UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una plataforma HW/SW de Smart Parking basada en tecnología LoRa/LoRaWAN

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: D. Luis González Álvarez

Tutores: D. Valentín De Armas Sosa

D. Félix B. Tobajas Guerrero

Fecha: Junio de 2018

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una plataforma HW/SW de Smart Parking basada en tecnología LoRa/LoRaWAN

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: _____

Vocal **Secretario/a**

Fdo.: _____ Fdo.: _____

Fecha: Junio de 2018

Agradecimientos

A toda mi familia y amigos, por el apoyo incondicional.

A todos mis educadores, por formarme como persona y como estudiante.

A mis tutores Félix y Valentín, por creer en mí y siempre estar presentes.

A todos y cada uno de los compañeros que he tenido durante la carrera.

A ti.

¡Muchísimas gracias a todos!

Índice

Índice de Figuras

Índice de Tablas

Pliego de condiciones

Presupuesto

Tabla de Acrónimos

ABP	*Activation By Personalization*
ACK	*Acknowledgement*
ADC	*Analogue to Digital Converter*
ADR	*Adaptative Data Rate*
AES	*Advanced Encryption Standard*
BLE	*Bluetooth Low Energy*
Bps	*Bits por segundo*
CD-ROM	*Compact Disc Read-Only Memory*
CO	*Monóxido de Carbono*
CO_2	*Dióxido de Carbono*
COITT	*Colegio Oficial de Ingenieros Técnicos de Telecomunicación*
CRC	*Código de Redundancia Cíclica*
CSS	*Chirp Spread Spectrum*
dB	*Decibelio*
dBm	*Decibelio milivatio*
DCM	*Data Confidence Metric*
DGT	*Dirección General de Tráfico*
EITE	*Escuela de Ingeniería de Telecomunicación y Electrónica*
ETSI	*European Telecommunications Standards Institute*
FEC	*Forward Error Correction*
FSK	*Frequency Shift Keying*
GPIO	*General-Purpose Input-Output*

GPS	*Global Positioning System*
HTTP	*Hypertext Transfer Protocol*
HW	*Hardware*
I²C	*Inter-Integrated Circuit*
I²S	*Inter-IC Sound*
ICSP	*In Circuit Serial Programming*
ID	*Identificador*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
IGIC	*Impuesto General Indirecto Canario*
IoT	*Internet of Things*
IP	*Internet Protocol*
IR	*Infrared Radiation*
ISM	*Industrial, Scientific and Medical bands*
ISO	*International Organization for Standardization*
JDK	*Java Development Kit*
JSON	*JavaScript Object Notation*
Kbps	*Kilo bits por segundo*
KB	*Kilo Byte*
KHz	*Kilo Hercios*
Km	*Kilómetro*
kWh	*Kilovatio/hora*
LED	*Light-Emitting Diode*
LiPo	*Polímero de Litio*
LoRa	*Long Range*

LoRaWAN	*Long Range Wide Area Network*
LPWAN	*Low Power Wide Area Network*
LSB	*Least Significant Bit*
LTE	*Long Term Evolution*
mA	*Miliamperios*
MAC	*Medium Access Control*
mAh	*Miliamperios/hora*
mGa	*MiliGauss*
MHz	*Mega Hercios*
mm	*Milímetro*
MSB	*Most Significant Bit*
mW	*Milivatio*
NB-IoT	*Narrow Band-Internet of Things*
NFC	*Near Field Communication*
NO_2	*Dióxido de Nitrógeno*
O_3	*Ozono*
OTAA	*Over The Air Activation*
ONU	*Organización de Naciones Unidas*
OSI	*Open Systems Interconnection*
P2P	*Peer to Peer*
PC	*Personal Computer*
PDF	*Portable Document Format*
PM	*Particulate Matter*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*

REPL *Read-Eval-Print-Loop*

RF *Radio Frequency*

RFID *Radio Frequency Identification*

RGB *Red-Green-Blue*

RTC *Real Time Clock*

SCL *Serial Clock*

SD *Secure Digital*

SDA *Serial Data*

SF *Spreading Factor*

SO_2 *Dióxido de Azufre*

SPI *Serial Peripheral Interface*

SSID *Service Set Identifier*

SSM *Spread Spectrum Modulation*

SW *Software*

TCP *Transmission Control Protocol*

TDoA *Time Difference of Arrival*

TFG *Trabajo Fin de Grado*

TIC *Tecnología de la Información y las Comunicaciones*

TTN *The Things Network*

TV *Television*

µA *Microamperio*

UART *Universal Asynchronous Receiver/Transmitter*

UDP *User Datagram Protocol*

ULPGC *Universidad de Las Palmas de Gran Canaria*

URL *Uniform Resource Locator*

USB	*Universal Serial Bus*
WiFi	*Wireless Fidelity*
WUP	*World Urbanization Prospects*
XML	*eXtensible Markup Language*

Memoria

Capítulo 1. Introducción

En los últimos años el éxodo de los habitantes rurales hacia las grandes ciudades está propiciando su masificación. Tal es así que hace 11 años, en 2007, el número de habitantes urbanos sobrepasó por primera vez en la historia a los habitantes de zonas rurales. Ya en 2015 las estadísticas indicaban que el 50% de la población mundial residía en ciudades [1], y estos datos no han hecho más que aumentar, previendo la Organización de las Naciones Unidas (ONU) que un 66% de población mundial estará ubicada en zonas urbanas para el año 2050 [2]. Por lo tanto, en estos núcleos urbanos se encuentran serios problemas de saturación, consumo de energía y movilidad. Esta es la razón por la que se reciben con gran aceptación las diferentes soluciones tecnológicas que favorecen la agilidad, la comodidad, y el bienestar de los habitantes en las principales metrópolis.

El concepto de *Smart City* introduce la relación entre tecnología y calidad de vida, pues partiendo de soluciones de carácter tecnológico se pretende desarrollar el entorno de las ciudades para lograr un beneficio común en cuanto a sostenibilidad, accesibilidad y bienestar social. Varios estudios afirman que para el año 2020 se llegará a tener entre 25.000 y 50.000 millones de dispositivos conectados a Internet [3], [4] , a lo que se debe sumar el número de teléfonos inteligentes existentes, ya que estos permiten en gran medida conectar a las personas y hacerlas protagonistas de la gran revolución tecnológica que precisan las *Smart Cities*.

Dentro del concepto de *Smart Cities* existen multitud de campos de estudio, uno para cada área de mejora en la calidad de vida de las personas. A modo de ejemplo se enumeran, entre otros muchos, el desarrollo en la seguridad, en la educación, en el ocio, en la sostenibilidad, en la sanidad, en la domótica, en la monitorización de recursos, o en la movilidad. En referencia a este último campo se incluye el ámbito de estudio de este Trabajo Fin de Grado (TFG), el estacionamiento de vehículos, o como se define desde la nomenclatura que concierne a todos estos avances tecnológicos, *Smart Parking* o aparcamiento inteligente.

Este ámbito de las *Smart Cities* es cada vez más demandado, ya que actualmente un 30% del volumen del tráfico del centro de las ciudades está relacionado con la búsqueda de aparcamiento [5], dejando a los conductores durante una media de 20 minutos en la ardua tarea de encontrar un espacio disponible para estacionar su vehículo. En definitiva, con soluciones de *Smart Parking* se pretende simplificar esta tarea mediante diferentes procesos tecnológicos garantizando la sostenibilidad, el bienestar medioambiental, la eficiencia, y por supuesto el ahorro de tiempo de las personas, evitando de esta manera situaciones de malestar que puedan desencadenar conductas negativas y conflictivas en la conducción.

Un tipo de solución que se emplea habitualmente en las *Smart Cities* es el basado en *Internet of Things* (IoT). Este concepto se fundamenta en la interconexión de cualquier producto con otro, haciendo uso de Internet, y creando de esta manera un entorno inteligente y versátil. Para el caso de estudio de este Trabajo Fin de Grado se busca además que la solución IoT adoptada sea de bajo consumo, por lo que se puede acotar el desarrollo de la solución al uso de una serie de tecnologías que permitan una comunicación adecuada a las necesidades de la aplicación sin hacer uso de cantidades elevadas de energía. Dentro de este tipo de tecnologías se tienen varias opciones, como pueden ser *Bluetooth* en su variante *Bluetooth Low Energy* (BLE), *Narrow Band-Internet of Things* (NB-IoT), *Sigfox* o *Long Range*/*Long Range Wide Area Network* (LoRa/LoRaWAN). Dependiendo de la aplicación que se desee desarrollar se procede a la selección de un tipo de tecnología u otro, si bien para las especificaciones de este Trabajo Fin de Grado se requiere bajo consumo, largo alcance, y optimización de los costes de la solución. Bajo

este criterio se propone LoRa/LoRaWAN como tecnología de comunicación para utilizar en la solución que se va a desarrollar.

La especificación LoRa/LoRaWAN permite conectar nodos distanciados entre sí hasta 12 Km y sin un consumo elevado de potencia [6], lo que, a su vez, permite alargar la vida útil de la batería del dispositivo asociado a cada nodo de la red. LoRa representa la conexión física entre los nodos de la red y LoRaWAN es la propia red que interconecta y gestiona dichos nodos, teniendo en cuenta diferentes parámetros propios de una red de comunicación.

1.1 Objetivos

El objetivo principal de este TFG es el desarrollo de una plataforma Hardware/Software (HW/SW) para aplicaciones de *Smart Parking* que utilice LoRa/LoRaWAN como tecnología de comunicación a partir del uso inicial de dos dispositivos diferentes, Seeeduino LoRaWAN y LoPy, así como de un sensor magnetómetro para la detección de vehículos estacionados. Con esta plataforma se pretende proponer una solución basada en IoT para el ámbito de *Smart Parking* dentro del marco de las *Smart Cities*. Así, se desea monitorizar las plazas de estacionamiento para, de esta manera, tener información disponible y accesible sobre su disponibilidad. El objetivo social que se persigue en el presente Trabajo Fin de Grado es el de agilizar el tráfico, garantizar la sostenibilidad y el bienestar medioambiental, y disminuir la pérdida de tiempo de los conductores en la difícil tarea de buscar aparcamiento.

Dentro de los objetivos parciales de este TFG se incluye el estudio y aplicación de la tecnología LoRa/LoRaWAN como red de comunicación *Low Power Wide Area Network* (LPWAN), tratando de optimizar al máximo las restricciones de consumo de potencia y asegurando una comunicación fiable y segura. Para cumplir con estas exigencias, se plantea inicialmente el uso de dos dispositivos que integran módulos LoRa/LoRaWAN, siendo estos dispositivos de distintos fabricantes (Seeedstudio y Pycom) y por tanto aportando diferentes respuestas en cuanto a parámetros para la misma aplicación. De esta manera, se realizará la caracterización de ambos dispositivos, así como la

comparación entre ellos buscando cumplir de la mejor manera posible con las exigencias de este tipo de aplicaciones.

La topología que sigue la tecnología de comunicación LoRa/LoRaWAN es en estrella. Por lo tanto, se tienen conexiones individuales entre cada nodo final y su nodo *gateway* asignado, y la agrupación de todas estas conexiones individuales es lo que conforma la red de comunicación. Para su configuración, se considera como objetivo de este TFG la programación de los nodos finales y del nodo *gateway*. Además, en relación con la configuración y programación de los nodos *gateway*, se desarrollará una aplicación que interactúe con la información presente en dichos nodos, integrada inicialmente en *The Things Network* (TTN).

Otro objetivo parcial es la caracterización de los sensores que proveen al servicio de información. Se deben conocer las características de los sensores para poder integrarlos en la solución.

A continuación, se resumen los objetivos de forma estructurada:

- Estudio y aplicación de la tecnología LoRa/LoRaWAN como red de comunicación LPWAN.
- Caracterización de los sensores.
- Caracterización de los dispositivos Seeeduino LoRaWAN / LoPy y comparación entre ellos.
- Programación de la funcionalidad de los nodos finales.
- Programación de la funcionalidad del nodo *gateway*.
- Integración de la aplicación en TTN.

1.2 Peticionario

Actúa como peticionario del presente Trabajo Fin de Grado (TFG) la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC) como requisito indispensable para la obtención del título de Graduado en

Ingeniería en Tecnologías de la Telecomunicación, tras haber superado con éxito las asignaturas especificadas en el Plan de Estudios.

1.3 Estructura del documento

El presente documento está dividido en cuatro partes diferenciadas: Memoria, Pliego de condiciones, Presupuesto y Anexo. A su vez, la Memoria se ha estructurado en diez capítulos, además de las referencias bibliográficas, tal como se describe a continuación. La memoria está estructurada de la siguiente forma:

- **Capítulo 1.** ***Introducción.*** En este capítulo se presentan los motivos que han dado lugar al planteamiento de este Trabajo Fin de Grado (TFG), se explican los conceptos básicos que se tratarán en este y se presentan, además, los objetivos, el peticionario y la estructura del documento.

- **Capítulo 2.** ***Antecedentes.*** En este capítulo se realiza un estudio completo sobre la temática del presente TFG. Se comienza por la razón de ser de este tipo de soluciones IoT dentro del marco de las *Smart Cities*, para posteriormente comentar los antecedentes y diferentes aportaciones que se han desarrollado hasta la fecha.

- **Capítulo 3.** ***LoRa/LoRaWAN.*** En este capítulo se expone el estudio realizado sobre la tecnología de comunicación LoRa/LoRaWAN. Esta tecnología de comunicación es la que se utiliza en la solución de *Smart Parking* desarrollada en este TFG, por lo que en este mismo capítulo se explica la arquitectura de dicha solución, así como cada uno de los bloques que la forman.

- **Capítulo 4.** ***Sensores del nodo final.*** En este capítulo se comentan todos los aspectos relativos al desarrollo de los *Sensores del nodo final*. Se caracterizan los sensores utilizados y su integración en la solución completa a partir de las librerías desarrolladas.

- **Capítulo 5. *Nodo final.*** En este capítulo se presenta el desarrollo del bloque *Nodo final* de la arquitectura propuesta. Se analiza en profundidad el diseño y el desarrollo HW/SW de este elemento. Se caracterizan además los dispositivos utilizados como nodos finales.

- **Capítulo 6. *Nanogateway*.** En este capítulo se presenta el desarrollo del bloque *Nanogateway* de la arquitectura propuesta. Se analiza en profundidad el tipo de *gateway* usado, incluyendo su caracterización y estudio del *firmware* utilizado.

- **Capítulo 7. *The Things Network e integración HTTP.*** En este capítulo se presenta el desarrollo de los bloques *The Things Network* e *Integración HTTP* de la arquitectura propuesta *The Things Network* e *Integración HTTP*. Se documenta el proceso realizado en el desarrollo de la aplicación en el servidor y en la integración *Hypertext Transfer Protocol* (HTTP).

- **Capítulo 8. *Aplicación Java.*** En este capítulo se aborda el desarrollo del bloque *Aplicación Java* de la arquitectura propuesta denominado. Se documenta el desarrollo realizado en la aplicación *Java*. Se explica en detalle cada uno de los archivos que conforman la aplicación *Java*, así como el procedimiento previo a su desarrollo.

- **Capítulo 9. *Validación de la plataforma HW/SW.*** En este capítulo se valida la plataforma HW/SW desarrollada en el presente TFG. Para ello se documentan las pruebas de campo realizadas. También se validan en esta sección el resto de las funcionalidades desarrolladas.

- **Capítulo 10. *Conclusiones y líneas futuras.*** En este capítulo se resumen las conclusiones y líneas futuras que se valoran a partir de la experiencia obtenida a partir de la realización del TFG.

La segunda parte del documento consiste en el Pliego de condiciones, mientras que la tercera parte se corresponde con el Presupuesto, y por último un Anexo que contiene información adicional sobre la documentación aportada.

Capítulo 2. Antecedentes

2.1 Origen de las *Smart Cities*

En los últimos años el éxodo de los habitantes rurales hacia las grandes ciudades está propiciando su masificación. En el año 2015 las estadísticas indicaban que más del 50% de la población mundial residía en ciudades, y estos datos no han hecho más que aumentar, previendo la Organización de las Naciones Unidas (ONU) que un 66% de población mundial estará ubicada en zonas urbanas para el año 2050 [2], lo que representa un crecimiento muy significativo teniendo en cuenta que en el año 1950 este porcentaje era aproximadamente la mitad, un 30% [7]. Precisamente, la Organización de las Naciones Unidas publica periódicamente un estudio sobre el estado de la población a nivel mundial denominado *World Urbanization Prospects* (WUP) [7]. En este informe se recogen datos y estadísticas de interés, relativas al estado de la población mundial. Así, en el informe WUP de 2014 se indica por ejemplo cuáles son las regiones o continentes más urbanizados. Como se muestra en la Figura 2.1, en primer lugar, se tiene a Norte América (81.5% de población urbana), seguido del resto del continente americano y el Caribe (79.5%), mientras que, en tercer lugar se sitúa Europa (73.4%), y en contraste con estas tres regiones, se tiene que Asia (47.5%) y África (40.0%) se encuentran por debajo de la mitad, lo que significa que tienen más población rural que urbana. No obstante, estas regiones se están urbanizando a mayor velocidad que otras, por lo que se estima que en el año 2050 el porcentaje de urbanismo en Asia y África se incremente hasta alcanzar un

64% en el caso de Asia, y un 56% en el continente africano. En lo que respecta al continente oceánico, desde 1970 hasta la actualidad no se ha detectado un cambio tan significativo como en los otros continentes, y tampoco se espera un gran crecimiento para el año 2050. Sin embargo, el porcentaje de población urbana supera la mitad, y está en un 70.8%, casi al nivel de Europa.

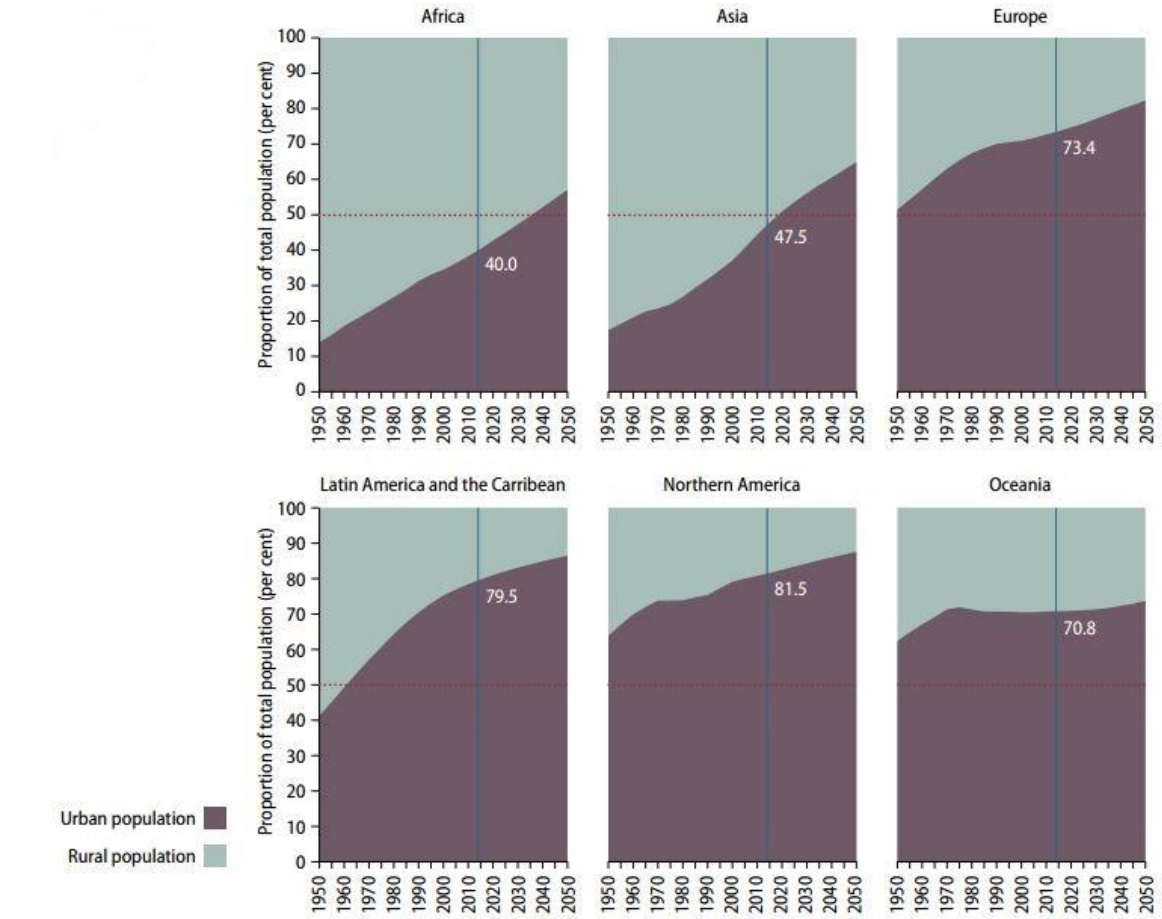


Figura 2.1: Estudio de población rural y urbana en las grandes regiones.

En general, a nivel mundial, las cifras oficiales establecen que en el año 1950 la población urbana era de 746 millones de personas, y actualmente es de 3.9 mil millones, y para el año 2050 se espera que esta cifra aumente en 2.5 mil millones de personas.

A nivel de países, el estudio también muestra datos de interés, como por ejemplo el porcentaje de población urbana y el nivel de población mundial, representado en la Figura 2.2. Así, la lista de países con mayor población urbana está encabezada por Bélgica, con un 98% de población urbana sobre la población total. Otros países con porcentajes muy elevados son Japón (93%), Argentina (92%) y Holanda (90%). A partir de estos datos

se entiende que existen varios países que acumulan la mayor parte de su población en ciudades.

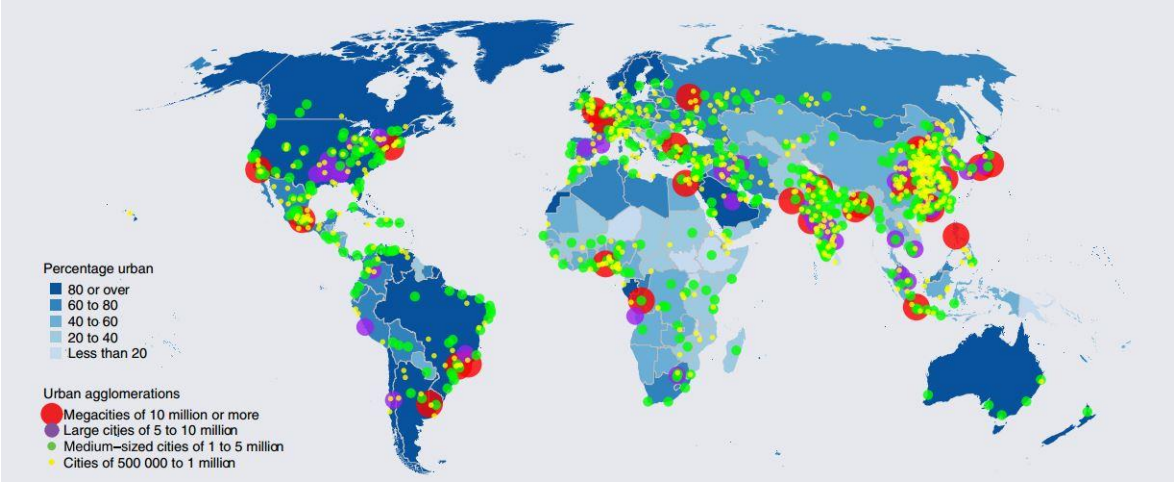


Figura 2.2: Porcentaje de urbanización y nivel de población de las ciudades a nivel mundial.

Con respecto a las ciudades, en 1990 había 10 ciudades con más de 10 millones de habitantes, o lo que es lo mismo, 10 megaciudades. En estas diez grandes ciudades residían un total de 153 millones de personas, menos del 7% de la población urbana mundial. Hoy en día, el número de megaciudades casi se ha triplicado, contabilizándose actualmente 28 megaciudades, habitadas por un total de 453 millones de personas, lo que ha incrementado el porcentaje del 7% de la población urbana mundial hasta un 12%.

Dentro de la lista de las 28 megaciudades, se encuentran en los primeros puestos ciudades como Tokio (Japón) con 38 millones de habitantes, Delhi (India) con 25 millones, Shanghái (China) con 23 millones, y Ciudad de México (México), Bombay (India) y São Pablo (Brasil) con 21 millones de habitantes cada una. Son las mayores ciudades en la actualidad, y se estima que lo sigan siendo en el futuro, algunas de ellas incrementando aún más su número de habitantes.

A partir de todos estos datos se puede deducir que el crecimiento de las ciudades es notable, y no solo en su número de habitantes, sino que cada vez surgen más megaciudades. Esta información se refleja en la gráfica de la Figura 2.3 extraída del informe WUP 2014 [7], en la que muestra claramente la aparición de nuevas megaciudades, y cómo han crecido en su número de habitantes. Además, el número de habitantes no se ha incrementado solamente en las megaciudades, sino que, en otros

tipos de espacios urbanos, como en el caso de pequeñas áreas urbanas (menos de 500.000 habitantes), ciudades pequeñas (500.000 a 1 millón de habitantes), medianas (1 - 5 millones de habitantes) y grandes (5 y 10 millones de habitantes), también se ha visto cómo su población ha aumentado considerablemente.

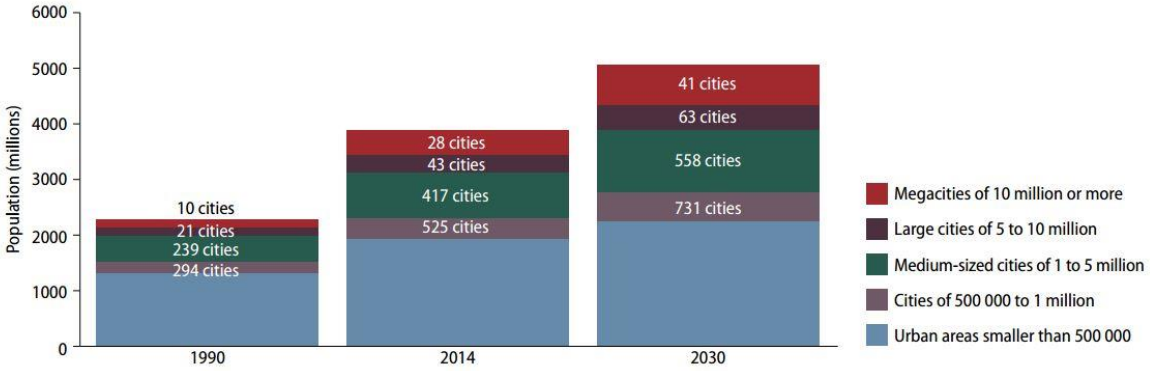


Figura 2.3: Nivel de población en los diferentes tipos de ciudades.

Ante este escenario, se puede afirmar que en estos núcleos urbanos se encuentran serios problemas de saturación, consumo de energía y movilidad. En consecuencia, una de las conclusiones que se extraen del informe WUP 2014 es que a medida que el mundo continúa urbanizándose, los desafíos asociados al desarrollo sostenible son cada vez más relevantes, sobre todo en los países de ingresos medios-bajos donde la velocidad de urbanización es mayor.

Está claro que la disponibilidad de infraestructuras, de educación, de rápido acceso al trabajo y de seguridad, aporta calidad de vida a las personas, y la ausencia de estos recursos, generan grupos humanos con baja o mala calidad de vida. Además, el crecimiento desordenado de las ciudades genera un proceso de desigualdad entre diferentes sectores sociales, pues la saturación de las zonas urbanas beneficia a unos y perjudica a otros. Todos estos problemas pueden derivar en riesgos para la salud, ya que se incrementan los niveles de estrés en las personas y provocan enfermedades, tanto físicas como mentales, que por supuesto, influyen también en la comunidad social. [8]

Esta es la razón por la que se reciben con gran aceptación soluciones tecnológicas que favorezcan la agilidad, la comodidad y el bienestar de los habitantes en grandes metrópolis.

Bajo estas circunstancias surge el concepto de *Smart City*, o Ciudad Inteligente que se presenta como una solución o alternativa al modo de gestión y control de diferentes sistemas determinantes dentro de las ciudades. Por lo tanto, es necesario transformar las ciudades tradicionales en *Smart Cities*, y con la evolución de la tecnología que se experimenta en la actualidad, esta transformación es cada vez más viable.

2.2 *Smart Cities*

Las grandes ciudades y las áreas metropolitanas se pueden entender como sistemas complejos con conexiones entre sus diferentes sectores e individuos. Por esto la planificación urbana y el desarrollo de mecanismos de decisión dinámicos resultan cada vez más importantes. Para realizar una buena planificación y desarrollar los mecanismos de decisión correctos es necesario conocer lo que sucede en las ciudades y tener capacidad para actuar sobre ellas. [9]

Las ciudades son sistemas complejos que se gestionan en tiempo real y que generan una gran cantidad de datos. A partir de estos datos se debe hacer un uso inteligente de las Tecnologías de la Información y las Comunicaciones (TIC) con el fin de afrontar los retos que presentan las ciudades del presente y del futuro.

La idea que subyace en el concepto de *Smart Cities* consiste en aprovechar el potencial de los avances tecnológicos para conseguir mejores resultados en diferentes ámbitos de la gestión de una ciudad. Con esta idea, lo que se pretende es ahorrar costes, un objetivo que solo se puede conseguir priorizando la eficiencia en todos los sistemas que componen las *Smart Cities*.

Si bien no existe una definición común de *Smart City*, ya que puede resultar un término un poco ambiguo, como definición simple se puede decir que se trata de: “Una ciudad que usa las Tecnologías de la Información y las Comunicaciones (TIC) para proporcionar una mejor calidad de vida a sus ciudadanos.”

Otra definición más amplia y exacta, realizada por el experto en *Smart Cities*, Boyd Cohense [10], acota aún más el concepto de ciudad inteligente: “Las *Smart Cities* utilizan las TICs para ser más inteligentes y eficientes en el uso de recursos, reduciendo costes y ahorrando energía, mejorando los servicios proporcionados y la calidad de vida, y

reduciendo la huella medioambiental, todo ello con la ayuda de la innovación y una economía baja en carbono."

Así, se puede establecer que una *Smart City* es un sistema compuesto por muchos subsistemas, ya que solo el desarrollo tecnológico en múltiples ámbitos de la ciudad consigue que ésta se considere inteligente. Por cada ámbito de la ciudad que requiera de gestión, control y seguimiento se puede encontrar una buena oportunidad para desarrollar un subsistema inteligente. La agrupación de todos los subsistemas inteligentes en la misma ciudad es lo que se conoce como el sistema inteligente completo de la ciudad, la *Smart City*. A partir de esta definición se entiende que existe un gran número de ámbitos en los que se pueden desarrollar soluciones inteligentes, es más, es imposible enumerar esta cantidad, pues surgen oportunidades de desarrollo urbano en cada pequeña necesidad social. Por esto, se engloban muchos de los ámbitos de desarrollo en términos más generales que permiten diferenciar bien los diferentes subsistemas que componen una *Smart City*. Los principales subsistemas que se pueden encontrar en una *Smart City* son los que se muestran en la Figura 2.4. En esta imagen se visualiza cómo el sistema completo o *Smart City* está compuesto por diferentes subsistemas como son *Smart Mobility*, *Smart Enviroment*, *Smart Grid*, *Smart Building and Home* o *Smart Government*.

