *Organización de la memoria flash.*

En la dirección de memoria 0x08000000 se encuentra el bootloader, conviene recordar que el bootloader se va a ejecutar en primer lugar y entre otras cosas gestiona las actualizaciones de firmware del chip. En la dirección 0x08008800 se encuentra la cabecera del firmware, esta cabecera contiene la versión del firmware y la firma del mismo, esta información será de utilidad más adelante. Por último, en la dirección 0x08009000 se encuentra el firmware, la dirección final dependerá del tamaño del mismo.

Por otro lado tenemos la memoria SPI, esta memoria se estructura de la siguiente forma:

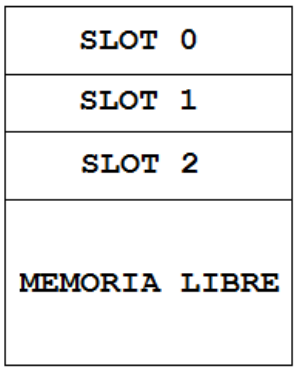

Figura 7.20: *Organización de la memoria SPI.*

- Slot 0: el slot 0 se utiliza para almacenar el firmware en el caso de estar recibiendo una actualización de tipo completa.
- Slot 2: en este slot hay un backup del firmware más reciente cargado en el microcontrolador. Utilizando esta copia podemos recuperar el chip en caso de corrupción en el firmware.
- Slot 1: en el caso de las actualizaciones de tipo delta este slot se utiliza para combinar el firmware actual guardado en el slot 2 con el parche recibido.

Durante el encendido de la placa el bootloader comprueba utilizando la versión almacenada en la cabecera de los slots cuál de los firmwares es el más reciente, en base a esto decide cual debe estar cargado en el microcontrolador y copiado en el slot 2 como respaldo. Para ejecutar el bootloader, nada más terminar una actualización del tipo que sea la placa se reinicia de forma automática, de esta forma se ejecuta el bootloader y realiza todas las actualizaciones en las memorias que correspondan.

7.9. Pruebas fuera del laboratorio

Todas las pruebas anteriores se han hecho en un entorno controlado, el laboratorio, las pruebas que se muestran a continuación han sido realizadas fuera de este, en concreto, en

el campus de Ciudad Universitaria.

En la primera prueba se ha querido comprobar la actualización OTA de dos nodos de forma simultánea, para ello se ha colocado un gateway en el laboratorio y dos nodos en diferentes ubicaciones dentro del campus, la situación de los nodos y el gateway se muestra en la siguiente figura:

Figura 7.21: *Situación de los nodos y gateway durante la prueba fuera del laboratorio.*

La distancia entre el gateway y el nodo más alejado ha sido de 215 metros, la conexión entre el gateway y ambos nodos se ha realizado con un date rate de 5, recibiendo ambos nodos el nuevo firmware de forma correcta. De los 32 paquetes que formaban la actualización delta, ambos nodos recibieron el 100 % de los mismos, no haciendo necesario recurrir a los

paquetes de redundancia.

En la segunda prueba se quiere poner a prueba la tecnología LoRa en un entorno de interferencias debido a los múltiples equipos informáticos y de radio utilizados en las distintas facultades. Para esta prueba se coloca el gateway en la Facultad de Informática y uno de los nodos en la azotea de la Facultad de Ciencias Físicas, a continuación se muestra una figura con la situación exacta:

Figura 7.22: *Situación del gateway y nodo durante la segunda prueba fuera del laboratorio.*

En esta ocasión la distancia entre el nodo y el gateway es de 542 metros, la conexión al igual que en la anterior prueba se ha realizado de forma correcta con un data rate de 5. Para actualizar este nodo se ha utilizado de nuevo la actualización delta frente a la completa; dicha actualización estaba compuesta por 30 fragmentos. El nodo ha sido capaz de recibir todos los fragmentos de forma correcta. No han sido necesarios los paquetes de redundancia.

Como conclusión de las pruebas, la valoración general es muy positiva, todas las actualizaciones fuera del laboratorio, en un entorno que podemos considerar que se asemeja más al de la futura ubicación de los nodos han sido un éxito.

Capítulo 8

Conclusiones

El objetivo marcado para este proyecto de fin de grado había sido conseguir actualizar el firmware de los nodos de una red de sensores mediante una tecnología LPWAN denominada LoRaWAN. Este objetivo no solo se ha logrado sino que además toda la información recopilada y todas las pruebas realizadas tanto en el laboratorio como en un entorno real van a ser de especial interés para el proyecto Solar Node, en el cual además de recopilar información acerca de la radiación solar van a poder tener soporte para actualizaciones OTA.