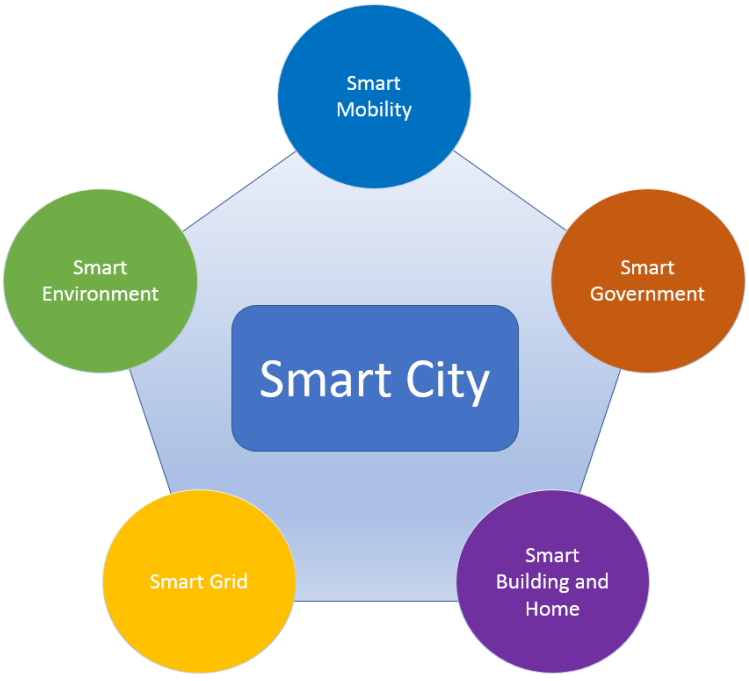


Figura 2.4: Smart City como sistema de sistemas.

2.2.1 *Smart Environment*

Dentro de este subsistema se engloban varios ámbitos de vital importancia para una ciudad. Uno de ellos es el uso eficiente y limpio del agua o *Smart Water*. Se sabe que el agua es un recurso limitado indispensable en toda civilización, y esto se ha demostrado a lo largo de la historia. Las grandes civilizaciones pasadas se asentaban en áreas marítimas o fluviales con el fin de obtener mayores facilidades en el día a día, y evitar tener que realizar grandes desplazamientos para acceder a este recurso asociado a la vida.

Actualmente, en la mayoría de las ciudades los sistemas de gestión y tratamiento del agua no son ni eficaces ni óptimos, ya que hasta un 50% del agua que entra en el sistema se pierde por fugas en las infraestructuras hídricas [11]. Por otro lado, se vuelve a apreciar cómo la saturación de las ciudades perjudica un sistema imprescindible como el del tratamiento de aguas residuales. Las plantas de tratamiento de aguas residuales no se han ido ajustando al crecimiento demográfico experimentado en los últimos años, y por ello se han visto sobrepasadas. Adicionalmente se tienen redes de abastecimiento de agua potable envejecidas, y que no se han renovado con el uso de materiales más eficientes. Muchas veces, al igual que en los sistemas de riego, existen puntos donde se pierde agua, aunque por lo general resulta muy difícil detectar dichos puntos al tratarse de redes de distribución de un tamaño considerable. Posiblemente el riego inteligente pertenezca a otro ámbito más específico en el que se desarrollen nuevas técnicas eficientes para una agricultura sostenible, pero está claro que es una actividad determinante para el ahorro de agua y por ello puede incluirse en el subsistema de *Smart Water*.

Entre los grandes objetivos de este subsistema de las *Smart Cities* se pueden destacar los siguientes [11]:

- Automatizar las infraestructuras que componen el ciclo del agua en las ciudades a partir de sensores y actuadores. Las principales infraestructuras serían las plantas de tratamiento y depuración, las redes de abastecimiento, el alcantarillado, y las diferentes tomas de agua.

- Centralizar y monitorizar la información del ciclo del agua urbano en una única plataforma de gestión.

- Desplegar sistemas de tele-lectura a través de contadores inteligentes de agua.

- Dotar de un sistema de riego inteligente a los parques y jardines públicos.

Al igual que con el agua, la basura y los residuos también tienen un ciclo que recorrer en las ciudades, siendo otro de los ámbitos que se incluyen en el subsistema de *Smart Environment*. Los servicios de limpieza, recogida y tratamiento de residuos son un factor vital de nuestra sociedad y suponen un coste importante, por lo que es deseable mejorar la eficiencia de estos servicios. Se debe destacar que la cantidad de residuos generados crece a un ritmo de 0.6% al año [11]. Entre los objetivos que más demandan las ciudades, algunos de ellos son [11]:

- Dotar de sensores a los distintos elementos del servicio de recogida de residuos para que se pueda conocer en tiempo real el estado de las basuras o contenedores y optimizar el proceso de recogida.

- Automatizar los procesos en las plantas de tratamiento de residuos.

- Monitorizar el ciclo de los residuos en una única plataforma disponible para las instituciones o entidades responsables de su recogida y tratamiento.

Además del agua y los residuos, en el subsistema *Smart Environment* también se estudia la contaminación y la calidad del aire, así como el respeto a los diferentes ecosistemas, como el mar, los ríos o los bosques. Representa uno de los subsistemas principales de las *Smart Cities*, ya que trata de mejorar las condiciones de los recursos más importantes para la vida.

2.2.2 *Smart Grid*

El mayor consumo de energía eléctrica se produce en las ciudades, y en términos generales este consumo es el responsable de un 30% en la emisión de CO_2 hacia la atmósfera. En los sistemas actuales la generación de energía es centralizada y normalmente depende de una sola fuente. Además, la falta de un sistema de gestión de datos no permite optimizar el uso de la energía.

En las *Smart Grid* se pretende implantar una gestión descentralizada, disponiendo de múltiples fuentes y que muchas de estas fuentes sean de carácter renovable. También se desea contar con automatización para tener posibilidad de acción inmediata sobre la red eléctrica, además de tener sistemas de información que permitan optimizar el consumo.

A diferencia de las redes eléctricas tradicionales, en las redes inteligentes, la participación de todos los agentes que intervienen en ella es fundamental. Se pueden definir dos interfaces que relacionen los distintos elementos de la *Smart Grid*. La interfaz eléctrica, que relaciona el sistema de generación con la transmisión y distribución, y que a su vez se conecta con el consumidor, y la interfaz de comunicación que interconecta a estos tres elementos con el propio mercado, con el centro de operaciones, y con el proveedor de servicios.

La participación de todos los agentes y la aceptación de las bases que se desean para la red eléctrica son dos factores indispensables para poder dar el salto hacia las *Smart Grid*. Los grandes retos para este subsistema son [11]:

- Mejorar los niveles de eficiencia energética en la transmisión y distribución de las redes eléctricas.
- Integrar energías renovables en la red eléctrica y disminuir pérdidas en su transporte.
- Disminuir las emisiones de CO_2.
- Implicar y concienciar al consumidor en el uso eficiente de la energía.

2.2.3 *Smart Building* y *Smart Home*

Este subsistema está orientado a desplegar soluciones tecnológicas que faciliten reducir la energía consumida por los edificios y hogares, además de minimizar sus costes económicos y ambientales.

En la Unión Europea, más de un 40% del consumo de la energía se produce en los edificios. Este consumo se estima que siga creciendo y por tanto se espera que se incrementen las emisiones de CO_2.

En este ámbito se trabaja con el concepto de domótica, que permite economizar el hogar mientras ofrece servicios que faciliten y acomoden la vida de las personas que residen en la vivienda. No se debe asociar esta idea solo a los hogares, ya que normalmente también se destina un tercio de la vida cotidiana a estar en el puesto de trabajo, que en muchas ocasiones se encuentra en edificios. Generar un entorno satisfactorio y saludable para el trabajador enriquece el producto o servicio realizado, además de incrementar la calidad de vida de las personas.

Cuando se habla de *Smart Building* y *Smart Home*, se entiende que es un subsistema de elevada importancia para las *Smart Cities*, ya que son los entornos más frecuentados por las personas y las familias, y dónde más efecto tienen las soluciones tecnológicas que incrementan la calidad de vida, lo que constituye uno de los pilares fundamentales de las *Smart Cities*.

Las principales operaciones que se llevan a cabo en este subsistema son la automatización de instalaciones e infraestructuras en los edificios y su gestión integral a través de una única plataforma de control. Realizando estas acciones se consiguen los siguientes objetivos o retos que se marcan en cualquier proyecto de *Smart Building and Home* [11]:

- Monitorizar en tiempo real la información de los diferentes consumos del edificio, como el agua, la electricidad, la climatización e incluso la calidad del aire.

- Contar con un sistema de control remoto del edificio inteligente.

- Detectar fallos y anomalías en cualquier instalación del edificio fácilmente.

- Regular automáticamente todos los parámetros de confort del edificio.

- Disponer de sistemas de seguridad fiables y cómodos, tanto sobre la vivienda como con la tecnología que convierte al edificio en inteligente.

Con todo esta gestión y control sobre los edificios se pueden alcanzar grandes ahorros en los consumos, que pueden alcanzar hasta un 30% [11], a lo que se debe sumar el ahorro que supone tener localizado el error en caso de que existiese, y poder realizar un mantenimiento más certero y eficaz.

2.2.4 *Smart Government*

Tradicionalmente, los gobiernos y sus administraciones son los últimos en acogerse a las revoluciones tecnológicas por el coste que supone la actualización de todos sus servicios. Sin embargo, ante un auge tan importante, y en el contexto de las *Smart Cities*, las instituciones no pueden quedarse atrás, siendo ellas las encargadas de aceptar y fomentar las ciudades del futuro. Se debe recordar que las TIC no son el objetivo, sino el medio para que la ciudad disponga de los elementos necesarios para que se convierta en una ciudad inteligente y sostenible que beneficie al ciudadano.

Desde esta idea se parte hacia la modernización de muchos de los servicios administrativos. Los principales cambios que se desean adaptar, y que de hecho ya están siendo ejecutados son, entre otros, los siguientes:

- Acelerar trámites y solicitudes de los ciudadanos mediante soluciones telemáticas.

- Contar con un servicio de ayuda al ciudadano para que en cualquier momento se pueda realizar una consulta administrativa.

- Crear herramientas que permitan hacer partícipe al ciudadano de decisiones públicas.

- Favorecer la transparencia y los datos abiertos mediante el uso de la tecnología de la información y comunicación.

Además de estos puntos, se desean desarrollar herramientas que favorezcan las versiones inteligentes del comercio habitual o de los negocios, así como de la sanidad y la educación.

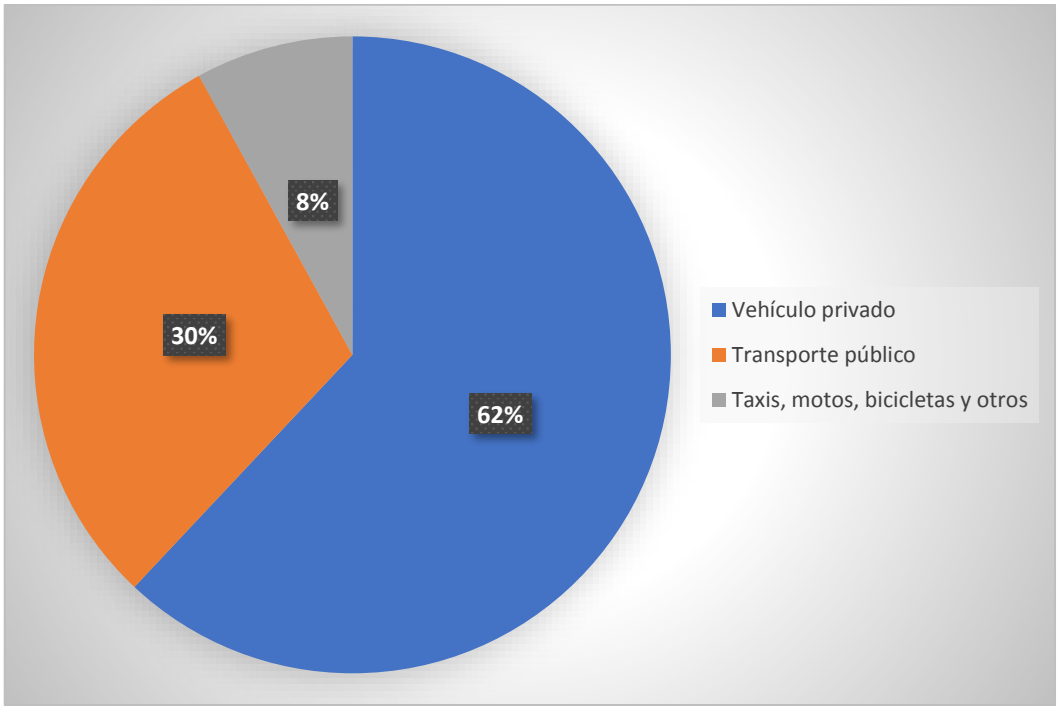
2.3 *Smart Mobility* y *Smart Parking*

La movilidad y el tráfico representan un gran desafío para las ciudades del futuro, ya que la mayoría de las ciudades han ido construyéndose según la necesidad de acoger a personas que deciden cambiar los pueblos o zonas rurales por grandes metrópolis. En esta expansión, se construyen muchos hogares, pero se descuidan aspectos como la movilidad. Así, se incrementa el número de habitantes, y el número de viviendas, pero no se mejoran ni ensanchan las carreteras o vías urbanas, causando así saturaciones que dificultan el transporte.

En las grandes ciudades de España, como en el caso de Madrid, el aumento de población y la falta de adaptación de las ciudades a este gran cambio está generando enormes problemas de movilidad y por tanto de contaminación. La estadística dice que cada conductor que reside en la capital española sufre aproximadamente 42 horas de atascos al año. Este es un gran indicativo de la cantidad de tráfico existente en las grandes ciudades. Por supuesto, el número de desplazamientos ya genera elevados niveles de contaminación, pero si a eso se le suma la falta de agilidad de los vehículos por el colapso de las carreteras, las emisiones de gases nocivos para el medioambiente se incrementan considerablemente. De hecho, el 41% de las emisiones de gases de efecto invernadero en el municipio de Madrid están relacionados con el transporte [12], lo cual equivale a 7 toneladas métricas de emisiones de dióxido de carbono.

Otro apunte importante es que, en la mayoría de las grandes ciudades, el transporte más utilizado es el vehículo privado. Volviendo al caso de la ciudad de Madrid, el 80% de la contaminación generada por transporte se debe al uso de vehículos privados. Si se considera el número de kilómetros que se recorren aproximadamente en una ciudad como la madrileña, esta cifra alcanza los 23.000 millones de Km al año, que se reparten

como se muestra en la Gráfica 1. Como se puede ver en esta gráfica los vehículos privados son el transporte que más Km recorren en comparación con el resto.



Gráfica 1: Porcentaje de Km recorridos al año por cada tipo de medio de transporte en Madrid.

Para mejorar el sistema actual de movilidad y transporte en las grandes ciudades, se plantean cuatro grandes cambios. Estos grandes cambios, recogidos en la siguiente lista, son al fin y al cabo los objetivos principales que concentran la idea de *Smart Mobility*:

- Mejorar el servicio de transporte público.
- Diseñar y fomentar el uso de vehículos sostenibles.
- Facilitar el estacionamiento de los vehículos.
- Agilizar la circulación.

2.3.1 Transporte público

En este objetivo del subsistema *Smart Mobility*, se deben acometer dos cambios importantes, siendo uno de ellos dependiente del otro. Así, se debe concienciar a la sociedad de que el uso del transporte público favorece una circulación ágil, menos

saturada y por supuesto menos contaminante. Sin embargo, para que todo este movimiento de concienciación cale en los ciudadanos, este debe ir acompañado de mejoras técnicas en las que tiene sentido el concepto de *Smart Mobility*.

Existen dos tipos de mejoras técnicas que se deben dar en el servicio de transporte público. La primera está asociada directamente a los vehículos, donde se pueden mejorar aspectos mecánicos, como reducir los niveles de contaminación, el ahorro de energía y combustible, etc… Y la segunda va más hacia la idea de confort del cliente, así como de su seguridad dentro del vehículo.

En los dos tipos de mejoras se pueden dar soluciones que sigan el patrón de *Smart City* o *Smart Mobility*. Por ejemplo, en el aspecto mecánico se pueden desarrollar soluciones que en tiempo real vayan diagnosticando el estado del vehículo y mostrar recomendaciones para una conducción sostenible o para un mantenimiento eficaz. Por otro lado, en la búsqueda de la comodidad y la satisfacción del usuario de este servicio público, se puede mejorar la información sobre el propio servicio y sobre todo la forma de acceder a ella. Existen varias soluciones de estas características que ya se están implantando, si bien para dar mayor vitalidad al servicio público de transporte, aún se deben acometer bastantes mejoras que permitan al ciudadano plantearse en todo momento la opción del transporte público. Una de las soluciones *Smart Mobility* que ya se ha desarrollado y está operativa, es la aplicación móvil de Guaguas de Las Palmas de Gran Canaria, *GuaguasLPA* [13]. En esta aplicación, entre otras utilidades, se muestra el tiempo que tardará en llegar la siguiente unidad de transporte público a una parada determinada por el usuario. La interfaz es como la que se muestra en la Figura 2.5.

Además, en este proyecto se ha contado con la colaboración de las instituciones públicas, que no solo han promovido el desarrollo de la aplicación móvil, sino que han colocado paneles informativos con la misma información en todas las paradas del servicio. De esta manera, se facilita el uso del transporte público en esta ciudad fomentando la movilidad inteligente o *Smart Mobility*.

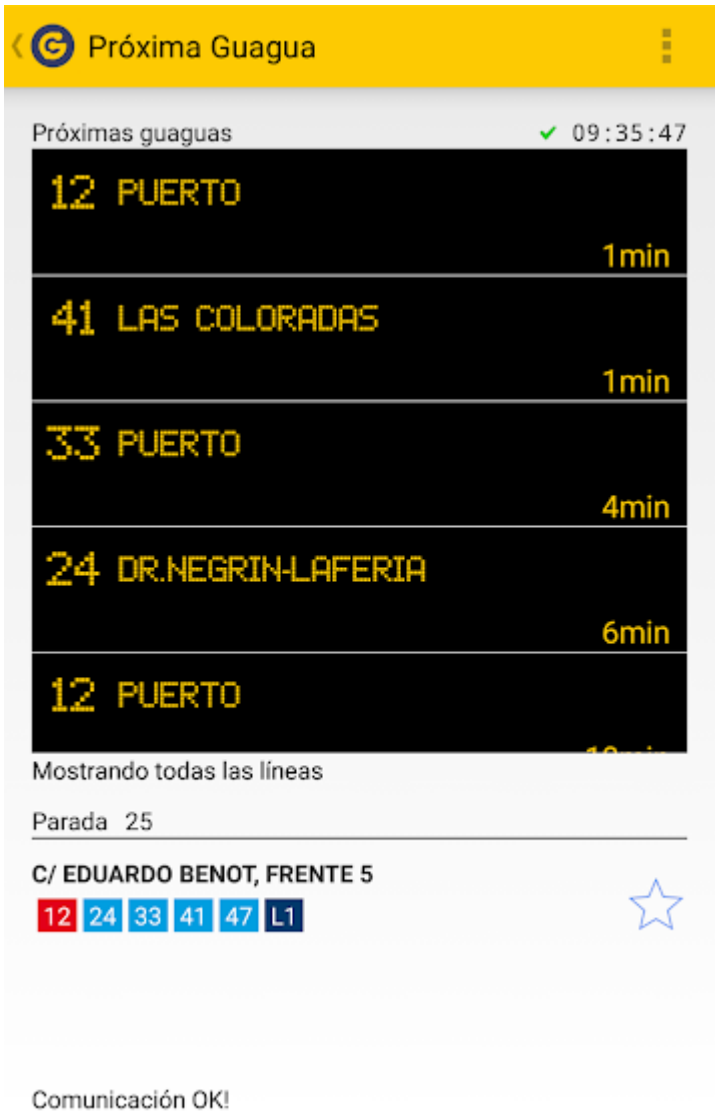


Figura 2.5: Interfaz de la aplicación móvil GuaguasLPA.

2.3.2 Circulación y gestión del tráfico sostenible

En este ámbito de *Smart Mobility* es imprescindible aportar desde el uso de la tecnología soluciones de monitorización que permitan a las autoridades disponer de una visión global del estado del tráfico y tener la capacidad de estudiar en tiempo real los acontecimientos en una zona concreta. Con toda la información disponible se pueden tomar mejores decisiones y favorecer una circulación ágil y sin congestiones.

En el caso español, desde hace bastantes años, la Dirección General de Tráfico (DGT) cuenta con información visual del estado de muchas vías, pero incrementar la cantidad y

la calidad de dicha información es una de las grandes aplicaciones que se espera que el subsistema *Smart Mobility* desarrolle para seguir mejorando la circulación y la gestión del tráfico.

Normalmente se disponen grandes pantallas en los centros de la DGT para visualizar el estado del tráfico desde cualquier posición del centro de control, como se muestra en la Figura 2.6. Es en estos centros, donde a partir de la información recibida, se decide recomendar rutas alternativas, cerrar carreteras o advertir a los conductores de situaciones de peligro. Por lo tanto, se puede ver cómo también se deben desarrollar herramientas que permitan la comunicación veloz y efectiva desde la DGT hacia los conductores o al revés.

Figura 2.6: Centro de la DGT en Madrid.

2.3.3 *Smart Parking*

El presente Trabajo Fin de Grado se centra en este campo de *Smart Mobility*. Las soluciones *Smart Parking* son cada vez más demandadas, ya que actualmente un 30% del volumen del tráfico en el centro de las ciudades está relacionado con la búsqueda de

aparcamiento [5], dejando a los conductores durante una media de 20 minutos en la ardua tarea de encontrar un espacio disponible para estacionar su vehículo.

Con soluciones de *Smart Parking* se pretende simplificar esta tarea mediante diferentes procesos tecnológicos garantizando la sostenibilidad, el bienestar medioambiental, la eficiencia, y por supuesto, el ahorro de tiempo de las personas, evitando de esta manera malestar en ellas, que puedan desencadenar conductas negativas y conflictivas en la conducción. Estas conductas negativas no se quedan en el momento del estacionamiento, sino que toda la tensión acumulada repercute directamente en la salud de los conductores, tanto física como mentalmente. Muchas veces, además, la búsqueda de aparcamiento se complica en hora punta, provocando que los trabajadores lleguen tarde a sus trabajos u obligaciones, y se reduzca la productividad.

Existen diferentes soluciones desarrolladas en este ámbito de las *Smart Cities*. Muchas de ellas se encuentran en fases de prueba, o tratando de optimizarse, por lo que queda mucho camino por recorrer en este ámbito para tratar de alcanzar un sistema de aparcamiento cómodo, sencillo, rápido y accesible. Actualmente las principales herramientas que se utilizan en *Smart Parking* son [14]:

- *Software* de análisis de fotos y vídeos: Comprueban la ocupación de las celdas y detectan vehículos estacionados de forma ilegal.

- Sensores de proximidad: Sensores que detectan el estado de la plaza, suelen ser sensores infrarrojos.

- Paneles *Light-Emitting Diode* (LED): Soporte informativo que muestra el estado de las plazas de estacionamiento.

- Parquímetros inteligentes: Recogen la información sobre las plazas que están ocupadas, además de cumplir con su función habitual.

- *Tags RFID*: Contienen información asociada al dueño del vehículo, aspecto útil para promociones o agilidad en el pago. La información se transmite

mediante campos electromagnéticos, con la ayuda de lectores que normalmente utiliza el personal de la zona de aparcamiento.

- Aplicaciones web y móvil: Con capacidad para realizar reservas, pagos y renovaciones desde cualquier dispositivo con conexión a Internet.

Con todas estas soluciones, que son algunas de todas las que están desarrollándose, y de las que están por venir, se logran unos beneficios importantes. Según [14] se reduce en un 43% el tiempo empleado en buscar aparcamiento y un 30% menos de Km recorridos en esta tarea. Además, con estas ideas de *Smart Parking* se consigue también una base de datos con la información de los diferentes comportamientos, horarios de uso más habituales, zonas más frecuentadas, etc. En resumen, información sobre el usuario que permita generar soluciones más aproximadas a las necesidades reales del conductor.

En la Figura 2.7, extraída también de [14], se muestran en cifras los beneficios de hacer uso de las soluciones de *Smart Parking*.

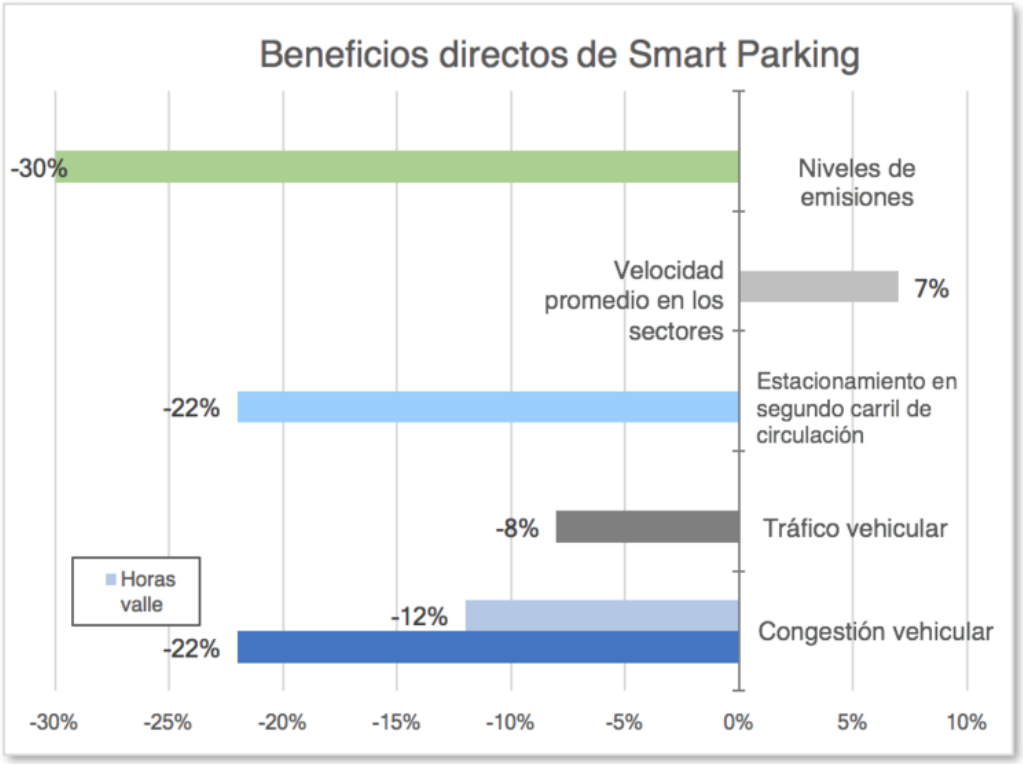


Figura 2.7: Beneficios directos de Smart Parking.

2.4 *Internet of Things*

Internet está evolucionando y, además, a un ritmo vertiginoso. Tanto es así, que actualmente se pueden conectar a Internet cámaras IP (*Internet Protocol*), *Smart Television* (TV), móviles, impresoras, y muchos otros dispositivos electrónicos. Desde hace algunos años, se comenzó a imaginar lo que hoy se conoce como *Internet of Things (IoT)*, pero todavía no se consideraba más que una propuesta futurista. Sin embargo, hoy en día, ya es una realidad, y cada vez se están desarrollando más ideas en este ámbito. A esto hay que sumar que la mayor parte de los dispositivos electrónicos disponen de alguna opción que permite conectarlos a Internet, favoreciendo aún más el desarrollo de soluciones IoT.

El concepto de IoT se basa fundamentalmente en la interconexión de cualquier producto con otro, haciendo uso de Internet, y creando de esta manera un entorno inteligente y versátil. La interconexión de productos, máquinas, y objetos, permite crear una red de dispositivos conectados que debe ser gestionada y controlada. Al mantener los objetos conectados, se puede conocer su estado, y en consecuencia interactuar con ellos.

Si se analiza el crecimiento en este ámbito, se puede deducir que es un mercado creciente, ya que los grandes protagonistas, los dispositivos conectados a Internet, no hacen más que incrementar su número, tal y como se indica en la Figura 2.8, extraída de *Statista* [15], el portal de estadísticas *online*.

La gráfica de la Figura 2.8 muestra que cada vez hay más dispositivos conectados a Internet, por lo que el número de dispositivos que se pueden usar para aplicaciones IoT se incrementa también. En el año 2012 se alcanzó la cifra de los 8.700 millones de dispositivos conectados, y para el año 2020 se estima que se alcancen 50.100 millones. Cada año el número de dispositivos conectados se incrementa, y en las estimaciones a corto plazo el crecimiento se espera que sea aún mayor.

Para hablar de IoT, se debe estudiar el nivel de inteligencia. El nivel de inteligencia de los objetos es el grado de conocimiento y control sobre el dispositivo conectado. Así, existen diferentes niveles, los más bajos, implican un menor grado de interactividad con el

dispositivo y los más altos, permiten un mayor número de funcionalidades al tener mayor control y conocimiento del dispositivo conectado.

- *Nivel 1*: Identificación. Nivel mínimo e imprescindible. Se puede identificar el objeto de forma única.
- *Nivel 2*: Ubicación. Se conoce la ubicación del dispositivo, tanto actual como pasada.
- *Nivel 3*: Estado. Se conoce la información, el estado y las características del dispositivo. Asociado al uso de sensores.
- *Nivel 4*: Contexto. El dispositivo es capaz de conocer su entorno y transmitir información de este.
- *Nivel 5*: Criterio propio. El dispositivo es capaz de ejecutar acciones según la información, la ubicación, el estado, el entorno, y otras circunstancias.

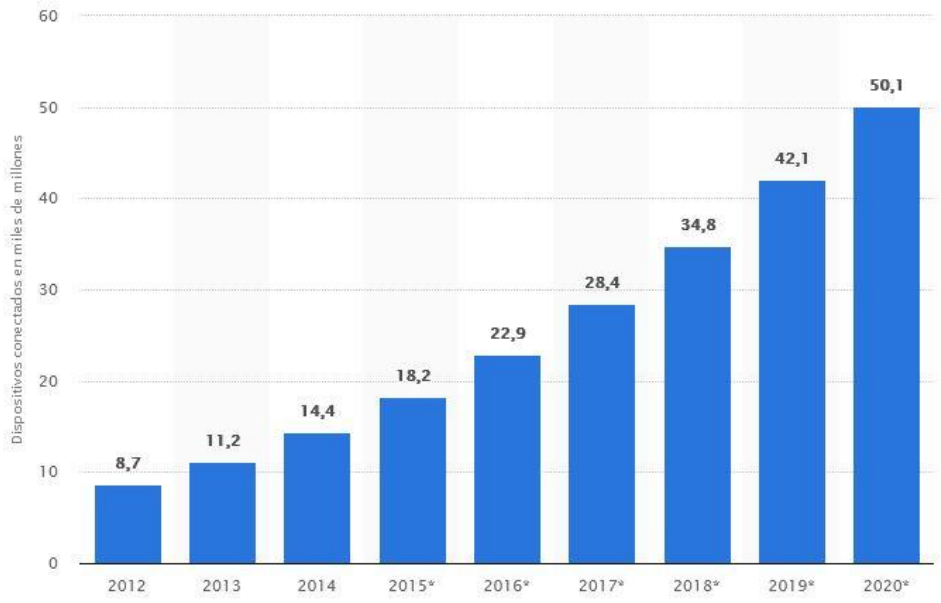


Figura 2.8: Número de dispositivos conectados a Internet.

Para los diferentes niveles de inteligencia se utilizan tecnologías asociadas. Como referencia, para el nivel de identificación resulta muy útil hacer uso de tecnologías como *Radio Frecuency Identification* (RFID) o *Near Field Communications* (NFC), mientras que para el nivel de ubicación se suele utilizar la tecnología *Global Positioning System* (GPS),

con el propósito de medir el estado y en contexto se incluyen sensores, y para el nivel de criterio propio del dispositivo se pueden añadir directamente actuadores.

Si se resumen las capacidades que acreditan a los objetos con un nivel de inteligencia u otro, se pueden destacar las siguientes.

- Comunicación y cooperación: Los objetos se conectan a los servicios de Internet, y también entre ellos, favoreciendo la cooperación.
- Capacidad de direccionamiento: Localizable en la red IoT.
- Identificación: Se logra identificar al objeto mediante diferentes tecnologías.
- Localización: Asociado al nivel de inteligencia de ubicación.
- Actuación: El objeto posee actuadores y sabe cómo gestionar su uso.

2.5 El binomio *IoT - Smart Cities*

Internet of Things es un tipo de solución tecnológica muy utilizada en el desarrollo de *Smart Cities*. En *Smart Cities*, la comunicación es un elemento fundamental, y haciendo uso de las ideas y aplicaciones IoT, se pueden completar las comunicaciones de una forma robusta y eficiente. Además, para las diferentes aplicaciones se puede optar por una solución IoT u otra, dependiendo de las características que demande en cada caso. La idea de IoT se ajusta perfectamente al modelo de *Smart City*. La interconexión de los objetos en una ciudad posibilita la comunicación entre ellos, y, por tanto, se permite aumentar el nivel de automatismo e independencia de los sistemas. Además, haciendo uso de soluciones IoT, se logra establecer comunicaciones entre los objetos y los usuarios, facilitando así muchos procesos.

A continuación, se muestran como referencia varios ejemplos que hacen uso del binomio formado por tecnologías IoT, y el concepto de *Smart City*.

- En 2011 se lanzó un proyecto de *Smart City* en la ciudad de Santander, España, que se llamó *SmartSantander* [16]. El proyecto consiste en dotar a

la ciudad de múltiples sensores para que recojan información que posteriormente será utilizada. Estos sensores se colocaron en fachadas de edificios, marquesinas y otros puntos estratégicos de la ciudad. El número de sensores implantados se aproxima a la cifra de 15.000, y toda la información que recogen, se envía a una base de datos, donde herramientas de *Big Data* ordenan y clasifican la información. Esta información está destinaba a mostrar datos sobre la calidad del aire, estado de los contenedores, tráfico y circulación, y otros aspectos de interés en el ámbito de la ciudad cántabra. Con toda esta información se pueden tomar mejores decisiones por parte de las autoridades responsables de la ciudad, y facilitar información eficaz al ciudadano.

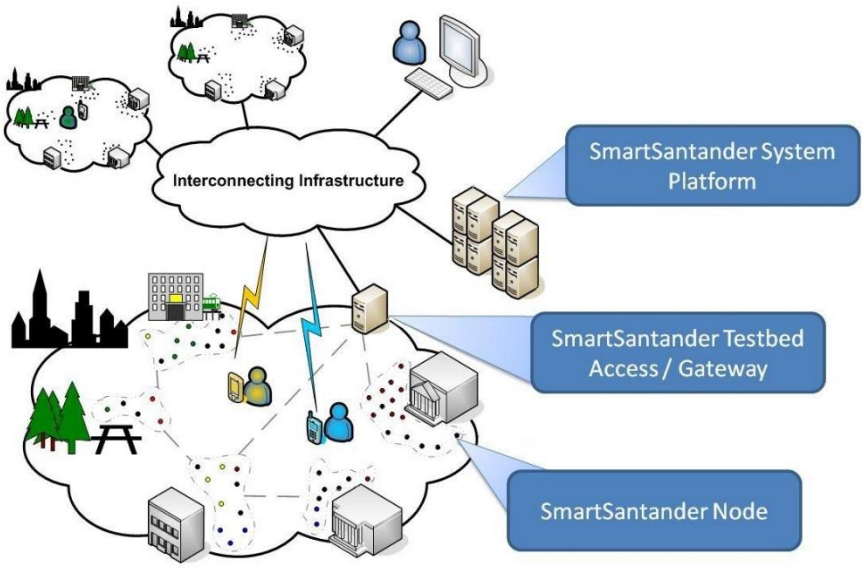


Figura 2.9: Estructura del proyecto SmartSantander.

En la Figura 2.9 se muestra la estructura que sigue el proyecto *SmartSantander* [16]. En esta estructura se pueden apreciar los tres niveles típicos de una red IoT:

- Nodos finales: Incluyen los sensores y si fuera el caso, también los actuadores.

  - Nodo concentrador o *gateway*: Encargado de comunicarse con los nodos finales y hacer de enlace con los servicios de Internet.

  - Plataforma: Donde se acumulan los datos. Sobre este nivel suelen trabajar las estrategias de *Big Data*. Desde aquí se ofrece la información.

- Otro ejemplo es el proyecto que se está probando en Noruega, relativo a las farolas inteligentes [17]. Es cierto que en otras muchas ciudades se han utilizado sensores de movimiento para detectar si hay una persona en un habitáculo y apagar o encender las luces en función de la respuesta del sensor, pero este proyecto va más allá. No se trata de la gestión del sistema de alumbrado de un parque, de un hotel o de una casa, sino de una carretera. En la localidad de Hole se han instalado farolas inteligentes en vías públicas, concretamente en un trazado de 9 Km. Cuando se detecta que no hay vehículos o transeúntes en el tramo, los LEDs de las farolas funcionan al 20% de su potencia, mientras que si se detecta movimiento, la potencia se incrementa hasta lograr el 100% disponible.

  Esta idea, que se basa en el uso de sensores e iluminación LED, es una buena propuesta de *Smart City*, concretamente en el ámbito de *Smart Mobility*, ya que según las estimaciones realizadas permite ahorrar 2100 kWh a la semana, por lo que la inversión estaría amortizada en 4.5 años.

- Como tercer ejemplo para resaltar la importancia del binomio de las tecnologías IoT con el concepto de *Smart City*, se presenta una solución *Smart Environment*. Se trata de la gestión inteligente de los residuos, una idea que está siendo explotada por *eCube* [18], una empresa surcoreana que está recibiendo muy buena aceptación en todo el mundo.

  El tratamiento de residuos que la compañía propone se basa en la utilización de contenedores de basura inteligentes, como los mostrados en

la Figura 2.10 [18]. Estos contenedores detectan si están llenos, y en ese caso avisan a los encargados de su recogida. De esta manera, se puede trazar una ruta óptima para la recogida de basura diaria y no mantener una ruta fija e ineficiente en muchas ocasiones. Además, la solución hace uso de datos históricos para ofrecer análisis predictivos que permitan una mejor planificación. Bajo esta idea innovadora, los costes operativos se pueden reducir hasta en un 80%. Una limpieza bien planificada y precisa, favorece el entorno de la ciudad, al no dejar las calles abarrotadas de basura, como ocurre en muchos casos. Los contenedores además se alimentan con energía solar.

En este ejemplo la solución es claramente del tipo IoT, y respetando los pilares fundamentales de *Smart Cities*.

Figura 2.10: Contenedores de basura inteligente de la empresa eCube.

2.6 Aplicaciones desarrolladas

2.6.1 *Smart Parking*

La compañía de Reino Unido *SmartParking* [19] ha desarrollado una plataforma similar a la que se expone en este TFG. Los aspectos que mantienen en común son principalmente los beneficios que se desean aportar al conductor. En este caso se hace uso de una

aplicación móvil, que se recomienda consultar antes de iniciar la conducción para localizar las plazas de aparcamiento que se encuentran disponibles. Esta misma aplicación, una vez comenzado el trayecto, va dando indicaciones para llegar a la plaza haciendo uso de tecnologías de localización, como GPS. Una vez que el vehículo haya estacionado, el sensor reúne información de este para el pago, el cumplimiento y la gestión del espacio. El sensor utilizado es de carácter magnético, aparte del uso de infrarrojos.

Además de la aplicación y del sensor ubicado en la plaza de aparcamiento, se cuenta con un *software* para la gestión y administración de las plazas que permite realizar planificaciones diariamente y, además, a largo plazo. Para este estudio a largo plazo, es necesario la recopilación de una cantidad significativa de información que se recoge en cada estacionamiento, almacenando datos, como la hora o la duración del estacionamiento. En la Figura 2.11 se muestra como referencia una captura de este *software*, denominado *SmartRep*.

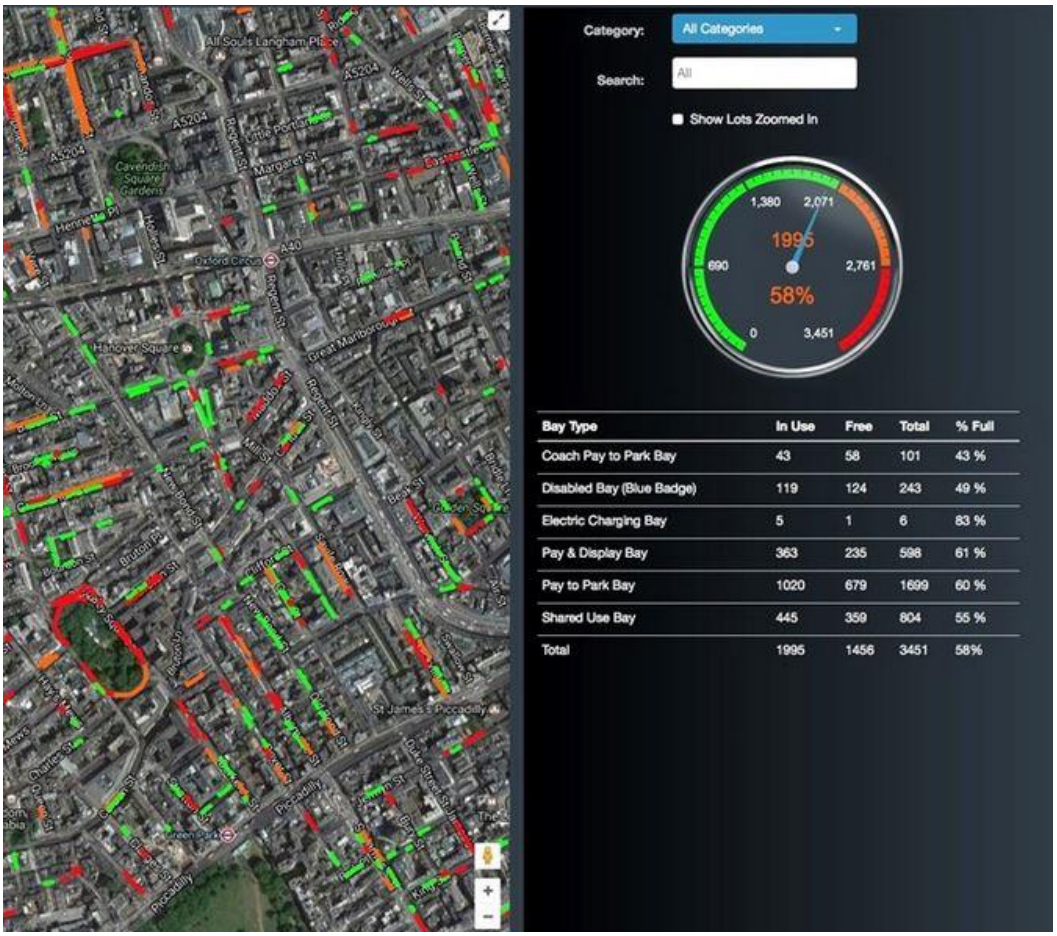


Figura 2.11: SmartRep. Software de aparcamiento inteligente.

Otro servicio parecido es el proporcionado por la empresa I+3D. La gran diferencia con la solución anterior es que el epicentro de los servicios ofrecidos es la gestión y control de técnicas de *Big Data* sobre toda la información recogida. El proyecto 3SCPARK [20] se presenta como "un *Big Data* que permite a los sistemas conectados, compartir datos y vincular servicios para facilitar la movilidad de los ciudadanos". Debido a la arquitectura que utiliza esta solución IoT, se dispone de un sistema centralizado que integra todos los servicios que un aparcamiento inteligente necesita. En la Figura 2.12 se muestra la arquitectura utilizada en el proyecto 3SCPARK.

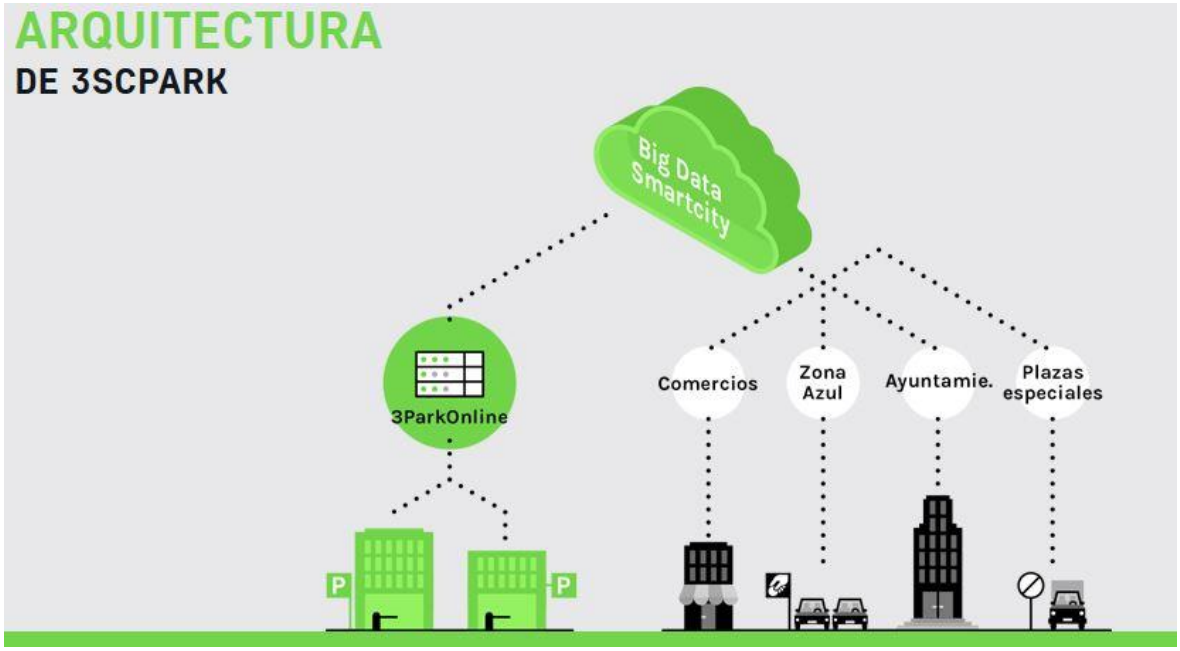


Figura 2.12: Arquitectura de 3SCPARK.

Al tener todos los elementos conectados a *Big Data* 3SCPARK, incluidas las zonas azules o comercios, se pueden vincular diferentes servicios, facilitando siempre el proceso al usuario. En resumen, se tienen cuatro elementos conectados constantemente al sistema 3SCPARK:

- Aplicación móvil: A través de la aplicación los clientes pueden buscar aparcamiento, abrir la barrera y pagar el estacionamiento.

- Portal web: Se puede buscar toda la información relacionada y realizar reservas.

- Centro de control: Para controlar y gestionar toda la infraestructura de los aparcamientos desde un único lugar.

- Parking: El propio *parking* está conectado, y al tratarse de un sistema centralizado, permite integrar más de un aparcamiento, pudiendo gestionarlos y controlarlos de manera conjunta.

2.6.2 LoRa/LoRaWAN

Con respecto al uso de la tecnología de comunicación LoRa/LoRaWAN, se comenta a continuación la aplicación instalada en una de las comunidades más concienciadas con el medio ambiente del Líbano. A veinte minutos de la capital, Beirut, se encuentra una población llamada *BeitMisk*. Este entorno está ideado para ser ecológico, autosuficiente y lo más limpio posible, medioambientalmente hablando. Uno de los aspectos que más cuidan en esta comunidad es la calidad del aire, motivo por el que han implantado una solución basada en un sistema de sensores que envían la información a una plataforma digital permite conocer el estado del aire en la zona. A esta información tienen acceso, tanto la administración de la urbanización, como los habitantes de esta comunidad. Este sistema también cuenta con la capacidad de monitorización en tiempo real.

La solución desarrollada por la empresa libanesa *Data Consult*, junto con la española *Libelium* [21], ha llegado al despliegue de sensores por toda el área de la comunidad para lograr una mayor interacción con el entorno. Un ejemplo de esto puede ser la notificación del mejor momento del día, en términos de calidad del aire, para dar un paseo, hacer ejercicio en el exterior, o que los niños jueguen en el parque. Además del despliegue de sensores, se ha procedido a la digitalización de los elementos físicos de la ciudad, añadiendo una capa de aprendizaje automático e inteligencia artificial, ofreciendo así información sobre los datos y análisis predictivos.

Los nodos finales, a través de los sensores, miden los parámetros de temperatura, humedad y presión atmosférica, monóxido de carbono (CO), dióxido de nitrógeno (NO_2), dióxido de azufre (SO_2), ozono (O_3) y las partículas contaminantes o *Particulate Matter* PM1, PM2.5, PM10 [21].

Para la comunicación entre estos nodos finales y los *gateways* se utiliza el protocolo de comunicación LoRa/LoRaWAN. Se debe destacar que los *gateways* utilizados forman parte del despliegue de *Ogero Telecom* para la red nacional, con el fin de extender las soluciones IoT en el país. Los *gateways* están conectados con el sistema de almacenamiento online que ofrece *Google*, y es desde aquí, donde se envía la información a la plataforma y a los servidores desarrollados por *Data Consult*. Una vez que se tienen los datos en estos servidores, se procesa y refleja la información de manera eficaz. En este proceso también se añade una capa de inteligencia artificial que realiza análisis predictivos que ayudan a los procesos de toma de decisiones. En la Figura 2.13 se representa el esquema de comunicaciones que se utiliza en esta aplicación.

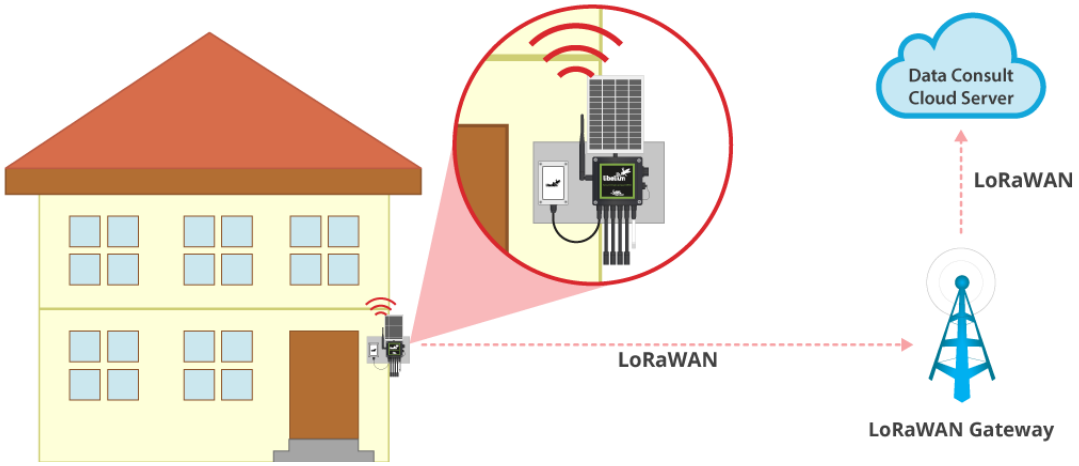


Figura 2.13: Estructura de comunicaciones en BeitMisk.

Otra utilidad, que se puede obtener de esta aplicación, aparte del beneficio diario de los ciudadanos, se da cuando *BeitMisk* acoge conciertos, festivales de cine, u otro tipo de actividades sociales. En estos casos, el sistema permite a los responsables de los eventos y a las autoridades de la comunidad planificar mejor las actividades, proyectos o construcciones sin incrementar los niveles de contaminación.

Por lo tanto, este despliegue permite, tanto a los habitantes como a la administración de la comunidad, conectarse con su ciudad a través de herramientas de comunicación intuitivas, ahorrando recursos y respetando en todo momento el medioambiente. Los distintos usuarios pueden acceder a la información de diferentes formas, dependiendo del consumidor. Los responsables del sistema en la comunidad pueden acceder a la

información a través de *dashboard web*, una interfaz web como la que se muestra en la Figura 2.14. En cambio, los vecinos no requieren de la información completa, y por eso cuentan con una aplicación móvil que contiene directamente la información útil para ellos.

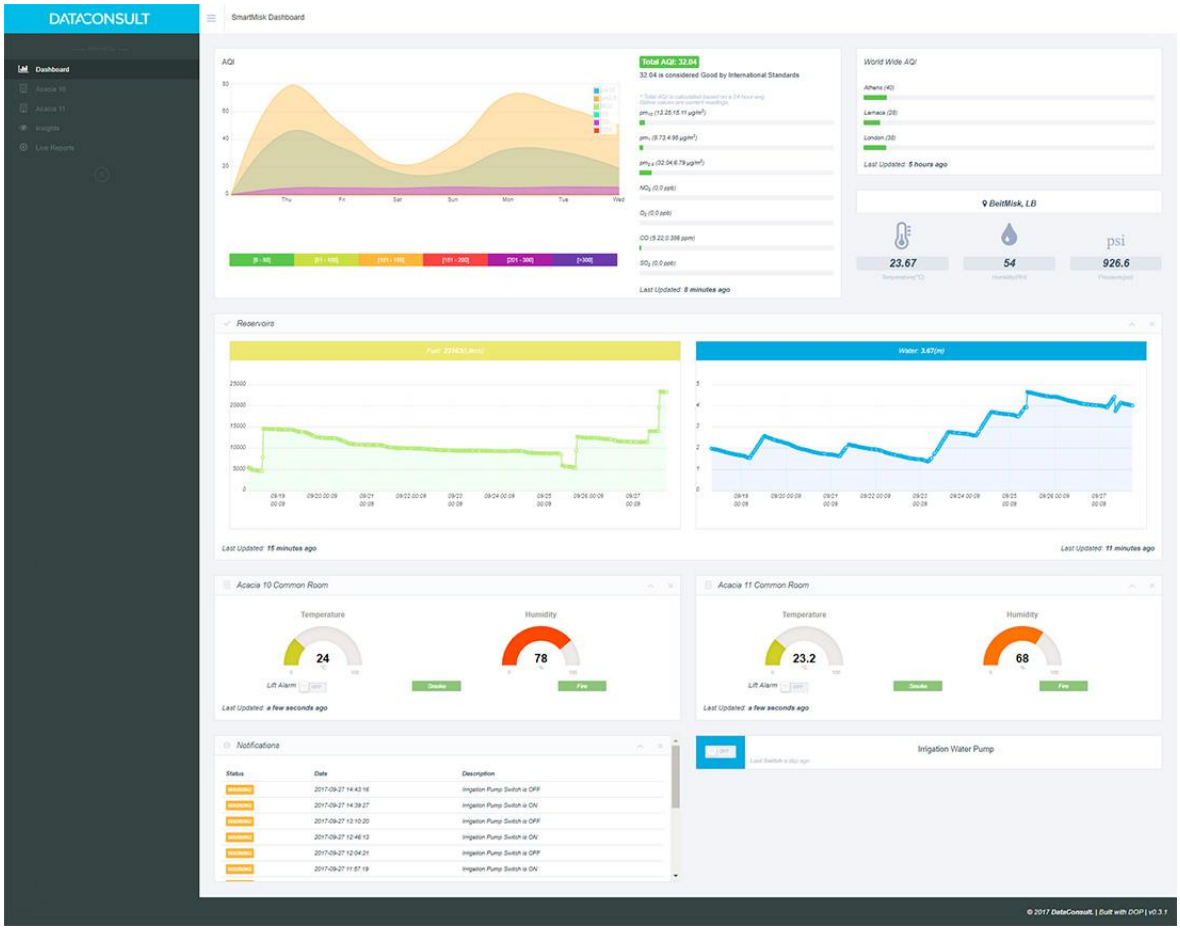


Figura 2.14: Dashboard web de la solución Smart Environment de BeitMisk.

Otra aplicación que hace uso de la tecnología de comunicación LoRa/LoRaWAN es *InteliLIGHT* [22]. *InteliLIGHT* es una solución de gestión remota de alumbrado público y una plataforma de ciudad inteligente desarrollada por *Flashnet*. La gestión y el control de la red en profundidad permite recibir una retroalimentación en tiempo real de cualquier cambio que ocurra en la red. Contar con esta posibilidad reduce la pérdida de energía y ofrece herramientas avanzadas de optimización para el mantenimiento. Concretamente, en la referencia [22] se especifica que el uso de esta solución supone hasta un 80% de ahorro de energía y un 42% de ahorro en costes de mantenimiento. Actualmente, *InteliLIGHT* cuenta con más de 80.000 reguladores de iluminación en 10 ciudades de todo

el mundo, consiguiendo así reducir el consumo de energía de iluminación y las correspondientes emisiones de CO_2.

La arquitectura que se utiliza en esta solución sigue el patrón de la aplicación anterior. Es decir, se tienen nodos finales, que en este caso serían los elementos de iluminación de la vía pública, que envían y reciben información vía LoRa/LoRaWAN a los *gateways* concentradores. Y desde los *gateways* se envía la información a través de comunicaciones *Transmission Control Protocol/Internet Protocol* (TCP/IP) hacia los servidores, en los que se hace uso de los datos recogidos y se toman decisiones que se enviarán a los nodos finales. En la Figura 2.15 se puede ver una representación de la arquitectura descrita.

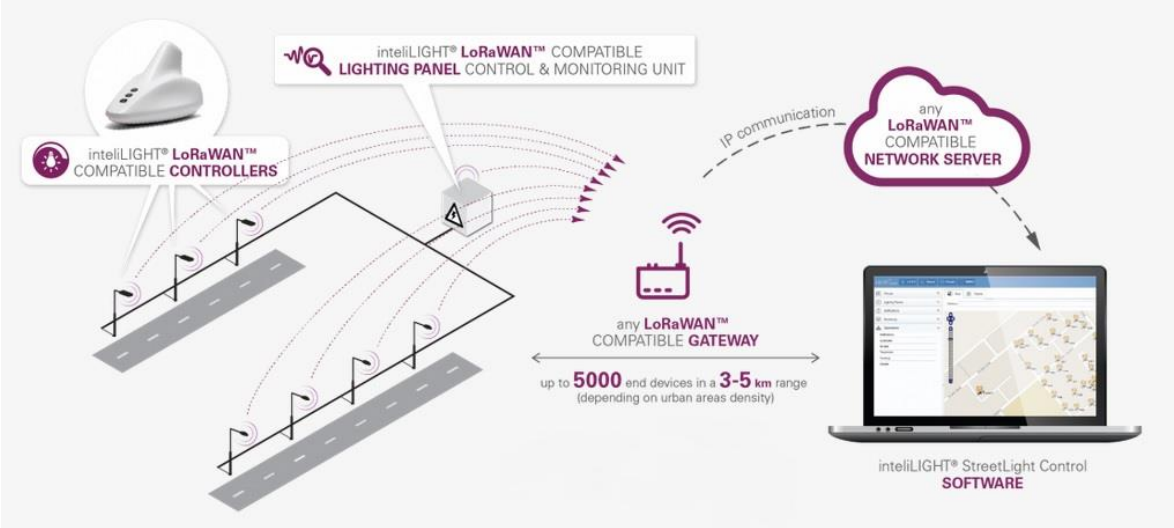


Figura 2.15: Arquitectura utilizada por la solución InteliLIGHT.

2.7 Tecnologías de comunicación

En la gran mayoría de las soluciones de *Smart City,* y en concreto en las soluciones *Smart Parking*, se sigue la arquitectura nodo final-*gateway*-servidor. La arquitectura habitual de este tipo de soluciones se muestra en la Figura 2.16, y como se ha visto en apartados anteriores, una de las claves de estas soluciones, que pretenden ser eficientes y sostenibles, es el uso de tecnologías de comunicación que favorezcan el ahorro en el consumo de energía.

Así, se debe tener en cuenta que los nodos finales suelen disponerse en las calles, recintos o en otros puntos estratégicos de la ciudad, pero que rara vez se encuentran en un edificio o en un hogar para poder ser alimentados, como un electrodoméstico más. Por ello, es necesario valorar todos los aspectos relacionados con el consumo de potencia de los nodos finales, y la tecnología de comunicación utilizada es uno de los aspectos fundamentales en este sentido. Por otro lado, los *gateways* de las soluciones analizadas si suelen estar ubicados en puntos donde pueden ser alimentados fácilmente sin hacer uso de baterías, ya que no necesitan estar en la zona a monitorizar como en el caso de los nodos finales. Por lo tanto, la comunicación *gateway*-servidor no es tan determinante en términos de consumo de potencia como lo es la comunicación nodo final-*gateway*.