Se ha podido concluir que en la mayoría de los casos es mucho más conveniente realizar actualizaciones OTA parciales frente a actualizaciones completas. Sin embargo en algunas ocasiones nos podría ocurrir que el tamaño de la actualización parcial sea superior a la actualización completa, en este caso es mejor idea dividir una actualización grande en actualizaciones más pequeñas para evitar fallos de comunicación.

La placa B-L475E-IOT01A[3.2] con procesador STM32L475VGT6, elegida finalmente como placa de pruebas para las actualizaciones OTA, ha sido todo un acierto; la memoria SPI que lleva integrada en placa se ha vuelto indispensable para poder tener siempre una copia de respaldo del firmware, en caso de que se corrompiera alguna parte del firmware, el bootloader podría fácilmente reemplazar el firmware corrupto. También dicha memoria es utilizada para generar el nuevo firmware cuando se ejecuta una actualización delta. Gracias a la memoria integrada evitamos tener que añadir algún tipo de soporte adicional donde guardar esta información, este soporte podría ser un lector de tarjetas SD, un pen-drive

conectado a un puerto OTG mediante un adaptador, etc...

El conjunto de aplicaciones que forman LoRa Server han demostrado durante el desarrollo de este proyecto ser muy robustas y fiables, además son aplicaciones de código libre y con una gran comunidad y documentación detrás.

Como conclusión final de este proyecto, a pesar de que el protocolo elaborado por la asociación LoRa-Alliance para el envío de datos multicast es relativamente nuevo, se ha podido comprobar en multitud de ocasiones durante las pruebas realizadas que este protocolo es muy estable y fiable.

Capítulo 9

Conclusions

The objective set for this end-of-degree project was to update the firmware of the nodes of a sensor network using an LPWAN technology called LoRaWAN. This objective has not only been achieved but also all the information collected and all the tests carried out, both in the laboratory and in a real environment, are going to be of special interest for the Solar Node project, which in addition to gathering information about solar radiation, will be able to have support for OTA updates.

It has been concluded that it is much more convenient to perform partial OTA updates against complete updates. However, in some cases it could happen that the size of the partial update is larger than the complete update, in this case it is better to divide a large update into smaller updates to avoid communication problems.

The board B-L475E- IOT01A [3.2] with processor STM32L475VGT6, was finally chosen as a test board for OTA updates, has been a success. The SPI memory that is integrated in the board has become indispensable to always have a backup copy of the firmware, in case any part of the firmware is corrupted, the bootloader could easily replace the corrupted firmware. This memory is also used to generate the new firmware when a delta update is performed. Thanks to the integrated memory we avoid having to add some additional support to store this information, this support could be an SD card reader, a pen-drive connected to an OTG port through an adapter, etc ...

The set of applications that form LoRa Server have shown during the development of

this project to be very robust and reliable, they are also open source applications with a large community and documentation behind.

As a final conclusion of this project, despite the fact that the protocol developed by the association LoRa -Alliance for the sending of multicast data is relatively new, it has been proven on many occasions during the tests that this protocol is very stable and reliable.

Bibliografía

[1] LoRa Alliance. Lorawan 1.1 regional parameters. https://lora-alliance.org/sites/default/files/2018-04/lorawantm_regional_parameters_v1.1rb_-_final.pdf, 2017.

[2] LoRa Alliance. Lorawan application layer clock synchronization specification v1.0.0. https://lora-alliance.org/sites/default/files/2018-09/application_layer_clock_synchronization_v1.0.0.pdf, 2018.

[3] LoRa Alliance. Lorawan fragmented data block transport specification v1.0.0. https://lora-alliance.org/sites/default/files/2018-09/fragmented_data_block_transport_v1.0.0.pdf, 2018.

[4] LoRa Alliance. Lorawan remote multicast setup specification v1.0.0. https://lora-alliance.org/sites/default/files/2018-09/remote_multicast_setup_v1.0.0.pdf, 2018.

[5] LoRa Alliance. Lorawan™ 1.0.3 specification. https://lora-alliance.org/sites/default/files/2018-06/lorawan1.0.3_final_0.pdf, 2018.

[6] LoRa Alliance. Fuota process summary technical recommendation. https://lora-alliance.org/sites/default/files/2019-04/tr002-fuota_process_summary-v1.0.0.pdf, 2019.