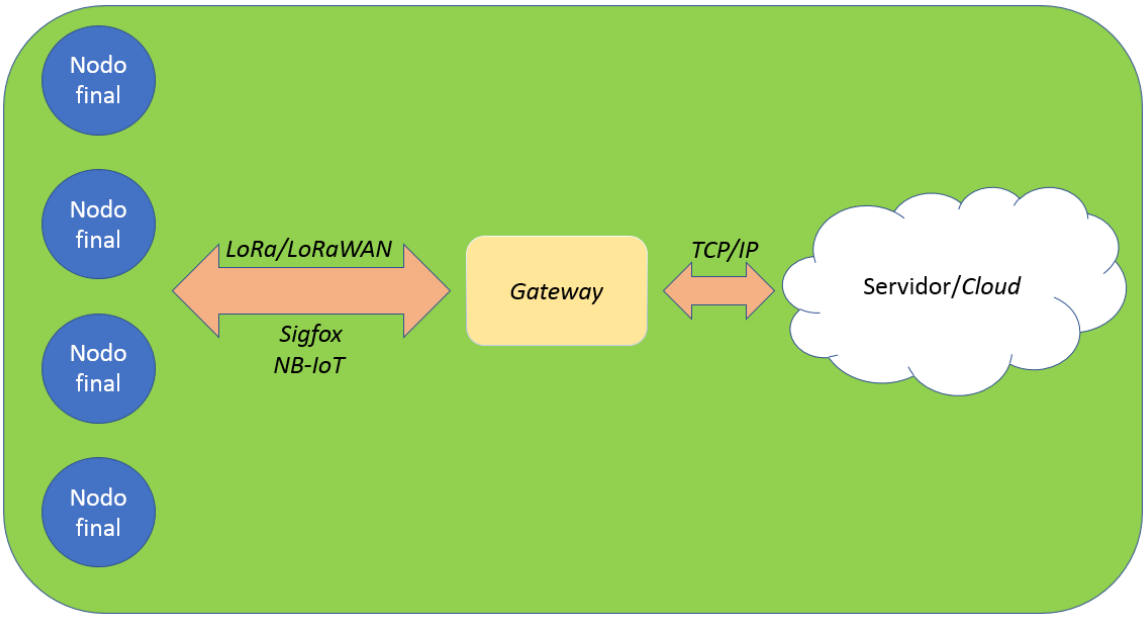


Figura 2.16: Arquitectura habitual de las soluciones Smart Cities.

La comunicación entre el *gateway* y el servidor suele ser vía TCP/IP, como se muestra en la Figura 2.16, para transmitir con un ancho de banda elevado todos los datos que se han recibido desde los nodos finales.

Con respecto a las comunicaciones entre los nodos finales y el *gateway,* existen diferentes alternativas. Dependiendo de las especificaciones del proyecto y de la aplicación, se puede optar por una tecnología u otra, pero todas ellas se caracterizan por ser de bajo consumo de potencia, largo alcance y bajo coste. Las más extendidas hasta el momento son las tres que se muestran en la Figura 2.16:

- NB-IoT: Es la gran apuesta de las operadoras para este sector [23] y representa versiones reducidas del 4G. Es una tecnología optimizada para aplicaciones de bajo consumo de datos y energía.

- Sigfox: *SigFox* es una compañía francesa fundada en 2009 que proporciona el servicio de red de cobertura amplia de bajo consumo (LPWAN), es inalámbrica y fue creada para que funcione e interactúe con dispositivos de bajo consumo energético. Un dispositivo de *Sigfox* puede enviar hasta 140 mensajes al día de 12 bytes como máximo. [24]

- LoRa/LoRaWAN: Es una tecnología de comunicación de tipo LPWAN. Esta tecnología fue ideada por la *startup* francesa *Cycleo*, y posterormente fue adquirida por la empresa *SemTech*, quien la desarrolló y patentó. Actualmente siguen manteniendo la patente, si bien es la fundación *LoRa Alliance* [25] quien lleva su desarrollo y su evolución. Es la tecnología de comunicación utilizada en este Trabajo de Fin de Grado, por lo que posteriormente se presentará con profundidad.

Capítulo 3. LoRa/LoRaWAN

3.1 Tecnología de comunicación LoRa/LoRaWAN

La solución *Smart Parking* desarrollada es una solución inalámbrica, tanto a nivel de comunicación como de alimentación, por lo tanto, el nodo final debe prestar especial atención al consumo de potencia y al alcance de las comunicaciones.

El consumo de potencia es un aspecto importante, ya que, al no contar con alimentación cableada de la red eléctrica, lo normal es hacer uso de baterías, y para alargar la duración de estas, se requiere el uso de técnicas de bajo consumo.

Por otra parte, el alcance de las comunicaciones entre el nodo final y el *gateway* debe ser considerable, de esta manera se permite mayor movilidad a la hora de ubicar el nodo final en la plaza de estacionamiento.

En consecuencia, se requiere de una tecnología de comunicación de bajo consumo de potencia y de largo alcance. Un tipo de red que cumple con estos requisitos, son las LPWAN. Actualmente, estas redes se encuentran en pleno auge y siguen desarrollando ayudas que agilicen su implementación en soluciones *Smart City*. Las tecnologías de este tipo que más importancia tienen hoy en día son *Sigfox* y LoRa/LoRaWAN.

En este TFG se hace uso de LoRa/LoRaWAN, y por ello, a continuación, se estudia detalladamente esta tecnología de comunicación.

3.1.1 Introducción

LoRa y LoRaWAN son especificaciones de redes de elevado alcance y baja potencia. Están pensadas para utilizarse en el ámbito de IoT, y con ello, en el desarrollo de las *Smart Cities.* Esta tecnología fue ideada por la *startup* francesa Cycleo, y posteriormente fue adquirida por la empresa SemTech, quien la desarrolló y patentó. Actualmente siguen manteniendo la patente, pero es la fundación *LoRa Alliance* quien lleva su desarrollo y su evolución [26].

LoRaWAN representa una red de baja potencia y largo alcance. Es decir, una red del tipo LPWAN (*Low Power Wide Area Network*).

Se debe diferenciar desde un primer momento qué es LoRa y qué es LoRaWAN, porque no son lo mismo, habiendo puntualizaciones que se deben comentar, por ejemplo, el nivel o capa en el que desarrollan su comunicación, o qué misión tiene cada una dentro de una red LoRa/LoRaWAN.

Por una parte, LoRa solo trabaja a nivel de la capa de enlace y su uso está destinado a la comunicación *Peer to Peer* (P2P) entre diferentes nodos, mientras que en LoRaWAN se añade la capa de red o Internet, a partir de la cual se permite conectar con cualquier estación base que se encuentre conectada a la nube [27].

Una definición que ayuda a entender la arquitectura que utiliza LoRa y el papel que desempeñan LoRa y LoRaWAN dentro de ella, es la que se muestra en la referencia [28]: "LoRaWAN se encarga de unir diferentes dispositivos LoRa gestionando sus canales y parámetros de conexión: canal, ancho de banda, cifrado de datos, etc." Por tanto, se puede afirmar que los dispositivos LoRa son los elementos que permiten incluir la electrónica entendida como sensores o actuadores finales como elementos de la red LoRaWAN. Dicho con otras palabras, LoRa representa la conexión física entre los nodos de la red, y LoRaWAN es la propia red que interconecta y gestiona dichos nodos, teniendo en cuenta diferentes parámetros propios de una red de comunicación.

3.1.2 Comparación con otros estándares

Para comparar diferentes tecnologías de comunicación es necesario enmarcar bien para qué especificaciones se usa un estándar u otro. La primera comparativa que se realiza en este documento pretende ubicar la tecnología en términos de alcance y consumo de potencia. En la Figura 3.1 [6] se muestran los estándares más usados internacionalmente junto con la tecnología LoRa/LoRaWAN.

Como se puede ver en la Figura 3.1, dentro de las tecnologías de corto alcance se tienen, entre otras, *WiFi* y *Bluetooth*, siendo la primera una red de área local, y la segunda de área personal, pero en comparación con las otras dos que se muestran en la gráfica, se consideran de bajo alcance. Entre las tecnologías de largo alcance se pueden observar las redes móviles y LoRaWAN.

Las redes móviles y *WiFi* soportan mayor transferencia de datos y sus velocidades de transmisión son superiores a las de *Bluetooth* y LoRa/LoRaWAN, pero por esta misma razón consumen mucha más potencia. En el caso de LoRa/LoRaWAN se consigue el equilibrio perfecto entre una red de largo alcance y de bajo consumo de potencia. El alcance y la tasa binaria dependen de factores como el valor del parámetro *Spreading Factor* (SF) o el ancho de banda utilizado, por lo tanto, no son valores fijos, pero aun así superan en alcance a los estándares de *WiFi* y *Bluetooth*.

El reducido consumo de potencia y su elevado alcance, hace que LoRa/LoRaWAN se postule como una tecnología puntera en el ámbito de IoT y *Smart City*.

Sin embargo, dentro de las LPWAN existen otros estándares que están cobrando fuerza en el mercado e intentan posicionarse, siendo los principales los siguientes:

- Sigfox: Junto con LoRa/LoRaWAN es una de las tecnologías de comunicación LPWAN más importante. Cobra $1 por dispositivo y año conectado, es decir es propietaria y presta servicios bajo licencia de uso. Toda su tecnología es privada, y se basa en ancho de banda ultra-estrecho. Un dispositivo de *Sigfox* puede enviar hasta 140 mensajes al día de 12 bytes como máximo. Su mapa de cobertura es mayor que el de LoRa/LoRaWAN, ya que cuenta con conectividad en toda Francia, la

mayoría de los territorios de España, y algunos puntos de otros países como Reino Unido. LoRa/LoRaWAN solo tiene cobertura en Francia por ahora.

- Cat NB-IoT: Representa una evolución o adaptación de la tecnología *Long Term Evolution* (LTE) al ámbito de IoT. Cuenta con un ancho de banda y un consumo superior a los de LoRa/LoRaWAN y *Sigfox*. Aún está por ver su futuro para aplicaciones de requerimiento energéticos muy reducidos. [23]

Figura 3.1: Comparativa por alcance y consumo de potencia.

En líneas generales, las diferentes ventajas de LoRaWAN frente al resto de redes del tipo LPWAN son [29].

- La tasa binaria puede ir desde 0,3 kbps hasta 5 kbps para un ancho de banda de 125 KHz, lo que permite un ahorro en la vida útil de la batería y una duración de transmisión superior.

- La comunicación es bidireccional e ilimitada.

- Seguridad a nivel de red, de aplicación y de dispositivo.

- Localización sin uso de GPS al usar multilateración *Time Difference of Arrival* (TDoA).

- Se ofrece una gran variedad de *gateways* según el uso que se vaya a dar a la red.

- Posibilidad de crear redes públicas o privadas.

- Utilización de *Adaptative Data Rate* (ADR) para la optimización de la red, permitiendo que puedan incluirse más nodos, disminuyendo la tasa binaria de otros nodos.

3.1.3 LoRa

LoRa representa la capa física o la modulación inalámbrica utilizada para crear una conexión de comunicación de larga distancia [26]. Esta capa física se basa en la modulación de espectro ensanchado. El procedimiento que se sigue para esta modulación consiste en primer lugar en codificar la señal con una secuencia de alta frecuencia, propagando la señal con un ancho de banda superior, y reduciendo con ello el consumo de energía, además de aumentar la resistencia a interferencias electromagnéticas, tal y como se indica en la Figura 3.2 [30].

LoRa soporta seis valores de factores de propagación *Spreading Factor* (SF7 - SF12) y tres anchos de banda diferentes (125 kHz, 250 kHz, 500 kHz). Más adelante, en el apartado de características técnicas, se indicarán las implicaciones que conlleva la elección del factor de propagación, y también que al ancho de banda más utilizado es el de 125 KHz. Los factores de propagación y anchos de banda permitidos están definidos por las agencias reguladoras, en el caso europeo, por ETSI *(European Telecommunications Standards Institute)*.

Debido a la tecnología de espectro de propagación utilizada, los mensajes con diferentes velocidades de datos son ortogonales y no interfieren unos con otros.

El esquema LoRa se basa en una variante de *Spread Spectrum Modulation* (SSM) llamada modulación de *Chirp Spread Spectrum* (CSS). CSS codifica los datos con un *chirp*, que es esencialmente una señal sinusoidal de frecuencia modulada en banda ancha que aumenta o disminuye con el tiempo [30].

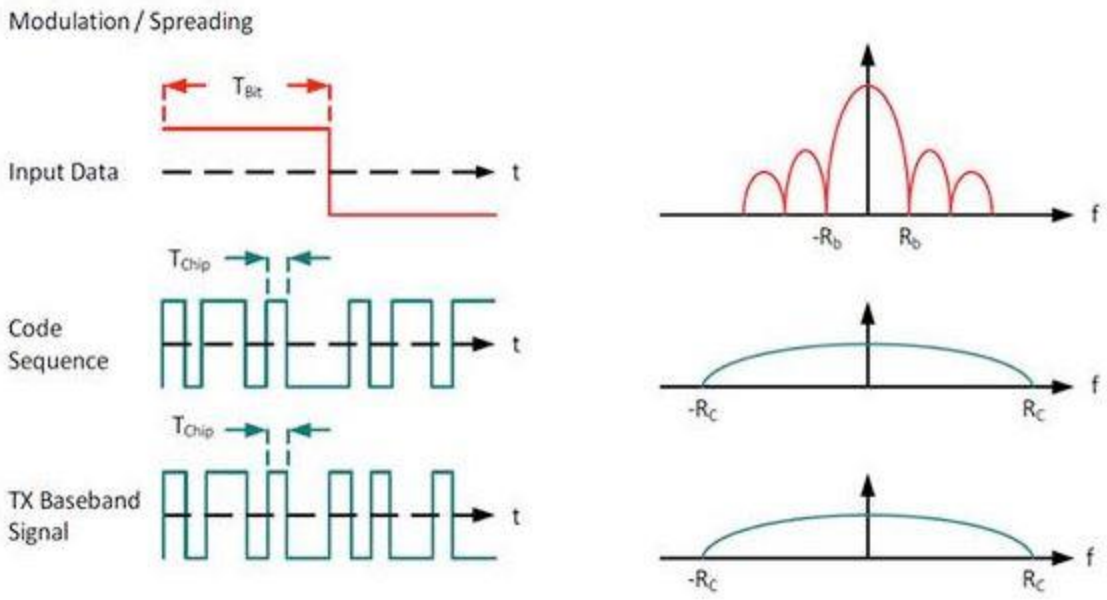


Figura 3.2: Modulación de espectro ensanchado.

En definitiva, LoRa es una comunicación al nivel de la capa física y no atiende a parámetros propios de la capa de red, como en el caso de LoRaWAN.

3.1.4 LoRaWAN

LoRaWAN define el protocolo de comunicación y la arquitectura del sistema para la red mientras que a nivel físico, LoRa permite crear la conexión para una comunicación a larga distancia. Las tareas de LoRaWAN resultan determinantes para el buen funcionamiento de la red, pues en función del protocolo usado y la arquitectura de la red adoptada, se determina la duración de vida de las baterías de los nodos, la capacidad de comunicación de la red, la calidad del servicio, la seguridad, y la variedad de aplicaciones que puede ofrecer la red.

En otras palabras, la especificación LoRaWAN define la capa de control de acceso al medio (MAC) para la red del tipo LPWAN. Según el modelo *Open Systems Interconnection* (OSI) de *International Organization for Standardization* (ISO), esta capa se ubica por

encima de la capa física que define LoRa. Esta capa de acceso al medio se desarrolla de forma abierta por la entidad LoRa-Alliance, a diferencia de LoRa, que es una capa física con propiedad de la empresa SemTech.

3.1.5 Topología

Una vez se han introducido LoRa y LoRaWAN, ya se pueden exponer los campos que se deben incluir en cualquier estudio sobre una determinada tecnología de comunicación, como, por ejemplo, la topología utilizada para la transmisión y recepción de datos.

En el caso de la red LoRaWAN, existe una topología claramente definida por los desarrolladores de esta tecnología, la disposición en estrella. Esta topología se divide en dos componentes fundamentales, los nodos finales y el nodo central. Este nodo central, que también se denomina como *gateway*, es el punto principal de la red, pues es donde confluyen todos los datos procedentes de los distintos nodos finales, y además también es donde se redirige la información que se desea enviar a cada nodo.

Por tanto, en una red LoRa/LoRaWAN, todos los nodos finales se conectan punto a punto con este *gateway*, y esta comunicación la realizan bajo la tecnología que ofrece LoRa. Por otra parte, el *gateway* es el que proporciona a la nube o a la plataforma de todos los datos recibidos por parte de los nodos sensoriales, y allí es donde se procesan los datos y se ejecutan acciones que se trasladan a los nodos actuadores, de la misma manera en que se transmitió la información desde los sensores, pero en este caso se tiene el flujo de datos en sentido inverso. La comunicación entre el *gateway* y la nube escapa a la comunicación LoRa/LoRaWAN, y lo normal es encontrar comunicaciones basadas en TCP/IP.

Sin embargo, la red LoRaWAN, no tiene que estancarse en varios nodos asociados a un *gateway,* pudiendo escalar esta topología a una red de redes en estrella o una red en estrella extendida. Esta topología también recibe el nombre de estrella de estrellas y en ella, algunos nodos de una subred en estrella se conectan como un nodo más de otra subred, lo cual puede repetirse más veces en la estructura de la red. En la Figura 3.3 se representa este tipo de red de redes de forma gráfica.

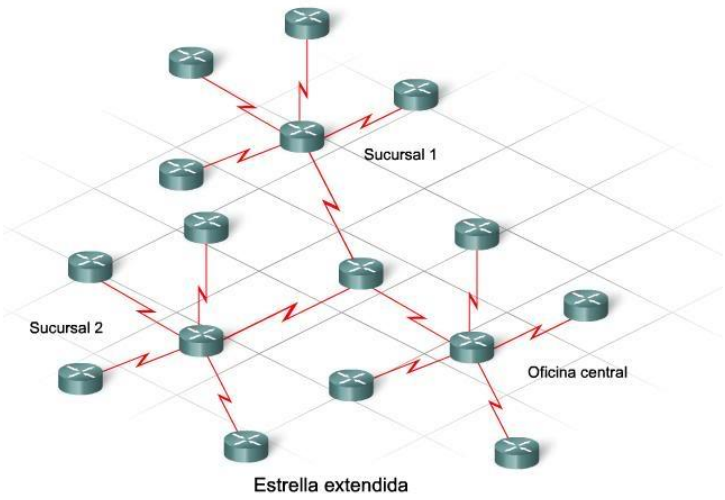


Figura 3.3: Topología: "Estrella de estrellas".

En lo que respecta a las comunicaciones que esta topología puede soportar, se tienen diferentes niveles de comunicación. Resumiendo lo comentado anteriormente, y añadiendo algún apunte más, los distintos tipos de comunicaciones que los elementos de la red pueden soportar, son los siguientes:

- Los nodos finales de una misma subred en estrella pueden comunicarse entre ellos, a través de un *gateway* usando LoRa.

- Pueden comunicarse con la nube a través de un *gateway*, siguiendo primero una comunicación nodo-*gateway* tipo LoRa, y posteriormente *gateway*-nube usando Ethernet, 3G/4G u otras tecnologías con acceso a Internet.

- Los nodos finales de distintas subredes en estrella pueden comunicarse entre sí, a través de los *gateways* de sus respectivas subredes.

- Una utilidad muy importante que permite LoRa/LoRaWAN es la de realizar multidifusión. La multidifusión suele utilizarse para actualizar nodos o distribuir mensajes masivos de emergencia.

En resumen, la mayor parte de las comunicaciones en este tipo de redes son bidireccionales, aunque hay casos excepcionales como el de la multidifusión.

Otro aspecto que se debe estudiar es el tipo de nodos finales que se pueden tener en una red LoRaWAN, y su forma de interactuar con ella. Existen tres tipos de nodos, todos ellos bidireccionales, ordenados según el consumo de potencia y su interacción con la red.

- Clase A: Son los nodos de más baja potencia y por cada subida de datos desde el nodo final le siguen dos ventanas de recepción para la descarga de datos. Por lo tanto, el *slot* programado para la transmisión se adecua a las propias necesidades de comunicación, con una pequeña variación basada en un tiempo aleatorio (sistema de slot del protocolo ALOHA). Esta clase de nodo final es ideal para aquellos casos donde un sensor debe enviar un dato puntual cada cierto tiempo y que no suele funcionar como receptor de una comunicación. En caso de que se desee enviar información a este nodo final desde el servidor, se debe esperar a que este realice una transmisión, y que tras completarla se habiliten las ventanas de recepción. En la Figura 3.4 [31] se muestra el eje temporal que sigue un nodo final de clase A. Se debe apuntar también que en este tipo de nodos finales no existe restricción en términos de latencia y que cualquier dispositivo puede mostrarse como de clase A.

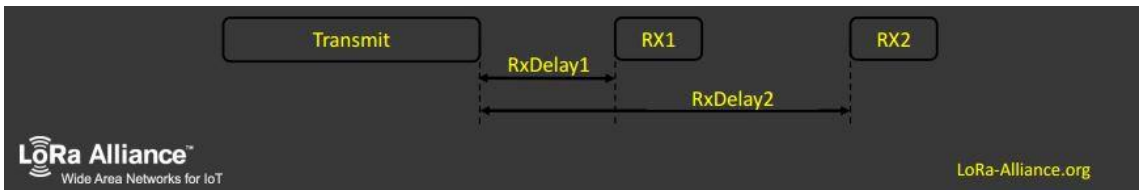


Figura 3.4: Estudio temporal de un nodo final de clase A.

- Clase B: Para los nodos finales de tipo B, aparte de las ventanas de recepción que tienen los de clase A, se añaden ventanas de recepción en momentos determinados previamente establecidos. Cuando ha llegado el momento establecido, desde la puerta de enlace o *gateway* se envía un *Beacon* (un paquete que se suele usar para mantener activo un

determinado receptor). Con esto lo que se consigue, aparte de habilitar la ventana de recepción del nodo final, es que el servidor sepa que el nodo final está en modo "escucha". En la Figura 3.5 [31] se muestra el eje temporal que sigue un nodo final de clase B. En este caso la latencia de descarga es controlada.

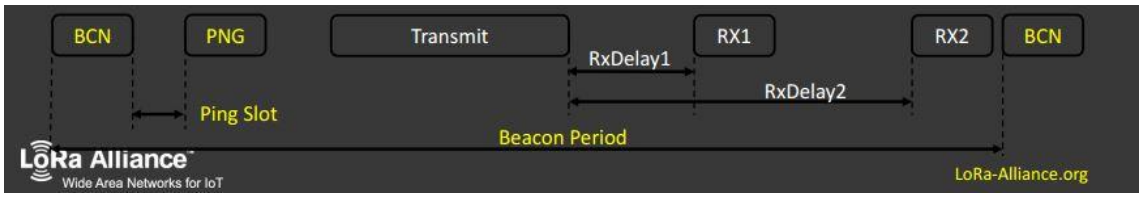


Figura 3.5: Estudio temporal de un nodo final de clase B.

- Clase C: Para este tipo de nodos, la ventana de recepción está prácticamente siempre abierta, salvo cuando se está transmitiendo, al estar en un modo de transmisión semidúplex, en el que la comunicación es bidireccional, pero no se puede transmitir y recibir información simultáneamente. El inconveniente de este tipo de nodos es que se consume mucha más potencia que en los otros dos casos. En la Figura 3.6 [31] se muestra el eje temporal que sigue un nodo final de clase C. En este último tipo no hay latencia, y solo lo soportan los dispositivos que puedan mantenerse a la escucha continuamente.

Figura 3.6: Estudio temporal de un nodo final de clase C.

En la Figura 3.7 [31] se muestra a nivel de capas la arquitectura que sigue LoRa, y en ella se puede ver cómo desde la capa en la que trabaja LoRaWAN se atiende de forma diferente a los nodos finales de diferentes clases. En la propia Figura 3.7 se enuncia esta atención a esos nodos finales como MAC *options*.

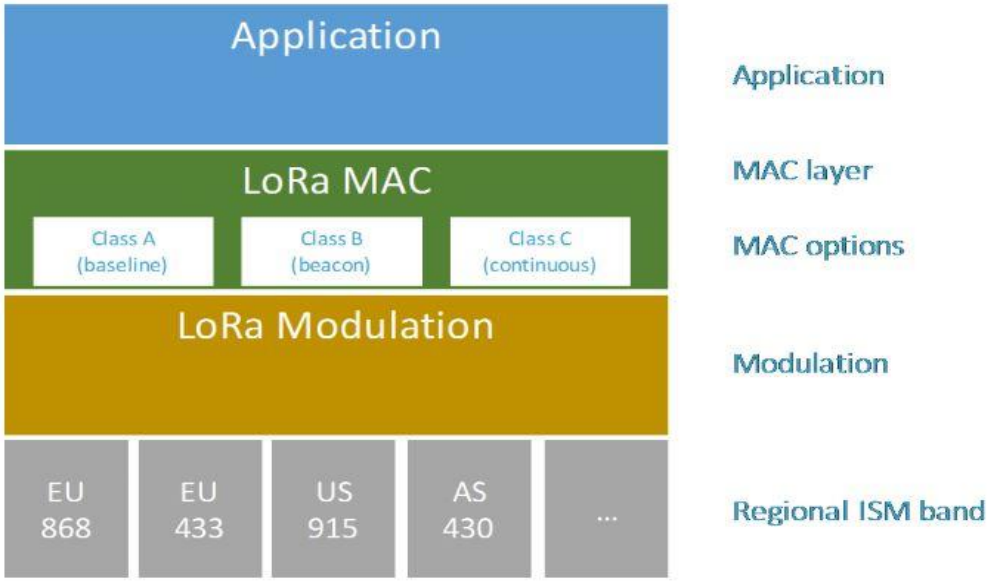


Figura 3.7:Arquitectura LoRa/LoRaWAN.

3.1.6 Características técnicas

En este apartado se presentarán las características técnicas de la tecnología LoRa/LoRaWAN.

3.1.6.1 Modulación

LoRa representa la capa física o la modulación inalámbrica utilizada con el fin de crear conexiones para comunicaciones de larga distancia. También se conoce que muchos de los sistemas inalámbricos usados hasta el momento se basan en la modulación *Frequency Shift Keying* (FSK) porque resulta una modulación muy eficiente para obtener un bajo consumo. Sin embargo, LoRa se basa en la modulación CSS que mantiene la característica de bajo consumo que aporta la modulación FSK, pero amplía significativamente la distancia de comunicación. Durante décadas ya se usaba esta modulación en aplicaciones militares o espaciales debido a la larga distancia que es capaz de soportar y la robustez que posee ante interferencias. La diferencia es que LoRa es la primera implementación de esta tecnología comercializada al público en general, y con un coste reducido. No obstante, en el nivel físico de la comunicación también se puede usar la modulación FSK en este tipo de redes LoRaWAN.

En el eje temporal para esta modulación se muestra un *chirp* como el de la gráfica mostrada en la parte derecha de la Figura 3.8. Como se observa, el *chirp* incrementa su

frecuencia en el tiempo, por lo que se estaría ante un *Up-chirp*, mientras que en el caso contrario, si se decrementa la frecuencia a medida que avanza el tiempo, se estaría ante un *Down-chirp*. Estos dos casos se muestran en la Figura 3.9. También se puede obtener el valor nulo para el *chirp*. La diferencia entre la frecuencia inicial del *chirp* y la frecuencia final es una buena aproximación del ancho de banda B del pulso del *chirp* en el espectro, como se representa en la Figura 3.8 [32], esta vez en la gráfica izquierda de la ilustración.

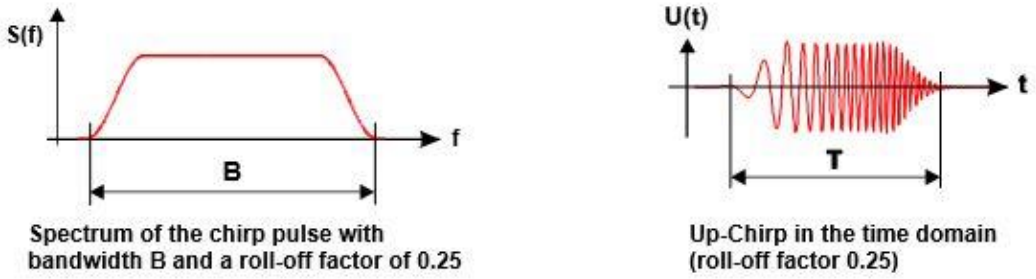


Figura 3.8: Chirp en el espectro y en el eje temporal.

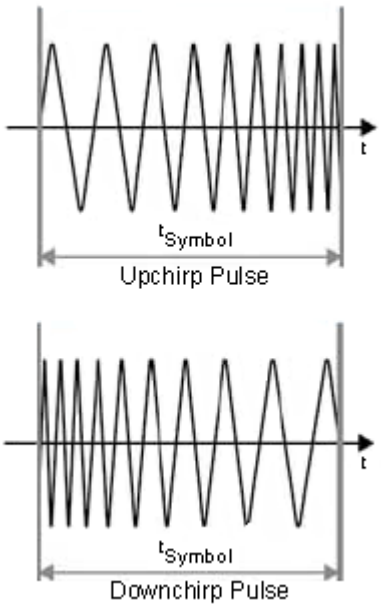


Figura 3.9: Pulsos Upchirp y Downchirp.

Sobre esta modulación hay que comentar algunos aspectos que resultan imprescindibles para poder entender por qué se utiliza en LoRa.

Así, los beneficios de CSS se manifiestan de forma clara cuando el ancho de banda (B) del pulso del *chirp* es mucho mayor que la tasa binaria (R). Es decir, interesa que B >> R. De aquí procede el interés del ensanchamiento en la frecuencia.

Por otro lado, la duración del pulso del *chirp* se puede elegir libremente, y así conseguir un producto B·T elevado que aporte robustez en la señal [32]. Es una modulación que favorece los sistemas en los que se varía la distancia y la tasa binaria rápidamente. Especialmente para aquellos sistemas de banda ancha o de ultra banda ancha en las que el ancho de banda es muy superior a la tasa binaria.

La combinación de LoRa con esta modulación ofrece una solución para cubrir la demanda de una comunicación de baja potencia, de largo alcance y asequible económicamente.

3.1.6.2 Estudio de parámetros asociados

La comunicación entre los nodos finales y el *gateway* se lleva a cabo en diferentes canales de frecuencia y con diferentes tasas binarias. Posteriormente se verá que se debe llegar a un compromiso entre la tasa binaria, el alcance de la transmisión y la duración del mensaje [6], pero por ahora se indica que esta tasa binaria es cambiante. LoRa puede mantener tasas binarias desde 300 bps hasta 5 kbps para un ancho de banda de 125 KHz. El ancho de banda puede llegar a ser de 250KHz, y para este caso la tasa binaria máxima alcanza los 11 kbps. También se puede escoger un ancho de banda de 500 KHz, si bien es el menos usado. En este rango, LoRa se adecua perfectamente a los márgenes en los que la modulación CSS proporciona sus mejores funcionalidades, ya que el ancho de banda es muy superior al régimen binario.

La red LoRa usa el modelo *Adaptative Data Rate* (ADR) para obtener un equilibrio entre la optimización de la vida útil de la batería y las prestaciones que el sistema es capaz de proporcionar. ADR organiza las tasas binarias individuales de cada nodo que se va a comunicar y las salidas en los canales de frecuencia que se van a usar.

Teóricamente los nodos finales pueden transmitir en cualquier momento y por cualquier canal que esté disponible, siempre y cuando respete ciertas restricciones. Una de estas restricciones es que los nodos finales deben cambiar de canal de transmisión de forma

pseudo-aleatoria en cada transmisión que realicen. El beneficio que se obtiene al cumplir este requisito es que se consigue una variedad en frecuencias, y con esto, un sistema más robusto ante interferencias.

Otra de las restricciones se centra en la legislación local del lugar en el que se vaya a usar esta tecnología. Para el caso de este TFG, se debe considerar la restricción aplicable al marco europeo. LoRaWAN usa franja libre del espectro, concretamente la banda *Industrial Scientific and Medical bands* (ISM). ETSI ha limitado el acceso a las bandas de 868 MHz y 433 MHz. LoRaWAN trabaja en concreto en la banda de 868 MHz ISM de Europa, y en esta franja el nodo final debe respetar el ciclo de trabajo máximo en referencia a la sub-banda utilizada y las regulaciones locales (1% para nodos finales y 10% para *gateways*) [26]. Otra restricción está relacionada con la potencia transmitida por los nodos, y que no debe exceder los 14 dBm o los 25mW.

En la referencia [26] se define el ADR como el proceso por el cual la red ordena a cada nodo la tasa binaria que debe adoptar, que será un valor suficiente para cumplir con la transmisión requerida. Se espera que, en un futuro no muy lejano, también se indique a cada nodo con qué potencia ha de transmitir.

Para poder ahondar sobre el funcionamiento de ADR es necesario introducir el concepto de *Spreading Factor* (SF). El *Spreading Factor* es un valor que indica la relación entre la tasa binaria y la tasa de símbolos, o lo que es lo mismo del período de bit y el período de símbolo. En la Ecuación 3.1 se muestra dicha expresión.

$$SF = \frac{T_{Símbolo}}{T_{Bit}} = \frac{R_{Bit}}{R_{Símbolo}} \qquad (3.1)$$

Aprovechando que se ha expuesto ya el concepto de *Spreading Factor*, en las siguientes líneas se mostrarán, mediante expresiones, todas las variables que intervienen en el cálculo de parámetros que se usarán en el desarrollo de la funcionalidad del nodo a la plataforma HW/SW propuesta en este TFG. Como se muestra en la Ecuación 3.2, el ancho de banda es la inversa del período del *chirp* [33]. Por otra parte, el período de símbolo guarda relación con los dos parámetros anteriores, tal y como se muestra en la Ecuación 3.3. Por último, en la Ecuación 3.4 se adjunta la expresión para el cálculo de la tasa binaria.

$$BW = \frac{1}{T_{Chirp}} \qquad (3.2)$$

$$T_{Símbolo} = 2^{SF} \cdot T_{Chirp} \qquad (3.3)$$

$$R_{Bit} = SF \cdot R_{Símbolo} = SF \cdot \frac{1}{T_{Símbolo}} = \left(\frac{SF}{2^{SF}}\right) \cdot BW \qquad (3.4)$$

En la Figura 3.10 [33] se muestra un ejemplo sobre la relación de todos estos parámetros. De esta manera, se aprecia la importancia de determinar el valor correcto de SF según la tasa de bits necesaria para transmitir un dato determinado. En esta figura se aprecia cómo la tasa de bits aumenta cuando se disminuye el *Spreading Factor*.

		$T_{Symbol} = 2^{SF}/BW$		ohne FEC	4/5 FEC (CR=1)	
BW / kHz	SF	T_{symbol} / ms	R_{symbol} / Hz	R_{bit} / bps	R_{bit} / bps	Sensitivity
125	7	1.024	976.56	6835.94	5468	S_{ref} = -125 dBm
125	8	2.048	488.28	3906.25	3125	- 2.4 dB
125	9	4.096	244.14	2197.27	1757	- 4.9 dB
125	10	8.192	122.07	1220.70	976	- 7.5 dB
125	11	16.384	61.04	671.39	537	- 10 dB
125	12	32.768	30.52	366.21	292	- 12.7 dB

Figura 3.10: Ejemplo para el estudio de los parámetros de transmisión.

Además, en la Figura 3.10 se puede apreciar cómo mejora la sensibilidad a medida que se aumenta el SF. En el ejemplo se adjunta la tasa binaria para cuando no hay corrección de errores, y también para cuando se usa *Forward Error Correction* (FEC) como método de detección y corrección de errores.

En la Figura 3.11 [6] se muestra otra relación con más parámetros que varían en función del SF escogido, como el caso del ya comentado *bitrate*, el alcance y el tiempo de transmisión.

Spreading factor (at 125 kHz)	Bitrate	Range (indicative value, depending on propagation conditions)	Time on Air (ms) For 10 Bytes app payload
SF7	5470 bps	2 km	56 ms
SF8	3125 bps	4 km	100 ms
SF9	1760 bps	6 km	200 ms
SF10	980 bps	8 km	370 ms
SF11	440 bps	11 km	740 ms
SF12	290 bps	14 km	1400 ms
(with coding rate 4/5 ; bandwidth 125Khz ; Packet Error Rate (PER): 1%)			

Figura 3.11: Realación SF-BitRate-Alcance-Tiempo de transmisión.

3.1.6.3 Estructura de paquetes

3.1.6.3.a Paquete físico: LoRa

LoRa emplea dos tipos de formato de paquete: explícito e implícito. El modo explícito incluye una cabecera que contiene información acerca del número de bytes, codificación de error, y si se usa o no Código de Redundancia Cíclica (CRC) para la detección de errores. El modo implícito, por el contrario, no contiene cabecera. A continuación, en la Figura 3.12 [34] se muestra el formato del paquete LoRa que consta principalmente de tres partes fundamentales: Preámbulo, cabecera opcional y área de datos.

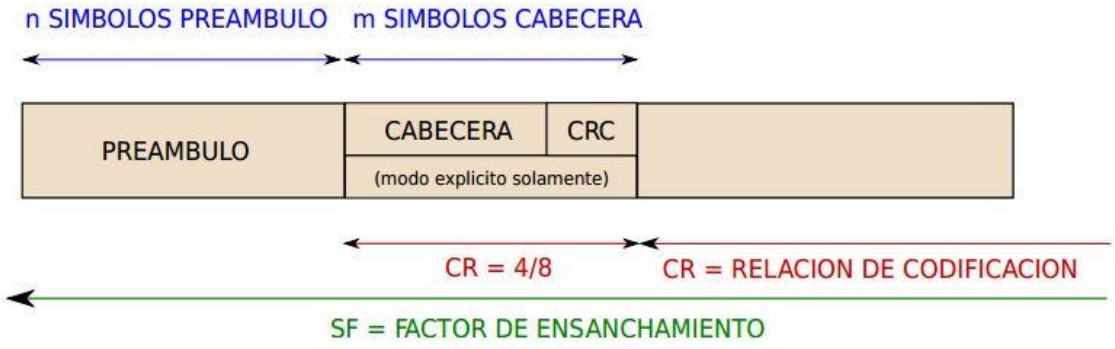


Figura 3.12: Formato de paquete LoRa.

El objetivo del preámbulo es sincronizar al receptor con el flujo de datos entrante. Por defecto, la longitud del paquete es de 12 símbolos de longitud, pero este valor es modificable mediante el registro *PreambleLength* que permite configurar su valor desde 6 a 65535, obteniendo así la posibilidad de tener una longitud de símbolos desde 6+4 hasta 65535+4, dónde se alcanza el límite y se entiende que se ha llegado a la sobrecarga fija de los datos del preámbulo. Como se puede apreciar por el límite máximo que acepta este registro, se puede determinar un campo de preámbulo de longitud amplia.

Un apunte importante es que, tanto en el receptor como en el transmisor, la configuración de la longitud del preámbulo debe ser idéntica, pues el receptor realiza un proceso de detección de preámbulo que se reinicia periódicamente, y manteniendo la misma longitud se facilita este proceso. En caso de que no se conozca la longitud, o esta varíe, se recomienda desde la empresa que desarrolla LoRa, SemTech, que se programe la longitud de preámbulo del receptor con el mayor valor posible. [34]

Para seleccionar un tipo de paquete implícito o explícito se modifica el valor del bit *ImplicitHeaderModeOn* que se encuentra en el registro *RegModemConfig1* [34]. Por defecto se tiene el tipo de paquete explícito.

La cabecera en el modo explícito proporciona información sobre diferentes aspectos del área de datos, como la longitud del área de datos (en bytes), el código de corrección de errores, y la presencia o no de un código de verificación CRC opcional de 16 bits. La cabecera se transmite con un código de corrección máximo, de 8/4, y la misma cabecera contiene su propio código de verificación CRC para desechar cabeceras inválidas. Para la inclusión del campo CRC se debe activar el bit *RxPayloadCrcOn* en el registro *RegModemConfig1* en el transmisor, mientras que en el receptor una vez se ha recibido el área de datos, se debe revisar el valor del bit *CrcOnPayload* en el registro *RegHopChannel*. Si el bit *CrcOnPayload* se encuentra al valor “1“, el usuario deberá revisar el *flag* IRQ *PayloadCrcError* para asegurarse que el CRC es válido. Si el bit *CrcOnPayload* se encuentra al valor “0”, el área de datos no lleva CRC, por lo que el *flag* IRQ*PayloadCrcError* no se debería activar aún en el caso de que el área de datos presentase errores. [34]

El uso de la cabecera implícita se justifica en los casos en las que interesa reducir el tiempo de transmisión. En este tipo de formato de paquete, el área de datos, la

codificación de error, y la presencia de código de verificación CRC, son fijos y conocidos previamente. Para que esta información sea conocida hay que configurarla manualmente en el transmisor y en el receptor.

Existe un caso en el que solo se puede usar este formato de paquete, y es cuando el valor del *Spreading Factor* es 6. La activación del CRC en la cabecera implícita es fija, debiendo estar activado siempre el bit *RxPayloadCrcOn* en el registro *RegModemConfig1*, tanto en el transmisor como en el receptor [34].

Respecto al área de datos, este es un campo de longitud variable que contiene los datos codificados con el error especificado en la cabecera explícita, o en el registro de configuración en caso de cabecera implícita, siendo la verificación CRC opcional.

Cuando se utilizan valores de *Spreading Factor* elevados, la duración de transmisión de un paquete es bastante alargada, y para estos casos existe un bit que al activarlo mejora la robustez de la transmisión mediante variaciones en la frecuencia durante la transmisión y recepción del paquete. Este bit es *LowDataRateOptimize* y su uso es recomendable para velocidades de transmisión bajas, siendo obligatorio cuando la duración de un símbolo excede los 16 ms. Si se usa este bit, hay que configurarlo tanto en el transmisor como en el receptor.

Con el objetivo de calcular la duración de la transmisión de un mensaje, se presentan a continuación diferentes expresiones que permiten obtener la duración de cada campo, y por tanto, la duración del paquete completo. Lo primero que se debe indicar es que la duración de un símbolo es inversamente proporcional a la tasa de símbolos, tal y como se muestra en la Ecuación 3.5 [34].

$$T_S = \frac{1}{R_S} \qquad (3.5)$$

La Ecuación 3.6 muestra la duración del preámbulo en función del número de la longitud de este campo. Dicha longitud se puede extraer de los registros *RegPreambleMsb* y *RegPreambleLsb* [34].

$$T_{Preámbulo} = (n_{Preámbulo} + 4{,}25) \cdot T_S \quad (3.6)$$

Por otro lado, la duración del área de datos viene determinada por la Ecuación 3.7 [34].

$$T_{Datos} = n_{Datos} \cdot T_S \qquad (3.7),$$

donde n_{Datos} es el número de símbolos del área de datos y se calcula a partir de la Ecuación 3.8 [34].

$$n_{Datos} = 8 + max\left(ceil\left[\frac{8PL - 4SF + 28 + 16CRC - 20IH}{4(SF - 2DE)}\right](CR + 4), 0\right) \qquad (3.8),$$

en la que:

- *PL* es el número de bytes del área de datos: [1 – 255].
- *SF* es el Spreading Factor: [6 – 12].
- *IH=0* cuando la cabecera está activada, y IH = 1 cuando no lo está.
- *DE=1* cuando *LowDataRateOptimize*=1, y DE=0 cuando el bit *LowDataRateOptimize* no está activado.
- *CR* es el *coding rate*: [Adquiere el valor 1 cuando es 4/5, y 4 cuando es 4/8].

Finalmente, se muestra en la Ecuación 3.9 [34] la duración total del paquete, que es la suma de la duración del preámbulo y de la duración del área de datos.

$$T_{Paquete} = T_{Preámbulo} \cdot T_{Datos} \qquad (3.9)$$

3.1.6.3.b Paquete MAC: LoRaWAN

LoRaWAN usa el formato de paquete físico que se indicó en el apartado Paquete Físico: LoRa, si bien existen ciertas puntualizaciones que se deben comentar, como que en comunicación ascendente en LoRaWAN la inclusión de la cabecera y del código de

verificación CRC son obligatorios. Por lo tanto, se imposibilita directamente la opción de usar un *Spreading Factor* de valor 6, pues como se explicó en el apartado anterior, existe un caso en el que solo es posible usar el formato de paquete implícito, y es cuando el valor del *Spreading Factor* es 6. Es decir, con SF=6 no se incluye cabecera, y esto va en contra de los requisitos del paquete de LoRaWAN, por lo que no se permite un *Spreading Factor* de valor 6. Otro apunte es que los paquetes descendentes llevan cabecera, pero no CRC. El formato del paquete LoRaWAN se muestra en la Figura 3.13 [35].

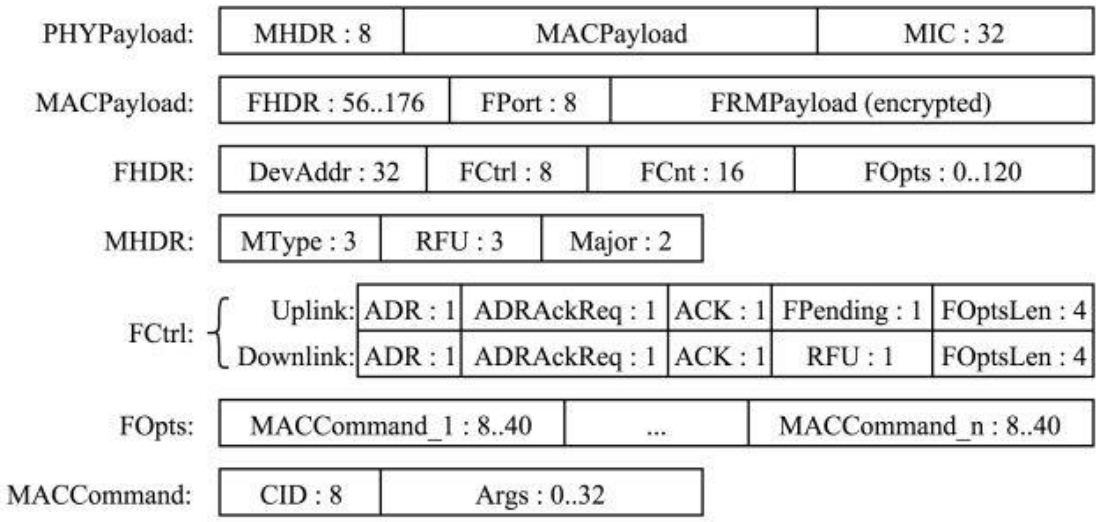


Figura 3.13: Formato de paquete LoRaWAN.

De forma estructurada y con el formato de paquete físico LoRa presente, en la Figura 3.14 se muestra el orden de encapsulado del mensaje [36].

A continuación, se procede a indicar el significado de cada campo de las tramas que forman el paquete.

- *Frame Header*:
    - *DevAddr*: Es la dirección corta del dispositivo que envía el paquete. 32 bits.
    - *FCtr*l: Control de trama. 8 bits.
    - *FCnt*: Contador de trama. 16 bits.

- *FOpts*: Opciones de trama para transportar comandos MAC. 0 - 120 bits.

- *MAC Payload*:

  - *FHDR*: Cabecera de trama. 56 - 176 bits.

  - *FPort*: Puerto opcional por multiplexado. 8 bits.

  - *FRMPayload*: Payload opcional encriptado usando *Advanced Encryption Standard* (AES) con una clave de 128 bits.

- *PHY Payload*:

  - *MHDR*: Cabecera MAC. 8 bits.

  - *MACPayload*: Datos de la capa superior.

  - *MIC*: Código criptográfico de integridad del mensaje, calculado sobre los campos *MHDR*, *FHDR*, *FPort* y *FRMPayload* encriptado. 32 bits.

- *Radio PHY Layer*:

  - *MType*: Indica el tipo de mensaje, si es ascendente o descendente, y si es o no un mensaje de confirmación. 3 bits.

  - *RFU*: Reservado para uso futuro. 3 bits. 1 bit en *Downlink FCtrl*.

  - *Major*: Indica la versión de LoRaWAN. 2 bits.

  - *ADR*: *Adaptative Data Range*. 1 bit.

  - *ADRAckReq*: Control sobre el mecanismo ADR.

  - *ACK*: Confirmación. 1 bit.

  - *FPending*: Indica que el servidor de red aún tiene datos pendientes para enviar e insta al nodo final a transmitir lo antes posible para de esta forma abrir la ventana de recepción. 1 bit.

- *FOptsLength*: Longitud de FOpts en bytes. 4 bits.
- *MACCommand*: Comando MAC. 8 – 40 bits.
- *CID*: Identificador del comando. 8 bits.
- *Args*: Argumentos opcionales del comando. 0 – 32 bits.

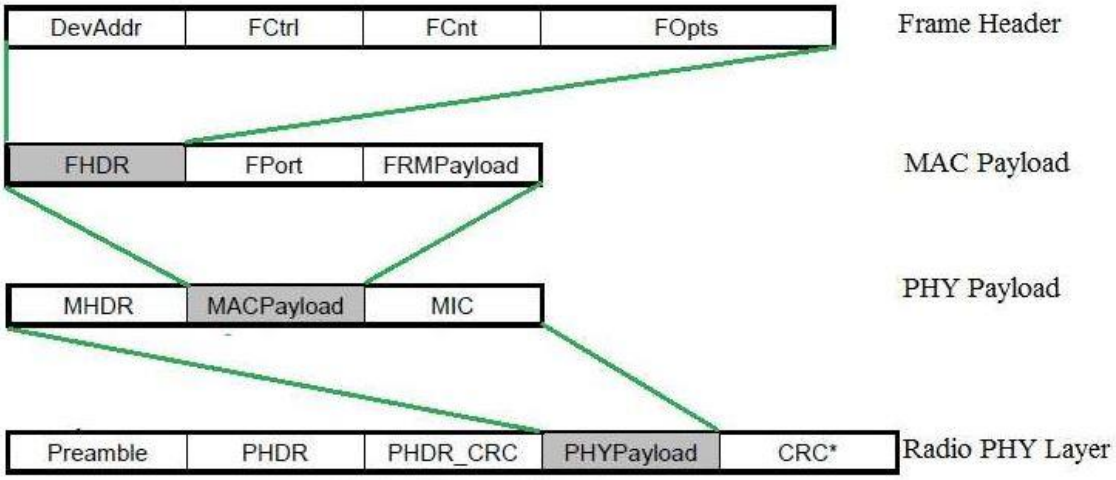


Figura 3.14: Encapsulado de las tramas del paquete LoRaWAN.

En esta descripción de los diferentes campos de los paquetes, se ha nombrado el concepto de comando MAC. LoRaWAN define multitud de comandos MAC que permiten configurar los parámetros de los dispositivos finales. Los comandos MAC pueden ser enviados por el nodo final o por el servidor [35]:

- *LinkCheckReq*: Es el único comando MAC que puede ser enviado por un nodo final, y se utiliza para poner a prueba su conectividad.
- *LinkADRReq*: Controla la tasa binaria y la potencia de salida usada por el dispositivo, así como el número de veces que se debe enviar un paquete no confirmado.
- *DutyCycleReq*: Determina el ciclo de trabajo global del dispositivo.

- *RXTimingSetupReq/RXParamSetupReq*: Permite cambiar parámetros de la ventana de recepción.

- *NewChannelReq*: Permite cambiar los canales usados por el dispositivo.

- *DevStatusReq*: Sirve para consultar el nivel de batería y la calidad de recepción de un dispositivo.

Se debe comentar que, debido a la topología de la red, en estrella, la comunicación siempre va a ser nodo final – *gateway*, y viceversa, lo cual conlleva que no se requiera de un campo de dirección del destino en mensajes ascendentes ni dirección de fuente en mensajes descendentes.

3.1.6.4 Acceso a LoRaWAN

Para añadir un dispositivo a una red LoRaWAN y poder considerarlo un nodo de ésta, se requiere de una activación en dicha red. Este proceso puede realizarse mediante *Over-The-Air-Activation* (OTAA), o *Activation By Personalization* (ABP). Sea cual sea el proceso elegido para realizar la activación, este ha de proporcionar al nodo final de la información que se indica a continuación. Esta información es imprescindible para garantizar una comunicación correcta y segura dentro de la red.

- *Application Identifier* (*AppEUI*): Identificador global único de aplicación en el espacio de direcciones de *Institute of Electrical and Electronics Engineers* (IEEE) EUI64, que identifica al propietario del nodo final. Longitud de 64 bits.

- *Device Identifier* (*DevEUI*): Identificador global único del dispositivo en el espacio de direcciones IEEE EUI64, que identifica al nodo final. Longitud de 64 bits.

- *Application Key* (*AppKey*): Clave de aplicación de la que derivan las claves de sesión. Longitud de 128 bits. No se usa en el proceso ABP.

- *Device Address* (*DevAddr*): Este es un campo que se incluye siempre en todos los paquetes de comunicación, y consiste en un identificador de 32 bits, de los cuales 7 bits se destinan como identificador de la red, y los 25 restantes para la dirección del dispositivo para esa red. Longitud de 32 bits.

- *Network Session Key* (*NwkSKey*): Clave utilizada por el servidor de la red y el nodo final para calcular y verificar el código de integridad de mensajes para asegurar la integridad de los datos. Longitud de 128 bits.

- *Application Session Key* (*AppSKey*): Clave utilizada por el servidor de la red y el nodo final para encriptar o desencriptar el campo de área de datos de los mensajes. Clave del tipo AES-128. Longitud de 128 bits.

La diferencia fundamental entre el proceso OTAA y ABP reside en la forma con la que se obtienen en cada proceso las claves para una nueva conexión o sesión.

En OTAA, lo primero que se debe hacer es introducir manualmente el identificador global único (*DevEUI*), que como se ha comentado anteriormente identifica de forma única al dispositivo utilizando el esquema IEEE EUI64 [37], el identificador de aplicación (*AppEUI*) y la clave de aplicación específica para el dispositivo (*AppKey*), a partir de la cual se derivarán las claves de sesión y la de sesión de aplicación. El proceso de OTAA realmente comienza cuando se envía por parte del dispositivo final un mensaje MAC denominado *join-request*. Esta petición de unión contiene los parámetros que se introdujeron manualmente con anterioridad, *AppEUI* y *DevEUI*, además de un parámetro adicional denominado *DevNone*. El parámetro *DevNone* es un valor aleatorio utilizado para evitar ataques de reenvío de *join-request* [37]. Si el dispositivo final tiene permiso para unirse a la red, el servidor envía otro mensaje MAC en el que se acepta la petición realizada (*join-accept)*. Esta aceptación está compuesta por un valor aleatorio (*AppNone*), la dirección del dispositivo (*DevAddr*) y el identificador de la red (*NetID*). A partir del parámetro *AppNone* y el valor ya conocido de *AppKey*, se derivan las dos claves de sesión, *AppSKey* y *NwkSKey*. Con estos parámetros almacenados en el dispositivo final, más los introducidos manualmente, ya se pueden realizar transmisiones.

La principal ventaja de usar OTAA como proceso de configuración es que la red genera y envía las claves de encriptación, lo que hace que el sistema resulte más seguro [37]. La desventaja que se puede citar en OTAA, es que se necesita la implementación de un mecanismo de conexión con una complejidad añadida.

En el proceso ABP toda la configuración se realiza de manera manual, se debe introducir en el dispositivo final la dirección del dispositivo y las claves de sesión. Una vez hecho esto, ya se puede transmitir.

La principal ventaja de usar ABP es clara, la facilidad y la rapidez con la que se puede conectar a la red LoRaWAN. El dispositivo puede estar disponible para transmitir en muy poco tiempo. La desventaja se muestra en que las claves de encriptación ya están preconfiguradas en el dispositivo, y esto resta seguridad.

3.1.6.5 Seguridad

Con todos los campos de las tramas de los paquetes LoRaWAN, y con la definición de todas las claves que intervienen en la configuración de los nodos de la red LoRaWAN, ya se puede hablar en términos de seguridad.

La seguridad en IoT es un aspecto fundamental, pues los datos que se intercambian muestran explícita o implícitamente información personal o privada. Y aunque no fuera así en muchos casos se desea guardar la confidencialidad de la información transmitida. LoRaWAN tiene esto muy en cuenta, y por eso posee mecanismos de autenticación mutua, de integridad y confidencialidad. Todo centrado en un esquema de claves simétricas.

Ya en el proceso de activación se requiere de una autenticación mutua entre un dispositivo inicial y el servidor de red. Es decir, solo los dispositivos autorizados pueden activarse en una determinada red. El dispositivo final utiliza la *AppKey* para calcular el campo *MIC* que se recuerda que es el código de integridad del mensaje [37]. En el otro extremo, el servidor tiene almacenada la *AppKey* del dispositivo final, y por tanto comprueba que el mensaje que recibe por parte del dispositivo final es auténtico. Además, para que el dispositivo final se asegure que la respuesta proviene del servidor

adecuado, este responde con el *join-accept* cuyo *MIC* ha sido calculado con la *AppKey* que comparten el dispositivo final y el propio servidor. Además, el mismo mensaje de aceptación se cifra con la *AppKey*.

Toda la seguridad en el tráfico normal de paquetes se realiza a través de las dos claves de sesión, la de sesión de red (*NwkSKey*) y la de aplicación (*AppSKey*) [37], que se encargan de cifrar y firmar los datos. Se recuerda que ambas claves derivan de *AppKey*, y por tanto solo son conocidas por el dispositivo final y el servidor de red.

Así, se usa la misma clave para cifrar y descifrar, y por eso se necesita que tanto el nodo final como el servidor conozcan las claves, que deben ser las mismas. Esto implica que se debe enviar en algún momento esas claves compartidas, poniendo en riesgo la seguridad de la red. LoRaWAN soluciona esto compartiendo el *AppKey*, que no hay problema en que sea compartida, y con la clave aleatoria que se genera en el proceso de activación [37]. A partir de estos dos componentes se derivan las clases de sesión que serán comunes para el dispositivo final y para el servidor, sin tener que ser compartidas entre ellos. Por tanto, esta información solo está disponible en estos dos puntos de la red.

La intercepción de la *AppKey* no tiene consecuencias en la seguridad, porque solo serviría para activar fraudulentamente un dispositivo en una red determinada. Se recuerda que el algoritmo de cifrado es AES-128 bits, que mantiene una buena relación entre el coste computacional y la robustez.

A modo de resumen, en este apartado se adjunta en la Figura 3.15 [25] el esquema que sigue una red LoRa con los tres elementos clave que intervienen en ella, el sensor, el *gateway* y el servidor de la red. También se muestran los diferentes niveles de comunicación que tiene cada elemento, y se remarcan aquellos en los que se establece una conexión segura con cifrado.

Tras realizar un análisis completo de la tecnología LoRa se puede concluir que es una tecnología que se adapta perfectamente a las necesidades de IoT, pues su consumo de potencia es muy bajo y su alcance elevado. Otra ventaja que presenta LoRa es la capacidad para ajustar la comunicación a las necesidades reales del nodo que quiere transmitir. Todos estos ajustes los establece LoRaWAN usando el ya comentado ADR. Se debe recordar que el alcance y la tasa binaria dependen del SF que se establezca.

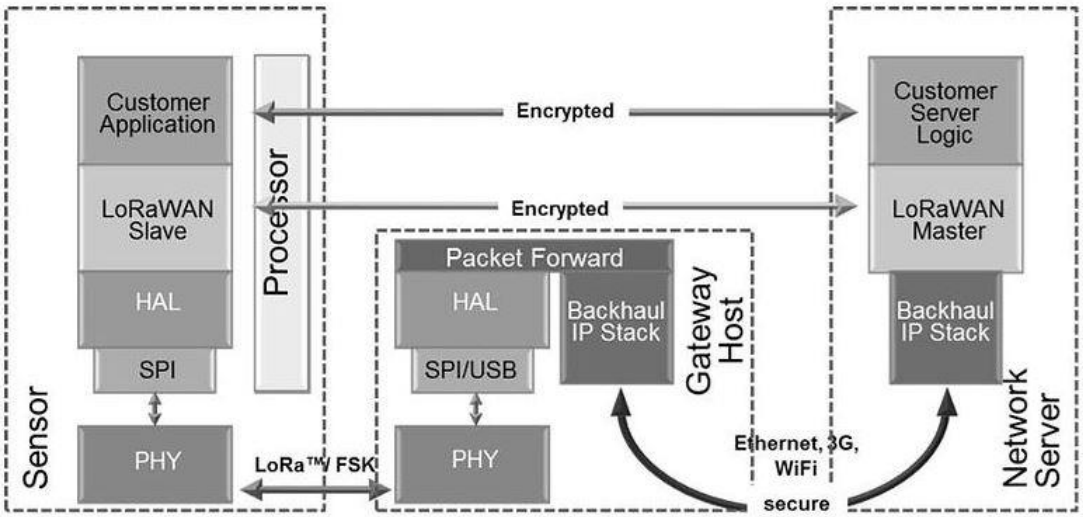


Figura 3.15: Relaciones entre los elementos típicos de una red IoT basada en LoRa.

3.2 Descripción de la plataforma HW/SW propuesta

Tras haber estudiado la tecnología de comunicación que se utilizará en la plataforma HW/SW propuesta se describe a continuación su arquitectura. La arquitectura de la plataforma HW/SW de *Smart Parking* propuesta en el presente TFG se muestra en la Figura 3.16 y consta de cinco bloques funcionales básicos:

- *Nodo final*: Elemento de la arquitectura que se encarga de la detección del estado de la plaza de estacionamiento, y que envía la información recogida por los sensores, al *gateway*.

- *Gateway:* Elemento que permite la comunicación entre el nodo final y el servidor.

- *Plataforma The Things Network (TTN)*: Elemento de la arquitectura que recoge y almacena toda la información procedente del *gateway*, y, por tanto, del nodo final.

- *Integración HTTP*: Elemento final que permite la visualización de la información recibida en TTN sin necesidad de acceder a la cuenta privada de dicha plataforma.

- *Aplicación Java*: Elemento final que permite la visualización de la información recibida en la plataforma TTN mediante interfaces visuales adaptadas al usuario.

Además, en la Figura 3.16 se indica el tipo de comunicación que utilizan los diferentes elementos de la plataforma HW/SW para la transmisión/recepción de información. Así, se usan dos tecnologías de comunicación, LoRa/LoRaWAN y TCP/IP. La tecnología LoRa/LoRaWAN permite establecer la comunicación entre el nodo final y el *gateway*, mientras que la conexión TCP/IP permite la comunicación entre el *gateway* y TTN, así como el intercambio de datos entre esta plataforma y la integración o la aplicación *Java*.