[7] ETSI. European standard; electromagnetic compatibility and radio spectrum matters (erm). https://www.etsi.org/deliver/etsi_en/300200_300299/30022001/02.04.01_40/en_30022001v020401o.pdf, 2012.

[8] Robert G. Gallager. Low-density parity-check codes. https://web.stanford.edu/class/ee388/papers/ldpc.pdf, 1963.

[9] HopeRF. RFM95/96/97/98(W) - Low Power Long Range Transceiver Module V1.0. https://www.hoperf.com/data/upload/portal/20190301/RFM95_96_97_98W.pdf, 2019.

[10] STMicroelectronics. STM32L475xx, Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM. https://www.st.com/resource/en/datasheet/stm32l475vg.pdf, 2019.

[11] Espressif Systems. Esp8266 wroom series. https://www.espressif.com/en/products/hardware/esp-wroom-02/resources/, 2019.

Apéndice A

Mensajes para la creación de un grupo Multicast

A.1. Sincronización del reloj

Mensajes para la sincronización del reloj del nodo:

- Mensaje AppTimeReq, este mensaje únicamente puede ser enviado desde los nodos hacia la aplicación, indica al servidor que el nodo quiere sincronizar su reloj, para ello envía:

Field	DeviceTime	Param
Size (bytes)	4	1

Figura A.1: *Formato mensaje AppTimeReq.*

Param Fields	RFU	AnsRequired	TokenReq
Size (bits)	3bits	1bit	4bits

Figura A.2: *Formato parámetro Param.*

  - DeviceTime corresponde con el número de segundos transcurridos desde las 00:00:00 del 6 de Enero de 1980 módulo 2^{32}
  - Param que contiene:

- AnsRequired, si está a uno el servidor debe enviar un mensaje AppTimeReqAns incluso aunque el reloj ya este correctamente sincronizado, si está a cero el servidor no debe responder si el reloj está correctamente sincronizado.
- TokenReq, es un contador módulo 16, aumenta en uno cada vez que el nodo recibe un mensaje AppTimeReqAns, este valor debe ser el mismo en la respuesta del servidor o el nodo ignorará la respuesta.

- Mensaje ForceDeviceResyncReq, mensaje enviado desde el servidor hacia un nodo concreto, indica a un nodo que debe iniciar el proceso para sincronizar su reloj, es decir, que debe enviar el mensaje AppTimeReq.

A.2. Configuración del grupo multicast

El mensaje McGroupStatusReq permite crear un grupo multicast, este mensaje contiene los siguientes campos:

Field	McGroupIDHeader	McAddr	McKey_encrypted	minMcFCount	maxMcFCount
Size (bytes)	1	4	16	4	4

Figura A.3: *Formato mensaje McGroupStatusReq.*

McGroupIDHeader Fields	RFU	McGroupID
Size (bits)	6bits	2bits

Figura A.4: *Formato parámetro McGroupIDHeader.*

- McGroupStatusReq se descompone en dos parametros, uno reservado para el futuro y McGroupID de 2 bits, indica el id de la sesión multicast, los nodos pueden estar hasta en 4 sesiones multicast al mismo tiempo.

- McAddr es la dirección lógica del grupo multicast.

- McKey_encrypted es la clave que van a utilizar los nodos para calcular las claves McAppSKey y McNetSKey, con estas claves se van a cifrar los datos de la sesión multicast.
- MinMcFCount Indica el número siguiente de trama que enviará el servidor por el grupo multicast
- MaxMcFCount Máximo número de paquetes que se van a enviar por el grupo multicast.

Este mensaje requiere confirmación por parte del nodo, el nodo responde con un mensaje McGroupStatusAns que tiene el siguiente formato:

Field	McGroupIDHeader
Size (bytes)	1

Figura A.5: *Formato parámetro McGroupStatusAns.*

McGroupIDHeader Fields	RFU	IDerror	McGroupID
Size (bits)	5	1	2

Figura A.6: *Formato parámetro McGroupIDHeader.*

- McGroupIDHeader
  - IDerror indica que el dispositivo no soporta el contexto del grupo, por ejemplo, solo tiene soporte para una sesión simultánea de multicast.
  - McGroupID id del grupo multicast.

A.3. Fragmentación de paquetes

El mensaje FragSessionSetupReq crea una sesión de fragmentación. Contiene los siguientes campos:

Field	FragSession	NbFrag	FragSize	Control	Padding	Descriptor
Size (bytes)	1	2	1	1	1	4

Figura A.7: *Formato mensaje FragSessionSetupReq.*

- FragSession, que se descompone en:
  - FragIndex identificador de la sesión, hasta 4 sesiones.
  - McGroupBitMask utilizado para asociar sesiones de fragmentación a grupos multicast, indica que grupos de multicast pueden recibir datos de esta sesión de fragmentación, un valor 0000 indica solo datos unicast, un valor 0001 indica sesión de multicast 0 y un valor 1111 indica que pueden recibir datos de esta sesión en los 4 grupos multicast más unicast.