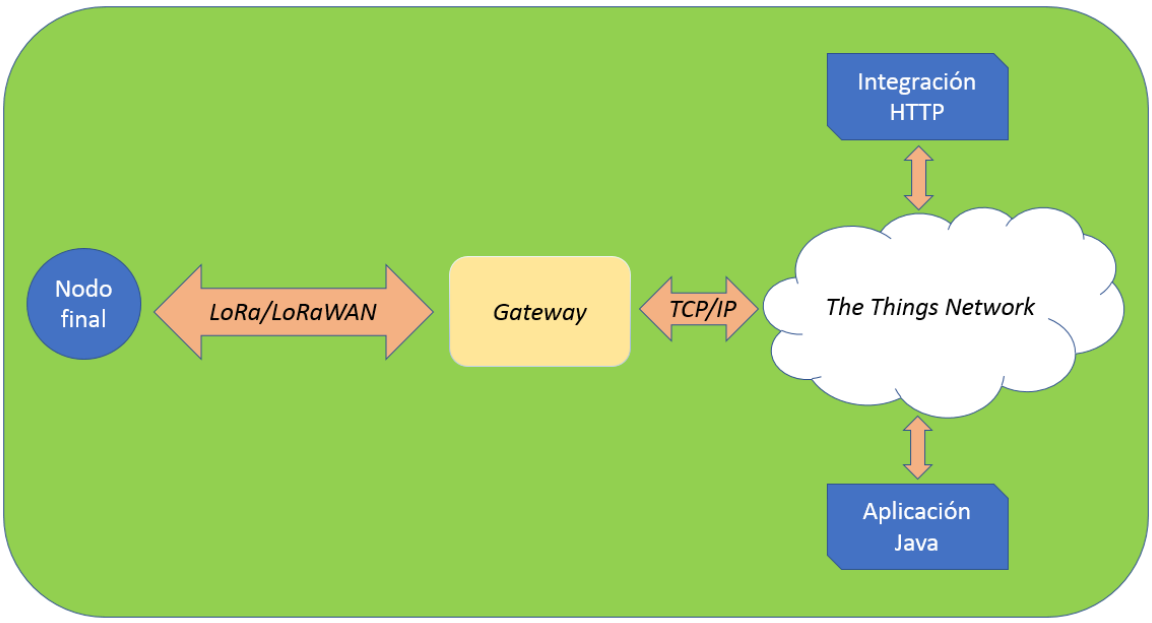


Figura 3.16: Arquitectura de la solución propuesta.

Capítulo 4. Sensores del nodo final

Dentro de la arquitectura propuesta para la solución desarrollada en este TFG se tiene como primer nivel, el correspondiente a la sensorización. Para llevar a cabo la tarea de sensorización se cuenta con dos tipos de sensores que cumplen con los requerimientos de la solución propuesta. Para el caso concreto del presente TFG se necesita hacer uso de sensores que sean capaces de detectar que hay un objeto, asociado a un vehículo, ocupando permanentemente un espacio determinado. Con el fin de conocer el estado de una plaza de estacionamiento.

Para detectar el estado de estas plazas de aparcamiento se han estudiado dos tipos fundamentales de sensores:

- *Sensores de proximidad*: Sensor que usa tecnología de rayos infrarrojos para la detección de un objeto en dos dimensiones. En concreto, se ha evaluado el uso del dispositivo *ZX Distance and Gesture Sensor* de la empresa *Sparkfun*.

- *Sensores magnetómetros*: Sensor que mide el campo magnético en las 3 dimensiones. En concreto, se ha evaluado el uso del dispositivo QMC5883L de la empresa *QST Corporation*.

Ambos dispositivos se conectan al nodo final mediante comunicación *serie Inter-Integrated Circuit* (I^2C), por lo que enviarán los valores correspondientes a las medidas realizadas al nodo final a través de las líneas *Serial Data* (SDA) y *Serial Clock* (SCL) propias de la interfaz I^2C.

Inicialmente se había planteado desarrollar la solución estudiando únicamente el sensor magnetómetro, pero tras realizar varias pruebas iniciales de campo se concluyó que estudiar las dos opciones para el nivel de sensorización enriquecería más la solución propuesta en este TFG. Por esta razón, se comentan en detalle cada uno de los sensores evaluados.

También se debe tener en consideración antes de comenzar a caracterizar los sensores, que el *firmware* de usuario del dispositivo LoPy se codifica en lenguaje *MicroPython*, mientras que el del dispositivo Seeeduino LoRaWAN se codifica en el lenguaje *Wiring* de Arduino. Esto resulta ser un aspecto relevante, dado que en la actualidad existen librerías asociadas a ambos sensores para el caso del dispositivo Seeeduino LoRaWAN, pero no para el caso del dispositivo LoPy, por lo que al no disponer de librerías en *MicroPython*, sería necesario desarrollarlas en caso de seleccionar el dispositivo de Pycom para implementar la funcionalidad del nodo final.

4.1 Sensor de proximidad

El sensor de proximidad utilizado inicialmente para la plataforma HW/SW desarrollada en el presente TFG ha sido el dispositivo sensor *ZX Distance and Gesture Sensor* [38], de la empresa *Sparkfun*, que se muestra en la Figura 4.1.

Figura 4.1: Sensor ZX Distance and Gesture Sensor.

La tecnología utilizada por el sensor de proximidad de *Sparkfun* se basa en el uso de *Infrared Radiation* o rayos infrarrojos (IR). Este sensor es capaz de detectar gestos simples sobre su superficie, como por ejemplo deslizamientos de izquierda a derecha, y viceversa, además de medir la distancia de un objeto alejado del sensor hasta aproximadamente 30 cm, para el denominado eje "Z", y también la ubicación del objeto de lado a lado a través del sensor en una distancia de 15 cm, para el denominado eje "X".

Como se puede observar en la Figura 4.2, el sensor de proximidad de *Sparkfun* cuenta con 11 pines. En la Tabla 4.1 se describe la función asignada para cada uno de ellos [39].

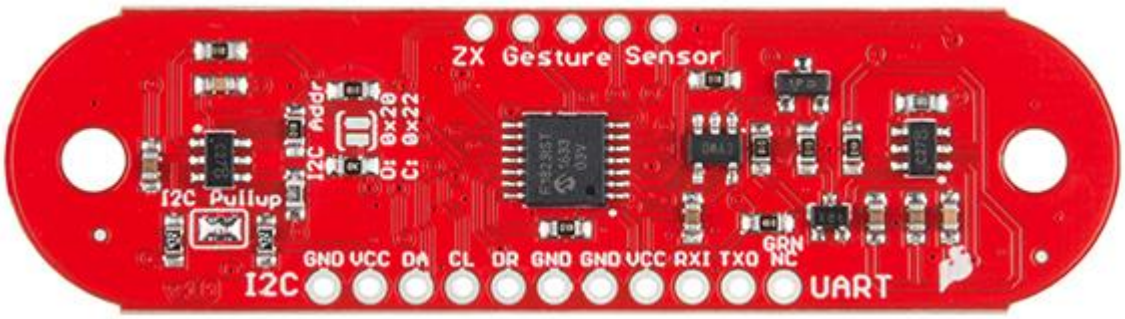


Figura 4.2: Pines del sensor ZX Distance and Gesture Sensor.

Pin	Descripción
GRN	No usado.
TXO	Transmisión UART.

RXI	Recepción UART.
VCC	Alimentación 3.3-5 V.
GND	Conexión a tierra.
BLK	No usado. Se conecta a GND.
DR	*Data Ready*. Se activa a nivel alto cuando hay un dato para ser leído vía I^2C.
CL	Línea de reloj de I^2C.
DA	Línea de datos de I^2C.

Tabla 4.1: Pines del sensor ZX Distance and Gesture Sensor.

En la Figura 4.2 también se observa que existen dos *Jumpers* que pueden ser configuradas por el usuario. El primero de ellos es el denominado I^2C *Pullups*, que se encuentra en la esquina inferior izquierda de la Figura 4.2. Por defecto, el sensor *ZX Distance and Gesture Sensor* viene con resistencias *Pull-Up* de 4.7kΩ en las líneas de I^2C, SDA y SCL. Para desconectar el *Pull-Up*, basta con eliminar la soldadura de este *Jumper*.

Por otra parte, se tiene otro *Jumper* que por defecto viene abierto. Este *Jumper* se denomina *I^2C Addr*, y permite modificar la dirección I^2C del sensor. Si el *Jumper* permanece abierto, la dirección del dispositivo I^2C es 0x10, mientras que si se cierra el *Jumper*, la dirección I^2C pasa a ser 0x11.

Este sensor ha sido diseñado para satisfacer las siguientes características [40]:

- Comunicación *Universal Asynchronous Receiver/Transmitter* (UART) a 115200 baudios, 8 bits y sin paridad.
- Interfaz I^2C.
- Salida de datos en I^2C y UART simultáneamente.
- Salida de datos continua a una tasa de 50 muestras/s.

Con el fin de poder desarrollar la librería que permita establecer la comunicación entre el sensor y los dispositivos LoPy que actuarán como nodos finales, se debe analizar el mapa de registros del sensor *ZX Distance and Gesture Sensor*, mostrado en la Tabla 4.2.

Dirección	Nombre	Descripción
0x00	STATUS	Estado del sensor.
0x01	DRE	Activación de *Data Ready*.
0x02	DRCFG	Configuración de *Data Ready*.
0x04	GESTURE	Último gesto detectado.
0x05	GSPEED	Velocidad del último gesto detectado.
0x06	DCM	Data Confidence Metric.
0x08	XPOS	Coordenada X.
0x0A	ZPOS	Coordenada Z.
0x0C	LRNG	Rango de datos del emisor izquierdo.
0x0E	RRNG	Rango de datos del emisor derecho.
0xFE	REGVER	Versión del mapa de registros.
0xFF	MODEL	ID del modelo del sensor.

Tabla 4.2: Mapa de registros de del sensor ZX Distance and Gesture Sensor.

En la documentación del sensor se explica la forma en la que este obtiene los valores para las coordenadas X y Z.

Así, los valores de la coordenada X se determinan midiendo la iluminación diferencial de los dos elementos ópticos (el par emisor-receptor). Cuando el objeto reflector está encima del sensor, ambos elementos ópticos son iluminados con la misma intensidad, produciendo un diferencial de brillo nulo. Por definición, esta posición se refiere al origen X, ó X = 0. Cuando el reflector no está centrado, el receptor obtiene más luz reflejada de uno de los emisores y menos del otro, por lo tanto, se puede determinar la ubicación del reflector en función del grado en que la recepción de luz no está equilibrada.

El rango Z es una medida de la iluminación directa de ambos receptores. La medida en la coordenada Z es posible cuando el reflector se coloca directamente encima del sensor.

Los valores de las coordenadas de salida X y Z son calculados directamente por el sensor y pasados a los registros correspondientes.

4.2 Sensor magnetómetro

El sensor magnetómetro utilizado en el desarrollo del presente TFG es el sensor QMC5883L, de la compañía *QST Corporation*, que se muestra en la Figura 4.3.

Figura 4.3: Sensor QMC5883L de QST Corporation.

En la Figura 4.3 se muestran las flechas que indican el sentido positivo del valor de lectura del campo magnético es positivo para las 3 coordenadas. Este modelo concreto proporciona una interfaz I^2C y cuenta con 5 pines que se describen en la Tabla 4.3.

Pin	Descripción
VDD	Alimentación 2.16-3.6 V.
GND	Conexión a tierra.
DRDY	Pin de interrupción *Data Ready.* Se activa a nivel alto cuando hay un dato para ser leído vía I^2C. Conexión opcional.
SCL	Línea de reloj de I^2C.

SDA	Línea de datos de I^2C.

Tabla 4.3: Pines del sensor QMC5883L.

El sensor QMC5883L ha sido desarrollado para satisfacer las siguientes características [41]:

- Lecturas del campo magnético en las 3 coordenadas mediante sensores magneto-resistivos.
- Rango de lecturas del campo magnético de ±8 Gauss.
- Resolución de 2 mGa.
- Interfaz I^2C.
- Bajo consumo de potencia (100µA).

Un aspecto importante es la dirección I^2C del sensor magnetómetro, que es 0x0D. Esta dirección se tendrá que utilizar en el desarrollo de la librería del magnetómetro para el dispositivo LoPy de Pycom, así como las direcciones de los registros que se muestran a continuación en la Tabla 4.4, que recoge el mapa de registros facilitados por el fabricante del sensor.

Dirección	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Acceso**
0x00	Data Output X LSB Register XOUT [7:0]								Read
0x01	Data Output X LSB Register XOUT [15:8]								Read
0x02	Data Output Y LSB Register YOUT [7:0]								Read
0x03	Data Output Y LSB Register YOUT [15:8]								Read
0x04	Data Output Z LSB Register ZOUT [7:0]								Read
0x05	Data Output Z LSB Register ZOUT [15:8]								Read

0x06						DOR	OVL	DRDY	Read
0x07	TOUT [7:0]								Read
0x08	TOUT [15:8]								Read
0x09	OSR [1:0]		RNG [1:0]		ODR [1:0]		MODE [1:0]		Read/Write
0x0A	SOFT_RST	ROL_PNT						INT_ENB	Read/Write
0x0B	SET/RESET Period FBR [7:0]								Read/Write
0x0C	Reserved								Read
0x0D	Chip ID								Read

Tabla 4.4: Mapa de registros del sensor QMC5883L.

Como se puede ver en la Tabla 4.4, los primeros registros se destinan a los datos de salida. Estos datos son los resultados de las medidas realizadas por el sensor, cuyos valores se encuentran entre -32768 y 32767. Para cada coordenada se tienen 16 bits, motivo por el que cada coordenada cuenta con dos registros.

En la dirección 0x06 se tiene el registro *Status Register*, que cuenta con tres *flags* de estado que indican la situación del sensor. Estos *flags* son *Data Ready* (DRDY), *Overflow* (OVL) y *Data Skip* (DOR). Cuando estos *flags* están activados, indican que se tiene un dato listo, que hay una de las tres medidas realizadas fuera de rango, o que todos los canales de los registros de salida de datos están omitidos en el modo de medida continua, respectivamente. En caso de que estos bits permanezcan desactivados, se entiende que no se ha producido ninguno de los eventos anteriores.

En este magnetómetro también se cuenta con medidas de temperatura que se pueden utilizar para la aplicación de factores de corrección en las medidas del campo magnético. Para almacenar el valor de la temperatura se hace uso de los registros TOUT en las direcciones 0x07 y 0x08.

Por otra parte, en las direcciones 0x09 y 0x0A se tienen los registros *Control Register 1* y *Control Register 2*, respectivamente.

En el registro *Control Register 1* es posible seleccionar la configuración del sensor en términos de modo de funcionamiento (Mode), *Output Data Rate* (ODR), fondo de escala (RNG), y ratio de muestreo (OSR). Para cada campo se cuenta con 2 bits, de manera que se pueden establecer diferentes configuraciones, conseguir una configuración como las que se adjuntan en la Tabla 4.5.

Registro	00	01	10	11
Mode	Standby	Continuous	Reserve	Reserve
ODR	10 Hz	50 Hz	100 Hz	200 Hz
RNG	2 Ga	8 Ga	Reserve	Reserve
OSR	512	256	128	64

Tabla 4.5: Tipos de configuraciones del sensor QMC5883L.

En lo que respecta al registro *Control Register 2*, se cuenta con 3 bits de control: SOFT_RST, ROL_PNT y INT_ENB. Este último permite habilitar o deshabilitar la interrupción que se genera cuando hay un nuevo dato disponible. ROL_PNT permite activar la función *Roll-Over*, mientras que SOFT_RST permite realizar una inicialización de todos los registros, dejándolos con los valores por defecto.

En la siguiente dirección del mapa de registros se tiene el registro *SET/RESET Period*, cuyo valor recomendado por el fabricante es 0x01, por lo que en la codificación de la librería desarrollada para el dispositivo LoPy, este registro se dejará con el valor recomendado.

Por último, en la dirección 0x0D se tiene el registro *Chip ID*, que almacena el valor 0xFF. Este registro se usa para la identificación del sensor, por lo que resulta útil en las funciones de inicialización de la librería desarrollada, que se comentará más adelante.

Este sensor posee dos modos de operación, desarrollados con el objetivo de obtener una mejor gestión del consumo de potencia. Los modos disponibles son: *Continuous Mode* y *Standby Mode*. Los cambios de modo se pueden realizar de acuerdo con el procedimiento que se muestra en la Figura 4.4.

En el modo *Continuous-Measurement Mode*, el dispositivo realiza continuamente medidas a una tasa y un rango seleccionados por el usuario, almacenando los resultados

en los registros *Data Output Registers*. Si fuera necesario, estos datos pueden leerse directamente de estos registros. Además, estos registros se mantienen actualizados constantemente según se van realizando nuevas medidas. Para reducir consumo de potencia, el dispositivo pasa a un estado similar al del modo *Idle*, pero sin modificar el modo en el registro *Mode Register*. Por otra parte, para modificar la tasa salida de datos y el rango de medida, se accede al registro *Control Register 1*.

El modo que viene configurado por defecto para el dispositivo es *Standby Mode*. En este modo, solo se mantienen funcionales algunos bloques con el fin de ahorrar consumo de potencia. En este modo no se realizan medidas continuas en el magnetómetro.

Una vez explicados los dos modos configurables en el dispositivo, se puede abordar el diagrama de funcionamiento del sensor, que se muestra en la Figura 4.4. Así, tras producirse un evento de encendido o reset (POR) o un reseteo por *software* (SOFT_RST), el magnetómetro pasa al modo por defecto, *Standby Mode*.

Si se produce un cambio en el campo MODE del registro *Control Register 1* (0x09), se pasa al modo *Continuous Mode*, en el que, de la misma forma, si se proporciona un cambio en el valor del campo *Mode* del registro *Control Register 1*, se pasa al modo *Standby Mode*. En caso de que el campo *Mode* no se modifique, tampoco lo hace el modo de funcionamiento del sensor.

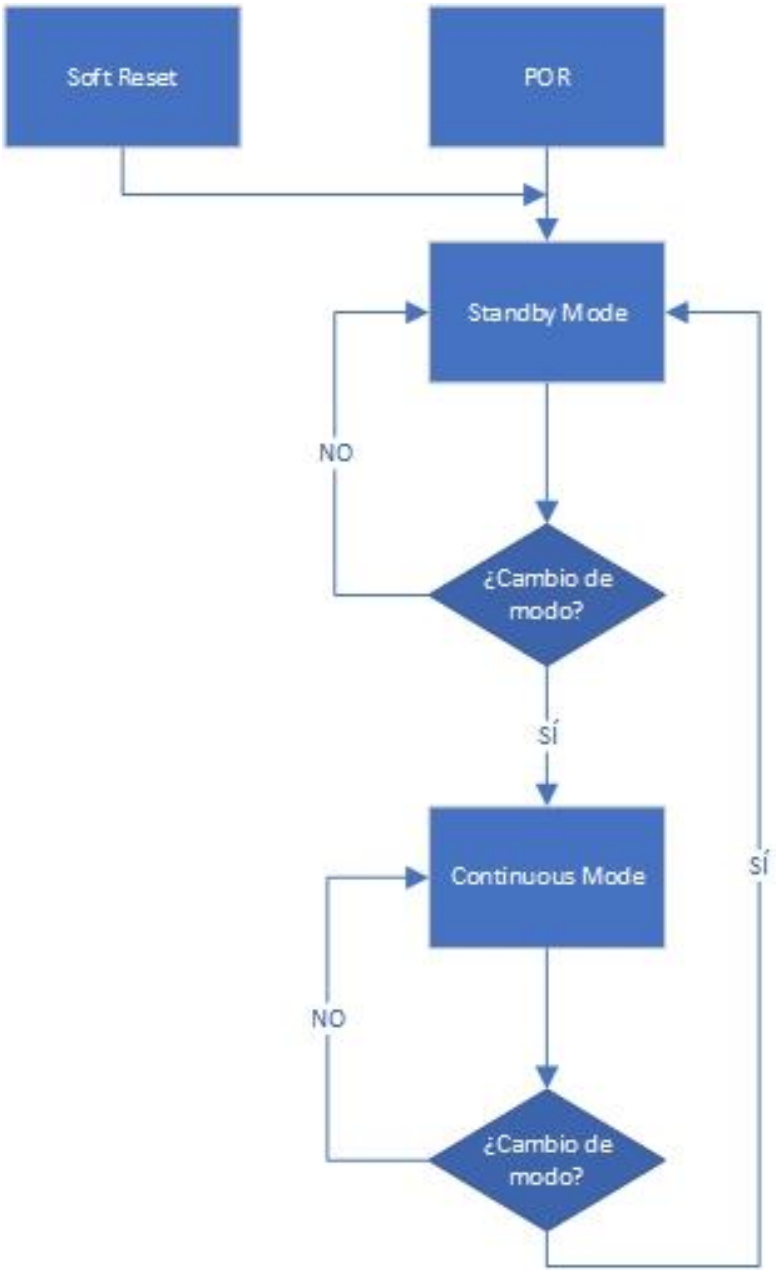


Figura 4.4: Diagrama de flujo del magnetómetro QMC5883L.

Capítulo 5. Nodo final

Dentro de las soluciones IoT se entiende por nodo final aquel dispositivo de la arquitectura IoT que se encarga de recoger las medidas realizadas por los sensores, acondicionarlas y transmitirlas hacia un elemento dispuesto en un nivel superior de la jerarquía, como puede ser en el caso concreto de este TFG, un dispositivo *gateway*. Para integrar estas tareas en un único elemento se utilizan dispositivos que cuenten con vías de comunicación en ambos sentidos:

- Descendente: Para la comunicación con los dispositivos sensores. Se suelen emplear comunicaciones serie como I^2C o *Serial Peripheral Interface* (SPI), o bien comunicaciones inalámbricas de área personal como *Bluetooth* o *ZigBee*.

- Ascendente: Para la comunicación con los elementos superiores en la jerarquía de la arquitectura IoT, habitualmente *gateways* o concentradores. Las tecnologías para cubrir esta comunicación son más variadas, y dependiendo de las necesidades de la aplicación, se puede optar por una tecnología u otra.

Para implementar la funcionalidad del nodo final se hace uso de un dispositivo IoT. Los dispositivos que se consideran inicialmente para desempeñar esta función son el dispositivo Seeeduino LoRaWAN, y el dispositivo LoPy. La decisión de seleccionar un

dispositivo u otro depende de la caracterización individual de cada dispositivo y la comparación entre ambos.

5.1 Seeeduino LoRaWAN

Seeeduino LoRaWAN es una placa de desarrollo Arduino que hace uso de la tecnología LoRa/LoRaWAN. El dispositivo, desarrollado por la empresa Seeedstudio, se muestra en la Figura 5.1.

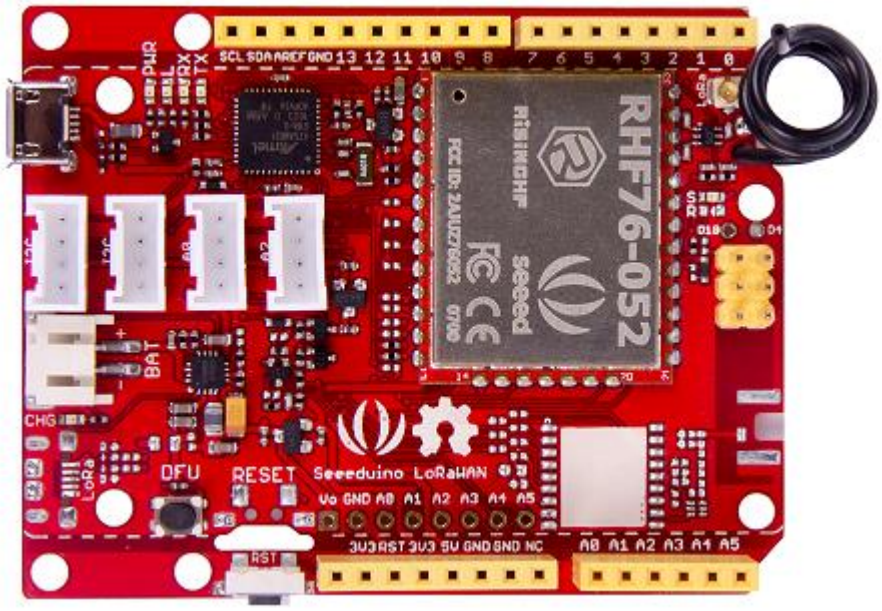


Figura 5.1: Seeeduino LoRaWAN de Seeedstudio.

Las principales características del dispositivo Seeeduino LoRaWAN son las siguientes: [42]

- Transceptor *RHF76-052* con LoRaWAN, compatible con dispositivos de Clase A y C.
- Conector *U.FL* de antena LoRa.
- Memoria *flash* de 256 KB.
- 2 x UART, 1 x I^2C, 1x SPI, 6 ADC (*Analogue to Digital Converter*) de 12 bits, 1 DAC (*Digital to Analogue Converter*) de 10 bits, y hasta 20 GPIO (*General-Purpose Input-Output*).

- Microcontrolador: ATSAMD21G18, 32-Bit ARM Cortex M0+.
- Dimensiones: 68x53 mm.
- 868/915 MHz (Europa) a +14 dBm máximo.
- 434/470 MHz (Norteamérica y Sudamérica, Australia y Nueva Zelanda) a +19 dBm máximo.

Para ubicar cada una de las funcionalidades que incorpora la placa de Seeedstudio, en la Figura 5.2 se representa un esquemático que enumera sus principales elementos.

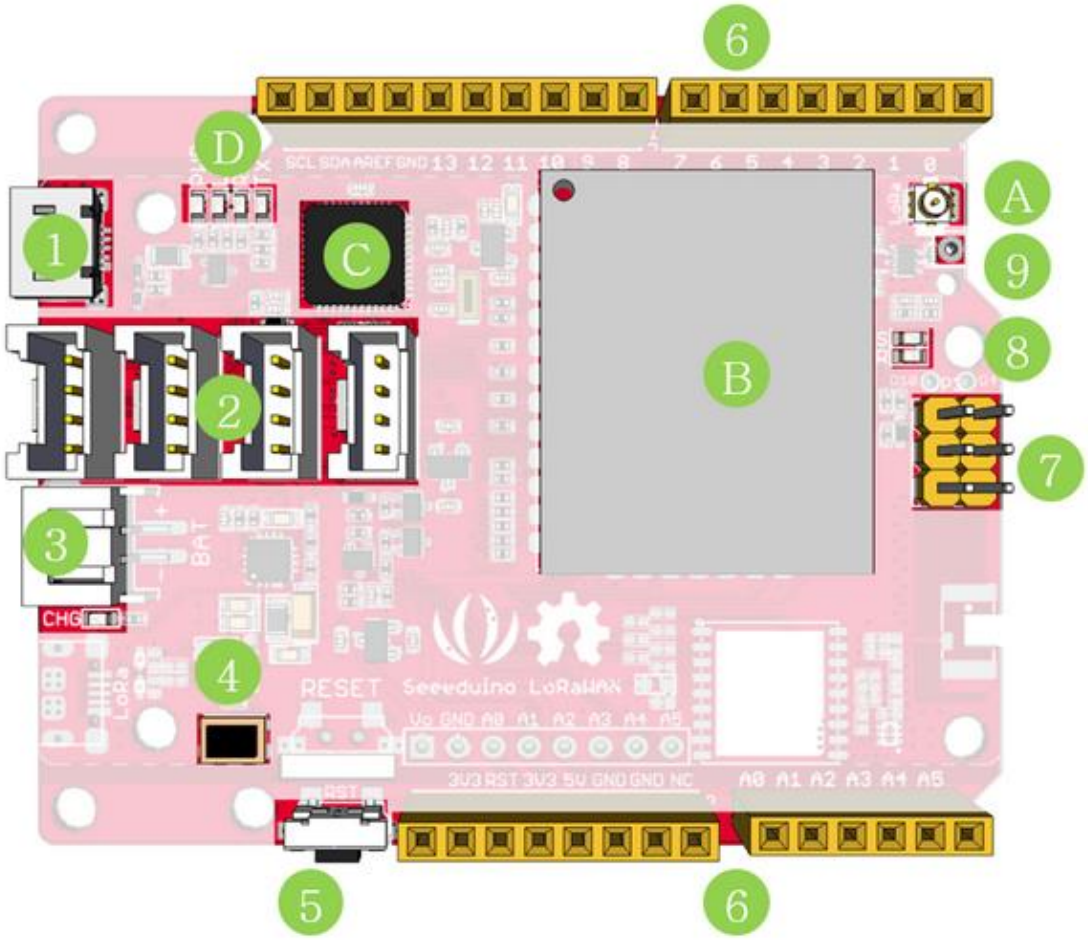


Figura 5.2: Hardware Seeeduino LoRaWAN.

Siguiendo la numeración de la Figura 5.2, los elementos HW integrados en la placa del dispositivo Seeeduino LoRaWAN son:

1: Micro USB.

2: Conectores *Grove*.

3: Conector JST2.0 *Lipo battery* (3.7V) y LED de estado de carga.

4: Botón DFU – botón modo *Firmware*.

5: Botón *Reset*.

6: Pines *Arduino*.

7: Pines ICSP (*In Circuit Serial Programming*).

8: LED modo *Firmware*.

9: Conector de antena Wire.

A: Conector de antena *U.FL*.

B: Módulo RF - RHF76-052AM.

C: Procesador ARM Cortex M0 - ATSAMD21G18.

D: LEDs

- ***RX/TX*** – Datos en UART (desde/hacia USB).
- ***L*** – Led conectado a D13.
- ***PWR*** – Power.

5.2 Dispositivo: LoPy

El dispositivo LoPy, mostrado en la Figura 5.3, ha sido desarrollado por la empresa Pycom [43]. LoPy [44] es una placa de desarrollo que integra tres tecnologías de comunicación en todas sus versiones: *Wireless Fidelity* (*WiFi*), *Bluetooth* y LoRa. En versiones más actuales se cuenta también con la tecnología LPWAN *Sigfox* [45]. La programación del dispositivo

se realiza en el lenguaje *MicroPython*. La placa ha sido diseñada específicamente para aplicaciones de IoT, como la que se presenta en este TFG.

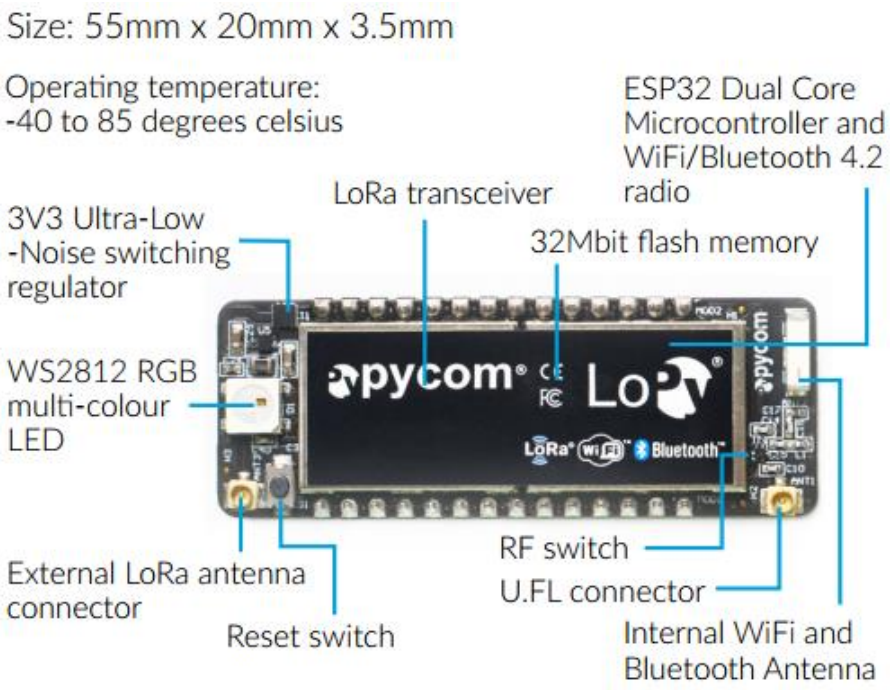


Figura 5.3: LoPy de Pycom.

En su núcleo se encuentra el *Sistema en Chip* (SoC) Espressif ESP32 que dispone de un microcontrolador de núcleo doble, *WiFi*, *Bluetooth*, LoRa y 512 KB de *Random Access Memory* (RAM). Como apunte importante se debe comentar que el procesador principal está completamente disponible para ejecutar la aplicación del usuario.

Las principales características técnicas del dispositivo LoPy1.0 son las siguientes [46]:

- Transceptor *Semtech* LoRa SX1272 con LoRaWAN, compatible con dispositivos de Clase A y C.
- Antena interna *WiFi* y Bluetooth y conector U.FL de antena LoRa.
- Memoria *flash* de 4 MB.
- 2 x UART, 2 x SPI, I^2C, *Inter-IC Sound* (I^2S), tarjeta MicroSD, 8 ADC de 12 bits, Timers: 4x16 bit con *Pulse-Width Modulation* (PWM) y hasta 24 GPIO.
- LED *Red-Green-Blue* (RGB) WS2812.

- *Real Time Clock* (RTC) - 32 KHz.
- IPv6.
- Interruptores *Radio Frequency* (RF) y reajuste.
- Temperatura de funcionamiento: -40 °C a 85 °C.
- Dimensiones: 55 x 20 x 3,5 mm.
- 868 MHz (Europa) a +14 dBm máximo.
- 915 MHz (Norteamérica y Sudamérica, Australia y Nueva Zelanda) a +20 dBm máximo.
- Alcance del nodo: hasta 40 km.
- *Nano-Gateway*: hasta 22 km.
- Capacidad *Nano-Gateway*: hasta 100 nodos.

Con respecto al consumo de potencia del dispositivo, se tienen las siguientes especificaciones [46]:

- Entrada: 3,3 V a 5,5 V.
- Salida: 3,3 V con hasta 400 mA.
- *WiFi*: 12 mA en modo activo, 5 µA en espera.
- LoRa: 15 mA en modo activo, 10 µA en espera.

Para hacer uso de todos los periféricos que incluye el dispositivo LoPy, se cuenta con 28 pines, distribuidos como se muestra en la Figura 5.4.

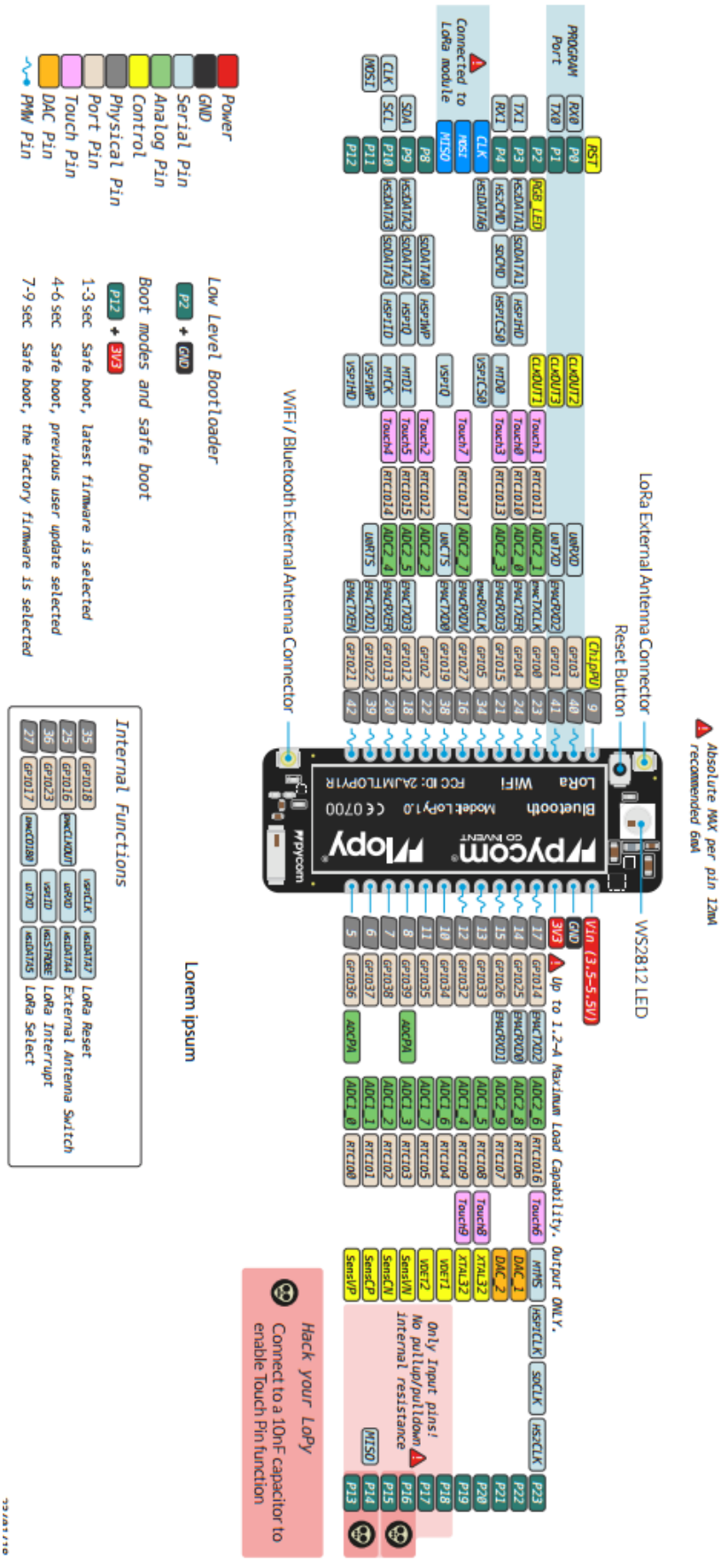


Figura 5.4: Pinout del dispositivo LoPy.

Además del propio dispositivo LoPy, se hace uso de una placa de extensión que ofrece la misma empresa, Pycom. Con la *Expansion Board* [47], que se muestra en la Figura 5.5, se puede programar el dispositivo LoPy a través de su puerto Micro Universal Serial Bus (MicroUSB), que sirve de alimentación y de comunicación serie. Además, cuenta con una conexión más cómoda para acceder a los pines del dispositivo LoPy, así como un conector de tarjeta *Micro Secure Digital* (MicroSD), y de batería Polímero de Litio (LiPo).

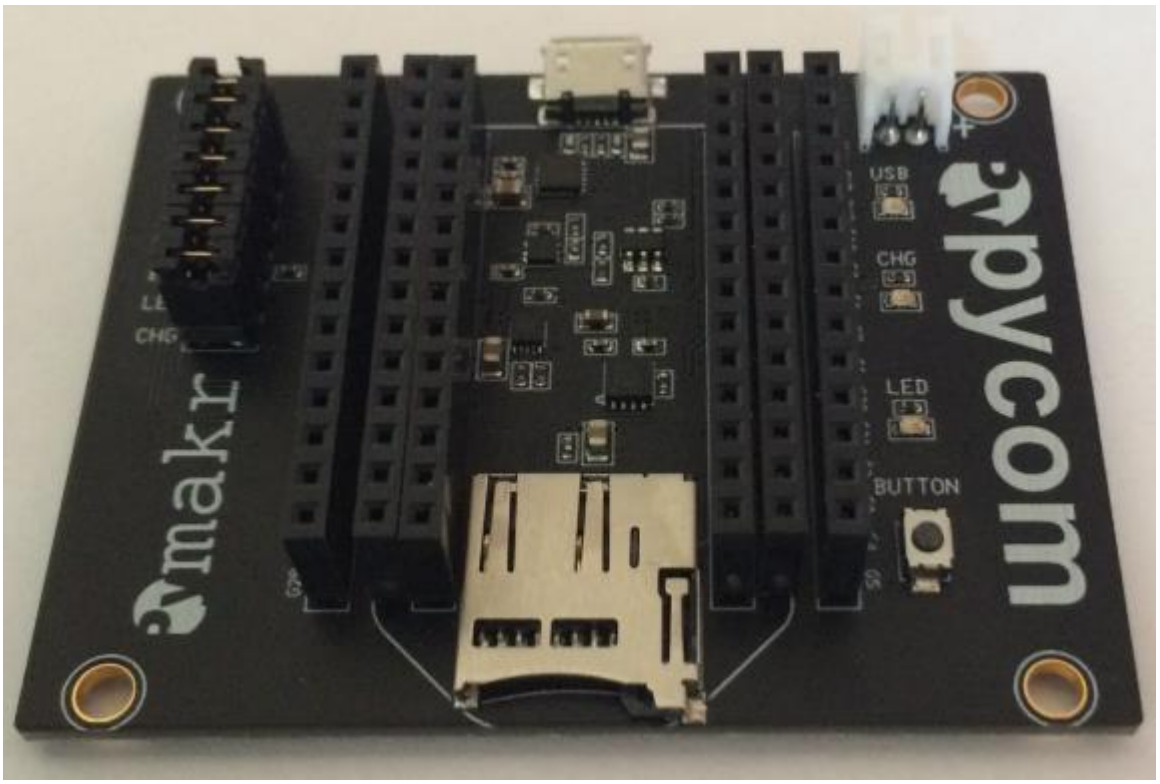


Figura 5.5: Expansion Board.

El diagrama de bloques de la *Expansion Board* de Pycom se muestra en la Figura 5.6. En ella se pueden observar las dos vías de alimentación de las que dispone la placa, y sus principales bloques funcionales.

Para ubicar cada una de las funcionalidades que incorpora la placa de expansión, en la Figura 5.7 se adjunta un esquemático que enumera los principales elementos que integra.

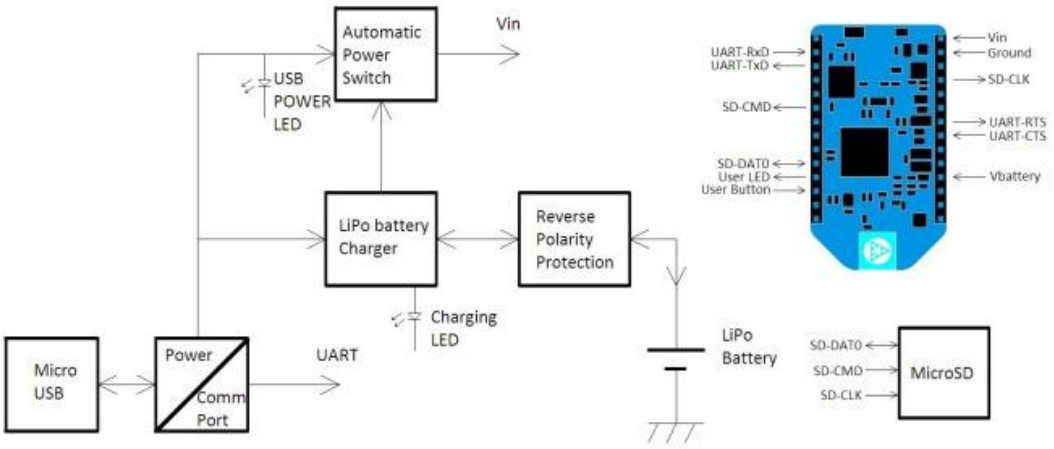
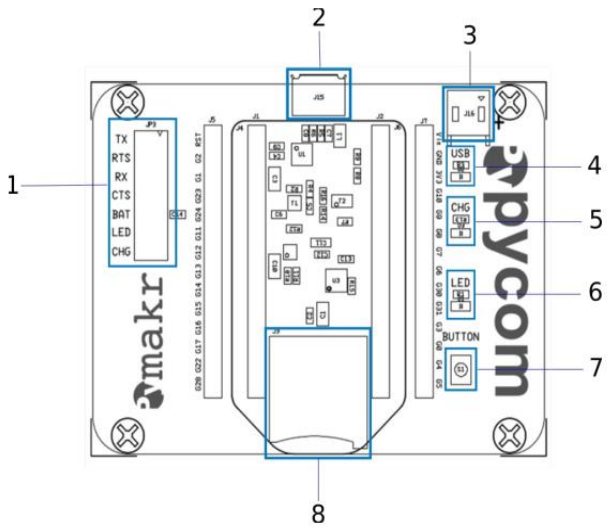


Figura 5.6: Diagrama de bloques.



1: Jumpers selectores de funcionalidad.

2: Conector Micro USB.

3: Conector JST de batería LiPo.

4: LED de carga USB.

5: LED indicador de carga de batería.

6: LED de usuario.

7: Botón de usuario.

8: *Socket* de tarjeta MicroSD.

Figura 5.7: Utilidades de la Expansion Board.

La *Expansion Board* también cuenta con unos pines cuya funcionalidad varía en función de si el *jumper* correspondiente está conectado (cerrado), o no (abierto). En la Tabla 5.1 se recoge la información asociada a estos pines.

Número de Jumper	Nombre del Pin	Jumper cerrado	Jumper abierto
1	GPIO2	RxD	GPIO2
2	GPIO6	CTS	GPIO6

3	GPIO1	TxD	GPIO1
4	GPIO7	RTS	GPIO7
5	GPIO3	VBat	GPIO3
6	GPIO16	Habilitado el LED de usuario	Deshabilitado el LED de usuario
7	-	Corriente de carga de batería: 450 mA	Corriente de carga de batería: 100 mA

Tabla 5.1: Funciones asociadas al estado de los Jumper en la Expansion Board.

5.3 Análisis comparativo

Inicialmente se realizó un análisis comparativo básico entre dos dispositivos que *a priori* son capaces de cumplir con las especificaciones de la solución propuesta en el presente TFG. Los dispositivos que se han comparado se corresponden con la plataforma Seeeduino LoRaWAN de Seeedstudio, y LoPy de Pycom.

El primer análisis se realizó en términos de fiabilidad de la comunicación. Para dicho análisis se llevaron a cabo cuatro pruebas experimentales de comunicación a partir de la combinación de dos dispositivos de cada fabricante. Los resultados de cada una de las pruebas realizadas se muestran en la Tabla 5.2, la Tabla 5.3, la Tabla 5.4 y la Tabla 5.5. En los resultados mostrados se han ido variando el valor de *Spreading Factor* (SF) y la distancia de la comunicación LoRa/LoRaWAN entre los dispositivos, con el fin de observar su respuesta. Las pruebas se han realizado en un mismo espacio cerrado con una serie de parámetros fijos y comunes para los cuatro casos. Estos parámetros comunes son la frecuencia, que se fija a la banda europea de 868 MHz, el ancho de banda, establecido a 125 KHz, los preámbulos de transmisión y recepción, que se fijan a un valor de 8 bits, y la potencia de transmisión que se configura al valor máximo permitido, que es de 14 dBm.

Antes de mostrar los resultados obtenidos, en la Figura 5.8, la Figura 5.9, la Figura 5.10 y la Figura 5.11 se adjunta el código utilizado los dispositivos para realizar transmisiones y recepciones de datos en cada caso, y por lo tanto, el código utilizado en esta prueba

inicial de fiabilidad de la comunicación. La programación del dispositivo LoPy se realiza en *MicroPython*, mientras que la del dispositivo Seeeduino LoRaWAN, en *Wiring*.

```
#include <LoRaWan.h>

unsigned char buffer[128] = {'hola', 0xff, 0x55, 3, 4, 5, 6, 7, 8, 9,};

void setup(void)
{
    SerialUSB.begin(115200);

    lora.init();

    lora.initP2PMode(868, SF12, BW125, 8, 8, 14);
}

void loop(void)
{
    lora.transferPacketP2PMode(buffer, 10);
    SerialUSB.println("Send hex.");
    delay(3000);
}
```

Figura 5.8: Transmisión LoRa en el dispositivo Seeeduino LoRaWAN.

Así, en la Figura 5.8 se muestra el código utilizado para realizar transmisiones LoRa con el dispositivo Seeeduino LoRaWAN. En primer lugar, se crea el paquete con el contenido que se desea enviar, seguidamente se inicializa la comunicación serie para visualizar los mensajes por pantalla, y la comunicación LoRa, con los parámetros previamente comentados.

Una vez que se establezca la comunicación LoRa, se realizan las transmisiones de paquetes con un período de 3 segundos.

Por otra parte, en la Figura 5.9 se muestra el código utilizado para realizar recepciones LoRa con el dispositivo Seeeduino LoRaWAN. Al igual que en el caso de la transmisión de paquetes, en primer lugar, se crea un paquete, de contenido nulo en este caso. Seguidamente, se inicializan las comunicaciones serie y LoRa. En el *loop* infinito se declaran variables con el propósito de medir la longitud de los paquetes y el nivel de intensidad de la señal recibida. Tras estas declaraciones, se recibe el paquete LoRa y se muestra por pantalla.

```
#include <LoRaWan.h>
unsigned char buffer[128] = {0, };
void setup(void)
{
    SerialUSB.begin(115200);
    lora.init();
    lora.initP2PMode(868, SF12, BW125, 8, 8, 14);
}
void loop(void){
    short length = 0;
    short rssi = 0;
    memset(buffer, 0, 128);
    length = lora.receivePacketP2PMode(buffer, 128,  &rssi, 1);
    if(length)
    {
        SerialUSB.print("Length is: ");
        SerialUSB.println(length);
        SerialUSB.print("RSSI is: ");
        SerialUSB.println(rssi);
        SerialUSB.print("Data is: ");
        for(unsigned char i = 0; i < length; i ++)
        {
            SerialUSB.print("0x");
            SerialUSB.print(buffer[i], HEX);
            SerialUSB.print(" ");
        }
        SerialUSB.println();
    }
}
```

Figura 5.9: Recepción LoRa en el dispositivo Seeeduino LoRaWAN.

```
from network import LoRa
import socket
import machine
import time
# initialize LoRa in LORA mode
# more params can also be given, like frequency, tx power and spreading factor
lora = LoRa(mode=LoRa.LORA)
# create a raw LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
while True:
# send some data
s.setblocking(True)
s.send('Hello')
# wait a random amount of time
time.sleep(machine.rng() & 0x0F)
```

Figura 5.10: Transmisión LoRa en el dispositivo LoPy.

Por otro lado, en la Figura 5.10 se muestra el código utilizado para realizar transmisiones LoRa mediante el dispositivo LoPy. En este código se inicia el dispositivo en modo LoRa y se crea un *socket*. En el bucle infinito se desbloquea el *socket* y se envía un mensaje de manera periódica.

En la Figura 5.11 se muestra el código utilizado para realizar recepciones LoRa con el dispositivo LoPy. Para recibir paquetes se inicializa el dispositivo igual que en el caso de la transmisión, se establece el modo LoRa, y se crea un *socket*, se deja bloqueado a mensajes salientes, de manera que se reciben los paquetes y se muestran por pantalla.

```
from network import LoRa
import socket
import machine
import time
# initialize LoRa in LORA mode
# more params can also be given, like frequency, tx power and spreading factor
lora = LoRa(mode=LoRa.LORA)
# create a raw LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
while True:
# get any data received...
s.setblocking(False)
data = s.recv(64)
print(data)
# wait a random amount of time
time.sleep(machine.rng() & 0x0F)
```

Figura 5.11: Recepción LoRa en el dispositivo LoPy.

Como se puede observar en el código de recepción del dispositivo LoPy, no hay ninguna variable destinada a medir el nivel de intensidad de la recepción, como sí había en el caso del dispositivo Seeeduino LoRaWAN. Para visualizar el nivel de señal recibido se puede añadir la línea:

```
Lora.stats()
```

El resultado de llamar a la función *stats()* es la visualización por pantalla del siguiente contenido, donde el nivel de intensidad de la señal recibida se corresponde el segundo elemento:

```
(rx_timestamp, rssi, snr, sftx, sfrx, tx_trials, tx_power,
tx_time_on_air, tx_counter, tx_frequency)
```

A continuación, se adjuntan los resultados cualitativos de los cuatro casos del análisis básico de la fiabilidad de ambos dispositivos.

- Emisor: Seeeduino LoRaWAN, Receptor: Seeeduino LoRaWAN:

SF	Distancia (m)	Fiabilidad
7	**1**	Muy buena
7	**7**	Muy buena
12	**1**	Muy buena
12	**7**	Muy buena

Tabla 5.2: Estudio de fiabilidad con Emisor: Seeeduino LoRaWAN, Receptor: Seeeduino LoRaWAN.

- Emisor: LoPy, Receptor: LoPy:

SF	Distancia (m)	Fiabilidad
7	**1**	Muy buena
7	**7**	Muy buena
12	**1**	Muy buena
12	**7**	Muy buena

Tabla 5.3: Estudio de fiabilidad con Emisor: LoPy, Receptor: LoPy.

- Emisor: LoPy, Receptor: Seeeduino LoRaWAN:

SF	Distancia (m)	Fiabilidad
7	**1**	Pobre
7	**7**	Buena
12	**1**	Regular
12	**7**	Buena

Tabla 5.4: Estudio de fiabilidad con Emisor: LoPy, Receptor: Seeeduino LoRaWAN.

- Emisor Seeeduino LoRaWAN, Receptor: LoPy:

SF	Distancia (m)	Fiabilidad
7	**1**	Pobre
7	**7**	Nula
12	**1**	Pobre
12	**7**	Nula

Tabla 5.5: Estudio de fiabilidad con Emisor: Seeeduino LoRaWAN, Receptor: LoPy.

En vista de los resultados obtenidos a partir de las pruebas experimentales realizadas se concluye que la comunicación entre dispositivos del mismo tipo se realiza con una fiabilidad muy alta, mientras que entre dispositivos de diferentes fabricantes la fiabilidad es escasa. De esta manera, se descarta a partir de estos resultados utilizar dispositivos de diferentes fabricantes. Como consecuencia de esto, el dispositivo seleccionado se usará como nodo final, y también como *gateway*.

Un apunte diferencial entre los dispositivos Seeeduino LoRaWAN y LoPy, es que en la documentación de la empresa Pycom se confirma la posibilidad de usar el dispositivo como *gateway*, de hecho, se incluye en dicha documentación un código de referencia desarrollado en *MicroPython* para implementar la funcionalidad de un *gateway* monocanal. Por otra parte, la empresa Seeedstudio no ofrece ningún código orientado a la implementación de un *gateway*, ni confirma la posibilidad de usar el dispositivo como tal.

A pesar de esto, Seeeduino LoRaWAN cuenta con las librerías de los sensores utilizados en la plataforma HW/SW propuesta, mientras que LoPy no. De esta manera, en caso de que se utilice el dispositivo LoPy para la solución propuesta se tendrán que desarrollar las librerías de cada sensor en lenguaje *MicroPython*.

Otro estudio realizado en este análisis comparativo entre los dispositivos Seeeduino LoRaWAN y LoPy, es el estudio del consumo de potencia. Para realizar este análisis se alimenta el dispositivo correspondiente con 3.3V, y se mide el consumo de corriente en dos estados: En reposo y transmitiendo un paquete haciendo uso de la tecnología LoRa.

Se debe destacar que la prueba experimental se ha realizado en igualdad de condiciones, es decir, no se ha deshabilitado la conexión *WiFi* en ninguno de los dispositivos, y no se ha usado ninguna sentencia *software* de bajo consumo en el dispositivo LoPy. Los resultados obtenidos son los que se muestran en la Tabla 5.6.

Corriente (mA)	Seeeduino LoRaWAN	LoPy
Reposo	40mA	120 mA
Transmitiendo	50 mA	190 mA

Tabla 5.6: Comparativa de consumo de corriente entre dispositivos Seeeduino LoRaWAN y LoPy.

Como se puede observar en la Tabla 5.6 el consumo de corriente del dispositivo LoPy es superior al del dispositivo Seeeduino LoRaWAN, tanto en estado de reposo como transmitiendo paquetes LoRa. No obstante, Pycom ofrece modos de bajo consumo para su dispositivo, a diferencia de Seeedstudio para el dispositivo Seeeduino LoRaWAN. La reducción del consumo de potencia en LoPy se puede realizar tanto mediante el uso de sentencias *software* como de elementos *hardware*. Según la especificación de LoPy el consumo de corriente en estos estados puede reducirse hasta el orden de microamperios.

A partir del análisis básico realizado se concluye que, técnicamente, se podría usar cualquiera de los dispositivos presentados como nodo final en la plataforma HW/SW propuesta en este TFG, pero con respecto a la implementación del *gateway*, el dispositivo LoPy cuenta con una mayor cantidad de documentación. Asimismo, la documentación a nivel general también es mayor para el dispositivo LoPy, contando además con un foro específico para desarrolladores de la tecnología que ofrece la empresa Pycom. Sin embargo, el dispositivo LoPy carece de librerías para la integración de los sensores utilizados en la solución propuesta, a diferencia del dispositivo Seeeduino LoRaWAN. También se ha visto que ambos dispositivos cuentan con los periféricos necesarios para el desarrollo de la solución propuesta, por lo que este no es un factor decisivo en la elección del dispositivo. Finalmente, el consumo de potencia es inferior en el dispositivo Seeeduino LoRaWAN, pero el dispositivo LoPy puede reducir de manera significativa su consumo de potencia en estado de reposo mediante herramientas *software* y *hardware*

que ofrece la empresa Pycom, por lo que el consumo de potencia estudiado tampoco resulta ser un factor determinante.

Teniendo en cuenta todo lo anterior, en este TFG se ha decidido finalmente adoptar el dispositivo LoPy de la empresa Pycom para el desarrollo de la plataforma HW/SW.

Una vez que se ha decidido el dispositivo que se va a utilizar como nodo final, se deben introducir algunos de los elementos *hardware* asociados a este nodo, como por ejemplo la batería. La *Expansion Board* asociada al dispositivo LoPy cuenta con un conector JST de batería LiPo. Haciendo uso de este conector se puede alimentar el dispositivo con una batería de Polímero de Litio (LiPo). La batería seleccionada en el caso de este TFG se corresponde con el modelo BAT573 de Electronic NIMO [48]. La tensión de las baterías LiPo depende del número de celdas de la batería. Una celda aporta una tensión de 3.7V, de forma que una batería de 4 celdas proporcionaría una tensión de 14.8V. La batería del modelo seleccionado cuenta con una única celda, por lo que aporta una tensión nominal de 3.7V y su capacidad es de 180 mAh.

El tamaño de la batería es adecuado para el tamaño del nodo final que se pretende implantar. En la Figura 5.12 se muestra la batería utilizada para alimentar al nodo final en el presente TFG.

Finalmente incluyendo la batería, el dispositivo LoPy, la antena asociada a dicho dispositivo, y el sensor magnetómetro, en una carcasa de protección, se puede obtener un prototipo de referencia del nodo final como el que se muestra en la Figura 5.13 y en la Figura 5.14.

Figura 5.12: Batería LiPo CS60.

Figura 5.13: Vista anterior del nodo final.

Figura 5.14: Vista posterior del nodo final.

5.4 Especificación funcional del nodo final

Uno de los puntos más importantes de una aplicación IoT-*Smart City* es el funcionamiento del nodo final. El funcionamiento del nodo final adoptado define qué acciones se deben realizar, cómo se deben realizar, y cuándo se deben realizar. Para establecer dicha funcionalidad se debe programar la plataforma en la que se implementará la funcionalidad del nodo final, que en el caso del presente TFG se corresponde con el dispositivo LoPy. Una vez programado el dispositivo, es este el que se encarga de comunicarse automáticamente con otros elementos de la red, como son el sensor y el dispositivo *gateway*. Pero antes de proceder a la codificación del nodo final, se muestra un diagrama de flujo con el funcionamiento que se pretende implementar en los nodos finales. Para desarrollar este diagrama se tienen en cuenta varios aspectos que son fundamentales para una aplicación de *Smart Parking* como la que se establece como objetivo en este TFG.

- Consumo de potencia: Se debe minimizar el consumo de potencia en todas las acciones del nodo final. Esto se debe a que el nodo se alimenta externamente mediante el uso de baterías, y por el enfoque de la aplicación se requiere que la duración de la batería sea lo más prolongada posible.

- Inactividad: En los intervalos de tiempo en los que no se envían datos desde el nodo final, se puede entender que se ha perdido la conexión o que se ha producido algún error en el funcionamiento del nodo. Por esto, resulta interesante enviar con un período fijo un tipo de información que indique que el nodo final sigue operando correctamente.

- Configuración remota del nodo: Se debe considerar la posibilidad de que se tengan que reajustar los parámetros de configuración del nodo final, dependiendo de las características particulares de cada aplicación de *Smart Parking*.

Teniendo en cuenta estos aspectos, y los requisitos de la aplicación, se especifica la funcionalidad del nodo final en la solución *Smart Parking* que se presenta en este TFG. El diagrama de flujo de la funcionalidad del nodo final se muestra en la Figura 5.15.

En el inicio del funcionamiento del nodo final, se envían una serie de las tramas de inicio. Las tramas de inicio son dos. La primera trama de inicio contiene información del nivel de la batería que alimenta al nodo final, mientras que la segunda trama indica los valores de configuración con los que se inicia el funcionamiento del nodo final.

Se recuerda que la tecnología de comunicación que se encarga de enviar las tramas es LoRa/LoRaWAN, y que el receptor inmediato de las mismas es el nodo *gateway* o nodo concentrador de la arquitectura de la red.

Tras enviar estas dos tramas, se pasa a la lectura del estado del sensor de distancia/magnetómetro. En caso de obtener una lectura correcta del valor del sensor, se envía una trama de tipo *Info Frame* con el estado inicial de la plaza de aparcamiento que se considerará inicialmente libre. Tras la primera comprobación, se almacena el estado de la plaza para que pueda ser comparado con el valor de la siguiente iteración. En otras palabras, en iteraciones posteriores se compara con el último estado de la plaza, es decir, si se está en la *iteración n* se compara el estado de la plaza en dicha *iteración n*, con el estado de la plaza obtenido en la *iteración n-1*. Si existe diferencia de estado entre las dos iteraciones se procede al envío de la trama *Info Frame*, mientras que si el estado no cambia no se envía otra trama de tipo *Info Frame*. Para determinar el estado de la plaza se hace uso de un valor umbral. Dependiendo de si se tiene inicializado el sensor de distancia o el magnetómetro, se compara el valor medido con el umbral correspondiente a cada caso. En el caso del sensor de distancia, el estado de la plaza se determina "Ocupado" si el valor medido es superior al umbral, mientras que, si el valor medido es inferior al umbral, el estado de la plaza se determina "Libre". El procedimiento con las medidas realizadas por el sensor magnetómetro es similar, pero en este caso, los valores medidos que sean superiores al umbral se interpretan como que la plaza está "Libre", y si son inferiores, como que el estado de la plaza es "Ocupado".