FragSession Fields	RFU	FragIndex	McGroupBitMask
Size (bits)	2bits	2bits	4bits

Figura A.8: *Formato parámetro FragSession.*

- NbFrag, número de fragmentos que vamos a enviar.
- FragSize, tamaño de los fragmentos.
- Control es una estructura de control que tiene los siguientes campos:
  - FragAlgo, indica el algoritmo utilizado para la fragmentación de los paquetes.
  - BlockAckDelay, máximo de tiempo para que los nodos confirmen un mensaje que requiera confirmación recibido por un grupo multicast. El nodo generará un valor aleatorio utilizando nuestro máximo tiempo para evitar las colisiones que tendríamos si todos los nodos confirmaran el mensaje al mismo tiempo.

Control Fields	RFU	FragAlgo	BlockAckDelay
Size (bits)	2bits	3bits	3bits

Figura A.9: *Formato parámetro Control.*

- Padding, número de bytes que se descartan del último fragmento recibido. Por ejemplo, para un tamaño de payload de 204 bytes, si una actualización solo requiere de 128 bytes tendríamos un padding de 76, 204 - 76 = 128 bytes de datos.
- Descriptor indica el tipo de datos que está recibiendo el nodo, por ejemplo, una actualización OTA.

Este mensaje requiere una confirmación por parte de los nodos, el mensaje de confirmación contiene los siguientes campos:

Bits	7:6	5:4	3	2	1	0
Status bits	FragIndex	RFU	Wrong Descriptor	FragSession index not supported	Not enough Memory	Encoding unsupported

Figura A.10: *Formato mensaje FragSessionSetupAns.*

Si alguno de los bits [0:3] está a uno entonces el nodo ha rechazado la sesión de fragmentación.

A.4. Ventana de recepción

Mensaje McClassCSessionReq para abrir una ventana de recepción de la clase C. Contiene los siguientes campos:

Field	McGroupIDHeader	Session Time	SessionTimeOut	DLFrequ	DR
Size (bytes)	1	4	1	3	1

Figura A.11: *Formato mensaje McClassCSessionReq.*

- McGroupIDHeader que contiene una estructura con los siguientes parámetros:

McGroupIDHeader Fields	RFU	McGroupID
Size (bits)	6bits	2bits

Figura A.12: *Formato parámetro McGroupIDHeader.*

McGroupID que corresponde con el identificador del grupo multicast.

- SessionTime es el tiempo en segundos para abrir la ventana de recepción.
- SessionTimeOut que contiene una estructura con los siguientes parámetros:

SessionTimeOut Fields	RFU	TimeOut
Size (bits)	4bits	4bits

Figura A.13: *Formato parámetro SessionTimeOut.*

Donde TimeOut es el máximo tiempo que va a estar la ventana de recepción abierta antes de volver a la clase A para ahorrar batería.

- DLFrequ Frecuencia en la que van a estar escuchando los nodos, se utiliza 869.525 MHz por tener un duty cycle del 10 %.
- DR indica el factor de propagación y el ancho de banda.

Este mensaje también requiere que los nodos confirmen su correcta recepción mediante el envío de un mensaje McClassCSessionAns que contiene los siguientes campos:

Field	Status&McGroupID	(cond)TimeToStart
Size (bytes)	1	3

Figura A.14: *Formato mensaje McClassCSessionAns.*

La estructura Status&McGroupID contiene los siguientes parametros:

- McGroupUndefined, a uno si el grupo multicast del campo McGroupID no existe.
- FreqEerror, a uno si el nodo no soporta la frecuencia.
- DRError, a uno si el nodo no soporta dicho data rate.
- McGroupID Identificador del grupo multicast.

Status&McGroupID Fields	RFU	McGroupUndefined	FreqError	DRError	McGroupID
Size (bits)	3bits	1bit	1bit	1bit	2bits

Figura A.15: *Formato parámetro Status&McGroupID.*

El campo (cond)TimeToStart se utiliza para comprobar que el reloj del nodo sigue correctamente sincronizado. El nodo indica en este campo los segundos restantes para abrir la ventana de recepción y de esta forma el servidor puede comprobar que está correctamente sincronizado.