Así, si se produce un cambio significativo de valor en la lectura del sensor de distancia/magnetómetro, se envía un paquete del tipo *Info Frame,* que informará sobre el

estado de la plaza de *parking*. En caso negativo, se pasa a la siguiente etapa del diagrama de flujo.

En la siguiente etapa, se evalúa si el *flag keep-alive* se encuentra activado. Este *flag* se activa cuando transcurre un determinado intervalo de tiempo. El propósito de incorporar esta funcionalidad surge ante la posibilidad de que el estado de la plaza de aparcamiento no varíe en un período considerable, y se necesite informar de que el nodo sigue funcionando correctamente, aunque no haya cambiado su estado.

Este tiempo depende del modo de funcionamiento, si se está en el *modo normal*, se recomienda que sea de 2 minutos aproximadamente, mientras que, si se está en *modo nocturno*, o de baja actividad, se recomienda que se incremente el período a más de 5 minutos. No obstante, se considera la posibilidad de que se pueda configurar remotamente el valor de estos intervalos, a través de una trama de configuración del nodo.

En caso de que el *flag keep-alive* se encuentre activado, se ha de enviar un paquete para recordar que el nodo final se encuentra funcionando, pero que no ha enviado ninguna trama de información, porque el estado no ha cambiado. Este tipo de trama se denomina *Keep-Alive Frame*. Si el *flag* no está activado, se pasa al siguiente estado del diagrama de flujo.

En el siguiente estado, se comprueba si ha llegado una trama de configuración desde el nodo *gateway.* Si se confirma que sí, que ha llegado una trama de configuración, se procede a la aplicación de la configuración especificada, ajustando los parámetros configurables del nodo a los valores que se indican en el paquete recibido. En caso de que no haya llegado ninguna trama de configuración, se llega al último estado del flujo de funcionamiento.

En este último paso se comprueba si el modo de funcionamiento es el normal o es el nocturno, según la respuesta, el tiempo de espera del nodo será menor o mayor respectivamente. Al igual que el período de envío de tramas de tipo *Keep*-Alive, este tiempo de espera es configurable a través del envío de una trama de configuración, pero se estima que la espera normal ronde los 2 minutos, y la espera nocturna los 5 minutos. Tras cada envío o recepción de trama el nodo final, el tiempo de espera siempre es el

normal. En este estado, el nodo se encuentra inactivo y no realiza ninguna tarea, facilitando de esta manera el ahorro de consumo de potencia.

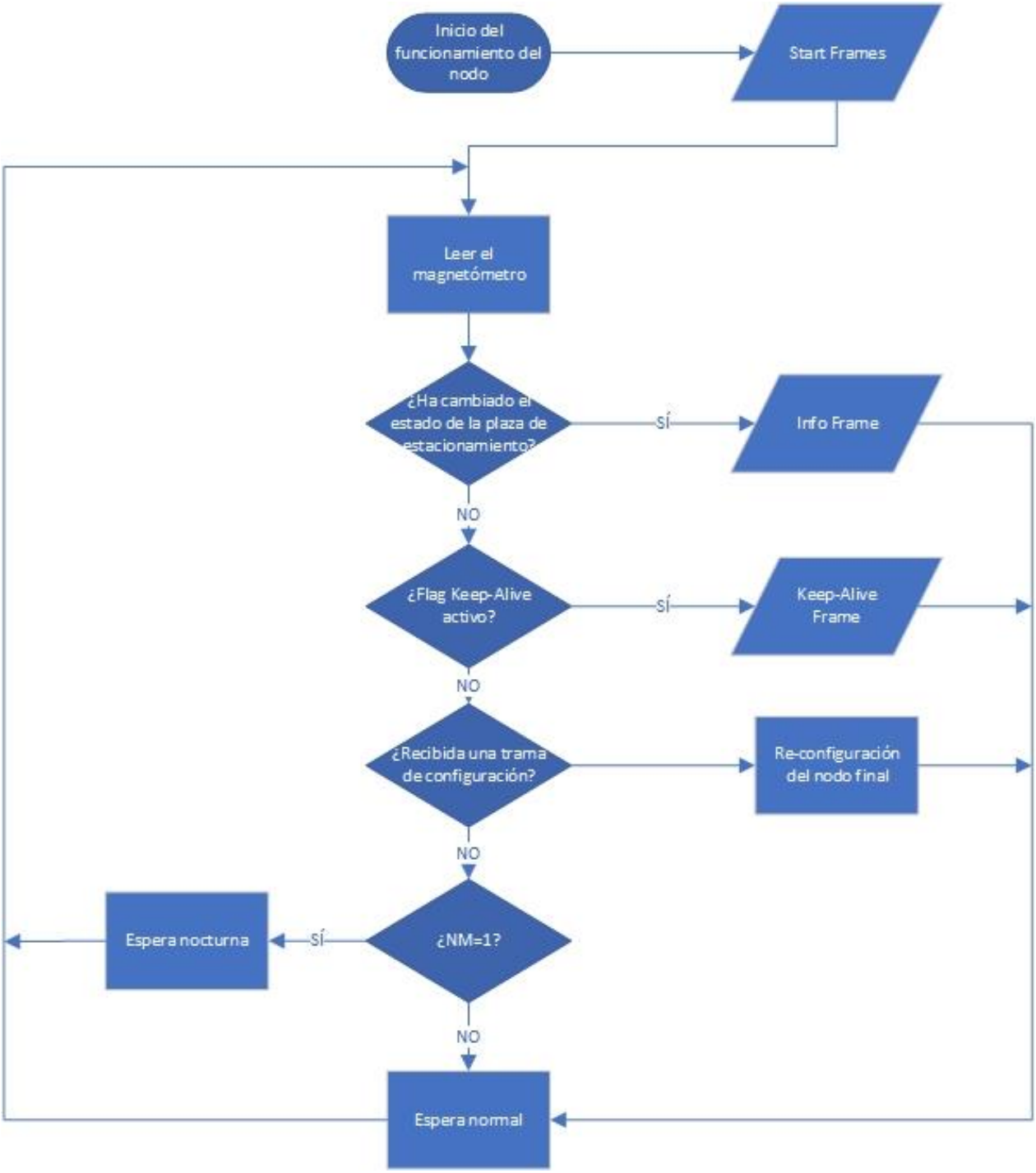


Figura 5.15:Diagrama de flujo del funcionamiento del nodo final.

5.4.1 Formato de los paquetes

En el formato de los diferentes tipos de trama, se ha establecido que los 2 primeros bytes de cada una de ellas sean comunes a todos los tipos de tramas.

Así, el Byte 0 en todas las tramas contiene la información mostrada en la Tabla 5.7.

Bit	Nombre	Descripción
7	Estado de la plaza de aparcamiento (solo para *Info Frame*)	'0' indica que la plaza está libre.
		'1' indica que la plaza está ocupada.
6	Estado de la batería	'0' indica que la batería tiene un nivel de carga suficiente.
		'1' indica que la batería tiene un bajo nivel de carga y necesita ser cambiada. Un nodo con un bajo nivel de carga en su batería puede no funcionar correctamente.
5-4	Reservados	Bits no utilizados en esta versión del nodo final.
3	Tipo de trama	0d(0000b) – *Info frame*
2		1d(0001b) – *Keep-Alive frame*
1		4d(0100b) – *Start frame 1*
0		5d(0101b) – *Start frame 2* Valores del 6 al 15 están reservados.

Tabla 5.7: Contenido del Byte 0 de las tramas.

Por otra parte, el Byte 1 representa un contador de tramas que va de 0 a 255, cuyo propósito es servir de mecanismo para detectar que no se hayan perdido tramas en la transmisión.

En concreto, estos dos bytes indican el tipo de trama que se envía (Byte 0) y el orden en el que son enviadas (Byte 1).

Una vez conocidos los dos bytes que siempre estarán en todas las tramas enviadas, se puede proceder a describir por separado cada uno de los tipos de tramas que se pueden enviar desde el nodo final hacia el *gateway*.

Como se comentó anteriormente, al iniciar el funcionamiento de un nodo final se envían dos tramas de inicio o *Start Frames*. Estas tramas contienen información sobre el estado y la configuración inicial del nodo. El contenido de estas tramas se detalla a continuación.

En la primera trama de inicio, o *Start Frame 1*, se han definido los campos mostrados en la Tabla 5.8.

Byte	Nombre	Descripción
0	Datos de la trama	Información de la trama.
1	Contador de tramas	Número de trama.
2-4	Reservados	Bits no utilizados en esta versión del nodo final.
5	ID	Identificador del nodo.
6-8	Reservados	Bits no utilizados en esta versión del nodo final.
9	Voltaje de la batería MSB	Voltaje de la batería en *mV*. Valor en 16 bits.
10	Voltaje de la batería LSB	

Tabla 5.8: Contenido de las tramas de tipo Start Frame 1.

En la segunda trama de inicio, o *Start Frame 2*, se incluyen los campos mostrados en la Tabla 5.9.

Byte	Nombre	Descripción
0	Datos de la trama	Información de la trama.
1	Contador de tramas	Número de trama.
2	Versión del firmware	Número de versión del firmware.
3	NM Start	Hora de inicio del modo nocturno.
4	NM Period	Duración del modo nocturno en horas.
5	ID	Identificador del nodo.
6	NM Keep Alive	Periodo de tiempo entre dos tramas *Keep-Alive* durante el modo nocturno.
7	NM Sleep Time	Tiempo de espera(dormido) del nodo durante el modo nocturno.
8	Sleep Time	Tiempo de espera(dormido) del nodo durante el modo normal.

9	Keep Alive	Periodo de tiempo entre dos tramas *Keep-Alive* durante el modo normal.
10	Reservado	Bit no utilizado en esta versión del nodo final.

Tabla 5.9: Contenido de las tramas de tipo Start Frame 2.

Por otro lado, cuando se detecta un cambio en el estado de la plaza de estacionamiento, se envía una trama de información o *Info Frame*, cuyo contenido se muestra en la Tabla 5.10.

Byte	Nombre	Descripción
0	Datos de la trama	Información de la trama.
1	Contador de tramas	Número de trama.
2	Zproxi/XmagnetoGSB	Valor medido por el sensor de proximidad. Si está funcionando el sensor magnetómetro coge el valor más significativo de la medida del magnetómetro (GSB).
3	XmagnetoMSB	Valor intermedio de la medida del magnetómetro (MSB). Si está funcionando el sensor de proximidad se mantiene a '0'.
4	XmagnetoLSB	Valor menos significativo de la medida del magnetómetro (LSB). Si está funcionando el sensor de proximidad se mantiene a '0'.
5	ID	Identificador del nodo.
6-10	Reservados	Bits no utilizados en esta versión del nodo final.

Tabla 5.10: Contenido de las tramas de tipo Info Frame.

Para indicar que el nodo final sigue funcionando correctamente, se envían periódicamente tramas de tipo *Keep-Alive*, que además del significado implícito a su recepción, recogen el tiempo transcurrido desde que se inició el funcionamiento del nodo final. El contenido de las tramas de tipo *Keep-Alive* es el mostrado en la Tabla 5.11.

Byte	Nombre	Descripción
0	Datos de la trama	Información de la trama.
1	Contador de tramas	Número de trama.
2	Tiempo transcurrido (hh)	Horas transcurridas desde que se inició el funcionamiento del nodo.
3	Tiempo transcurrido (mm)	Minutos transcurridos desde que se inició el funcionamiento del nodo. Se acumulan, es decir, si lleva 2h y 10min funcionando aparecerá en este campo el valor: 130.
4	Reservado	Bit no utilizado en esta versión del nodo final.
5	ID	Identificador del nodo.
6-10	Reservados	Bits no utilizados en esta versión del nodo final.

Tabla 5.11: Contenido de las tramas de tipo Keep-Alive Frame.

Otro tipo de trama definida en la aplicación desarrollada en este TFG es la trama de configuración. A diferencia de las tramas anteriores, esta trama se envía desde el *gateway*, y el nodo receptor es el nodo final. En esta trama se incluyen valores de configuración que debe adoptar el nodo final.

Para indicar qué nodo se desea configurar se debe introducir en el campo *NodoToConfig* el ID del nodo que se desea configurar. En el nodo final, se filtran las tramas, y solo se atienden las que lleven en el campo *NodoToConfig* el identificador asociado al nodo final.

En la Tabla 5.12 se muestran los campos definidos en una trama de tipo *Configuration*.

Byte	Nombre	Descripción
0	Datos de la trama	Información principal de la trama de configuración.
1	ID nodo	Identificador que se va a asignar al nodo (En caso de que el bit 6 del Byte 0 esté a ‘1’).
2	Versión del	Número de versión del f*irmware* actualizado.

	firmware	
3	NM Start	Hora de inicio del modo nocturno.
4	NM Period	Duración del modo nocturno en horas.
5	NM Sleep Time	Tiempo de espera(dormido) del nodo durante el modo nocturno.
6	NM Keep Alive	Periodo de tiempo entre dos tramas *Keep-Alive* durante el modo nocturno.
7	NodoToConfig	0xXX- ID del nodo que se desea configurar.
8	Sleep Time	Tiempo de espera(dormido) del nodo durante el modo normal.
9	Keep Alive	Periodo de tiempo entre dos tramas *Keep-Alive* durante el modo normal.
10	Threshold	Umbral para detectar cambios en la plaza de aparcamiento (mínima diferencia entre dos lecturas consecutivas para detectar cambio).

Tabla 5.12: Contenido de las tramas de tipo Configuration Frame.

En el caso de las tramas de tipo *Configuration Frame*, el Byte 0 es diferente al del resto de las tramas, ya que se requiere del uso de varios bits para identificar el tipo de trama, es decir, que el receptor de la trama pueda diferenciar una trama de configuración de otros paquetes recibidos. Además, al tratarse de una trama que se envía en el sentido inverso al de la comunicación utilizada en el envío de las otras tramas, no se adjunta en el paquete de configuración información del estado del nodo, como si ocurría en el Byte 0 del resto de tramas. El Byte 0 de la trama de configuración contiene la información mostrada en la Tabla 5.13.

Bit	Nombre	Descripción
7	Reservado	Bit no utilizado en esta versión del nodo final.
6	Asignar ID	‘1’ indica que se desea asignar un ID al nodo, ‘0’ que no. El

		ID a asignar se ubica en el Byte 1.
5	Código de configuración	Se fija el código a '110011'. Este código se comprueba en el nodo final, de manera que si se recibe este código, se interpreta esta trama como de configuración, y en caso contrario, se descarta.
4		
3		
2		
1		
0		

Tabla 5.13: Contenido del Byte 0 de las tramas de tipo Configuration Frame.

5.5 Codificación del nodo final

Para la programación del dispositivo LoPy se utiliza el editor Atom [49]. Se trata de un editor de código abierto para macOS, Linux, y Windows con soporte para *plugins* codificados en *Node.js* y control de versiones *Git* integrado, desarrollado por GitHub [50].

Para poder trabajar en este editor con los dispositivos de Pycom es necesario instalar el *plugin* Pymakr [51]. Este paquete añade una consola *Read-Eval-Print-Loop* (REPL) al editor Atom que permite la conexión con los dispositivos Pycom y ejecutar código en ellos. La principal diferencia entre una consola REPL y la ejecución de un *script*, es que en REPL múltiples clases o funciones imprimen por pantalla automáticamente su respuesta, mientras que en el caso de los *scripts*, no se muestra necesariamente la respuesta por pantalla [52].

Con el editor y el *plugin* instalado ya se puede proceder a la programación de la funcionalidad del dispositivo. En la Figura 5.16 se muestra la interfaz principal del entorno Atom. En la interfaz mostrada se puede ver que en la zona superior izquierda se encuentra la barra de herramientas que permite gestionar el entorno. También se cuenta con un espacio para visualizar el proyecto que se encuentra abierto, con sus respectivos *scripts*. La mayor parte del espacio habilitado se destina a la edición de los *scripts*. Por otra parte, en la zona inferior de la interfaz se puede visualizar la barra de herramientas

asociada al *plugin Pymakr*. Por último, debajo de la barra de herramientas *Pymakr* se muestra la ventana de comandos, en la que se pueden ejecutar instrucciones y aparece la respuesta del *software* correspondiente.

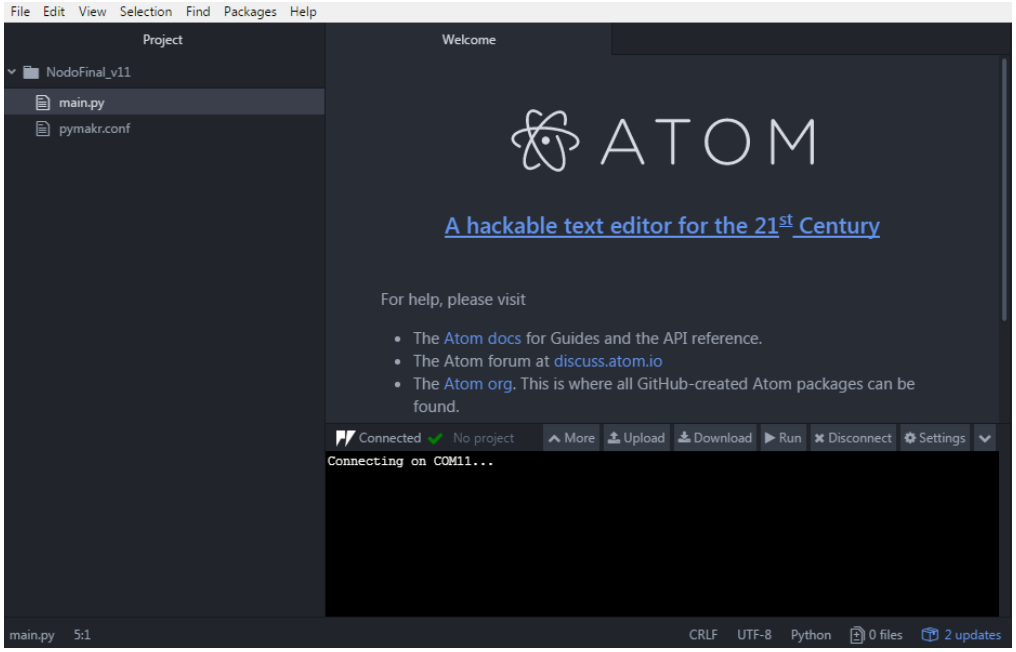


Figura 5.16: Interfaz principal del entorno Atom.

El código del nodo final se ha desarrollado tomando como referencia el diagrama de flujo presentado en el apartado anterior. El entorno utilizado para realizar la codificación del nodo final ha sido Atom [49], y el lenguaje de programación del dispositivo utilizado para la implementación de este nivel de la solución *Smart Parking,* ha sido *MicroPython*.

Lo primero que se encuentra en el código desarrollado son las librerías importadas, fundamentales para poder hacer uso de determinadas funcionalidades en el desarrollo del programa en el lenguaje de programación *MicroPython*. Las librerías importadas en el código del nodo final son las que se muestran en la Figura 5.17. Como se puede ver en esta captura, se incluyen librerías para la gestión de variables de tipo *array* o binarias, para la gestión de las funciones temporales, o para la gestión de los periféricos. En concreto, los periféricos que requieren del uso de una librería son *Timer*, ADC, RTC e I^2C. Además, se cuenta con la librería que incorpora todas las funciones para el uso de la tecnología LoRa/LoRaWAN. Las funciones utilizadas en el código desarrollado se han declarado, o bien en estas librerías, o bien en el propio código de la aplicación propuesta.

```
########################################Import#####################
from network import LoRa
import socket
import machine
from machine import Timer
import time
import binascii
from machine import RTC
from array import array
from machine import ADC
from machine import I2C
```

Figura 5.17: Importación de librerías.

Aparte de la llamada a las librerías de Pycom, también se deben desarrollar en el mismo código las librerías asociadas al sensor de proximidad y al sensor magnetómetro. Partiendo de la caracterización realizada en el capítulo 4 *Sensores del nodo final*, se presenta a continuación el código desarrollado como librerías en *MicroPython* para ambos sensores.

5.5.1 Librería Sensor *ZX Distance and Gesture*

Conociendo el mapa de registros y el funcionamiento del sensor, se puede proceder a la codificación de una librería que permita establecer la comunicación I^2C entre el sensor y el dispositivo asociado al nodo final, que en este caso es el dispositivo LoPy.

Para poder desarrollar esta librería en lenguaje *MicroPython* se debe asegurar que se encuentran declaradas las librerías que permiten la gestión de la interfaz I^2C y de funciones temporales, que son las que se muestran en la Figura 5.18.

```
1  from machine import I2C
2  import time
3
4  ADDR = 0x10
```

Figura 5.18: Librerías de I^2C y de tiempo utilizadas en el desarrollo de la librería del sensor ZX Distance and Gesture Sensor.

En la misma Figura 5.18 se puede ver que la dirección a utilizar toma el valor 0x10, ya que el *Jumper I^2C Addr* se encuentra abierto, como se había indicado con anterioridad.

Con la llamada a estas librerías, y con la creación de la constante que contiene la dirección I^2C del sensor, ya se pueden empezar a programar las funciones necesarias para poder establecer la comunicación entre el dispositivo LoPy y el sensor. Con esta finalidad se crea una clase que contenga todas las funciones requeridas, como se muestra en la Figura 5.19.

La clase creada se denomina ZXSENSOR, y en ella se incluyen las siguientes funciones: *__init__()*, *position_available()*, *get_x()*, *get_z()* y *get_position()*. A continuación, se describen cada una de estas funciones.

En la función de inicialización *__init__()* se inicia la comunicación I^2C a partir de la dirección del periférico I^2C del dispositivo LoPy y la dirección I^2C del sensor. Además, se almacena en el registro una variable del tipo *bytearray* de longitud 1 para almacenar los cambios en el registro que se irán dando durante el funcionamiento del sensor.

En la función *position_available()* se realiza una lectura del registro STATUS asociado a la dirección 0x00 del sensor. La función devuelve *True* si la lectura muestra un "1", y *False* si muestra un "0".

Las siguientes funciones implementadas son *get_x()* y *get_z()*. En estas funciones se procede a la lectura de los registros XPOS (0x08) y ZPOS (0x0A), respectivamente, para recuperar el valor de la medida realizada por el sensor en las coordenadas X y Z, respectivamente. Los valores consultados en las lecturas del registro son devueltos por estas funciones.

Por último, se analiza la función *get_position()*, en la que se devuelve la llamada a las funciones *get_x()* y *get_z()*, funciones que a su vez devuelven los valores medidos en las coordenadas X y Z, respectivamente.

Con el desarrollo de esta librería ya se puede gestionar el uso del sensor *ZX Distance and Gesture Sensor* desde el dispositivo LoPy, para lo cual es necesario inicializar previamente la comunicación I^2C mediante las sentencias de código que se muestra en la Figura 5.20.

```
class ZXSENSOR:
    def __init__(self, i2c, addr):
        self.i2c = i2c
        self.addr = addr
        self.reg = bytearray(1)

    def position_available(self):
        self.i2c.readfrom_mem_into(self.addr, 0x00, self.reg)
        return (self.reg[0] & 0b00000001) > 0

    def get_x(self):
        self.i2c.readfrom_mem_into(self.addr, 0x08, self.reg)
        return self.reg[0]

    def get_z(self):
        self.i2c.readfrom_mem_into(self.addr, 0x0a, self.reg)
        return self.reg[0]

    def get_position(self):
        return zx.get_z(), zx.get_x()
```

Figura 5.19: Clase ZXSENSOR y funciones asociadas a dicha clase.

```
i2c = I2C(0, I2C.MASTER, baudrate=100000)
zx = ZXSENSOR(i2c, addr=ADDR)
```

Figura 5.20: Inicialización I^2C y creación de un objeto de la clase ZXSENSOR.

Usando el objeto creado se pueden acceder a todas las funciones declaradas dentro de la clase.

5.5.2 Librería Sensor *QMC5883L*

Teniendo ya los conocimientos suficientes sobre el sensor magnetómetro, se puede desarrollar una librería en lenguaje *MicroPython* que permita comunicar el nodo final LoPy con el sensor QMC5883L. Esta librería se ha desarrollado siguiendo la funcionalidad explicada anteriormente, y que se recoge en el *datasheet* proporcionado por el fabricante.

Lo primero que se debe hacer antes de empezar a codificar la librería es asegurarse de que se encuentran declaradas las librerías propias de Pycom que se usarán en las

funciones que permiten la comunicación entre nodo final y el sensor. Las librerías importadas son las que se muestran en la Figura 5.21. Estas librerías han sido desarrolladas por Pycom para todos sus dispositivos, y en concreto se usan las que contienen las funciones que permiten gestionar la comunicación I^2C, y variables en binario, así como las funciones relativas a la gestión del tiempo.

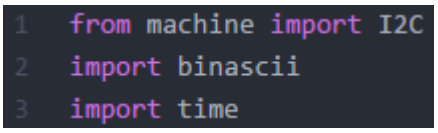


Figura 5.21: Librerías importadas en el desarrollo de la librería del sensor QMC5883L.

También se deben declarar las constantes que se usan en la gestión de los registros. Estas constantes contienen los diferentes valores con los que se pueden comparar y escribir los registros del magnetómetro. Los valores de estas constantes, por tanto, se obtienen consultando el *datasheet* del fabricante. En la Figura 5.22 se adjunta el valor de cada constante.

Tras importar las librerías necesarias, y declarar las constantes correspondientes a los registros, se declara la clase QMC5883L con las funciones que permiten comunicar al nodo final con el sensor magnetómetro. Esta declaración, y las primeras funciones codificadas, se muestran en la Figura 5.23.

Así, en la función *__init__()* se almacenan las direcciones que se pasan por parámetros, y además se declara un *bytearray* de longitud 1. Es la función que se utiliza cuando se crea un objeto de la clase QMC5883L.

En la función *initialize()* se lee el valor del registro *Chip ID* y se compara con el identificador esperado para comprobar que se ha conectado correctamente el magnetómetro al dispositivo LoPy. Si está conectado, se configura el registro *SET/RESET Period* con el valor recomendado. Tanto en el caso de que se haya conectado correctamente el magnetómetro, como si no, se muestran mensajes en pantalla que informen de la situación.

```
_QMC5883L_I2C_ADDRESS = const(0x0D)

_QMC5883L_CHIPID_REG = const(0x0D)
_QMC5883L_CHIPID_REG_RSP = const(0xFF)

_QMC5883L_SOFTRESET_REG =  const(0x0A)
_QMC5883L_SETRESET_REG = const(0x0B)
_QMC5883L_CTRL_REG1 = const(0x09)
_QMC5883L_STATUS_REG = const(0x06)
_QMC5883L_X_AXIS = const(0x00)
_QMC5883L_Y_AXIS = const(0x02)
_QMC5883L_Z_AXIS = const(0x04)

_QMC5883L_Mode_Standby = const(0x00)
_QMC5883L_Mode_Continuos = const(0x01)

_QMC5883L_ODR_10Hz = const(0x00)
_QMC5883L_ODR_50Hz = const(0x04)
_QMC5883L_ODR_100Hz = const(0x08)
_QMC5883L_ODR_200Hz = const(0x0C)
_QMC5883L_RNG_2G = const(0x00)
_QMC5883L_RNG_8G = const(0x10)
_QMC5883L_OSR_512 = const(0x00)
_QMC5883L_OSR_256 = const(0x40)
_QMC5883L_OSR_128 = const(0x80)
_QMC5883L_OSR_64 = const(0xC0)
```

Figura 5.22: Variables utilizadas para la librería del sensor QMC5833L.

En la tercera función que se puede ver en la Figura 5.23, *softReset()*, se pasa al registro, *Control Register 2*, el valor 0x80, correspondiente a la activación de la opción de *reset* por *software*. En la Figura 5.24 se muestra el código correspondiente a la implementación del resto de las funciones implementadas en esta librería.

Así, en la función *setMode()* se selecciona la configuración deseada para el sensor magnetómetro a través de los parámetros que se pasan a la función. Estos parámetros se escriben sobre el registro *Control Register 1*.

En la Figura 5.24 también se adjunta el código de la función *dataReady(),* una función que devuelve "True" en caso de que exista un nuevo valor para ser leído desde el sensor. Para ello consulta el registro *Status Register*.

Por último, se analiza la función *readMag()*, que es la función que consulta los 6 registros asociados a los valores del magnetómetro y lee los valores correspondientes a las medidas realizadas. También acondiciona la lectura realizada para poder devolver el valor de cada una de las coordenadas.

A partir de estas funciones, que se recogen en la clase QMC5883L, se puede establecer una comunicación entre el nodo final implementado en el dispositivo LoPy, y el sensor magnetómetro. No obstante, antes de utilizar las funciones asociadas al magnetómetro, se debe inicializar la comunicación I^2C, crear un objeto de la clase QMC5883L, e inicializar dicho objeto con la función *initialize()*. Esta inicialización se muestra en la Figura 5.25.

Por último, se debe apuntar que cuando se vaya a recoger un dato en el programa usado en el nodo final, se deberá cambiar del modo *Standby* al modo *Continuous*, y cuando se haya realizado la medida se deberá hacer una inicialización por *software* para que el estado del magnetómetro vuelva a estar en el modo *Standby*, donde el consumo de potencia es inferior. Este procedimiento se muestra en el Figura 5.26, donde además en la función *setMode()* se pasan por parámetro los valores de configuración deseados, lo que representa un ejemplo de posible uso de la librería desarrollada.

```
class QMC5883L:
    def __init__(self, i2c, addr=_QMC5883L_I2C_ADDRESS):
        self.i2c = i2c
        self.addr = addr
        self.value = 0
        self.reg = bytearray(1)


    def initialize(self):
        print("QMC5883L_initialize")
        chipID = self.i2c.readfrom_mem(self.addr, _QMC5883L_CHIPID_REG, 1)
        print(binascii.hexlify(chipID))
        if chipID[0] == _QMC5883L_CHIPID_REG_RSP:
            print("QMC5883L connected!");
            self.i2c.writeto_mem(self.addr, _QMC5883L_SETRESET_REG, 0x01)
            return True
        else:
            print("Could not find QMC5883L connected!");
            return False


    def softReset(self):
        print("QMC5883L_softReset")
        self.i2c.writeto_mem(self.addr, _QMC5883L_SOFTRESET_REG, 0x80)

```

Figura 5.23: Clase QMC5883L y funciones _init_(), initialize() y softReset().

```
    def setMode(self, mode, odr, rng, osr):
        print("QMC5883L_setMode : ")
        self.reg[0] = mode | odr | rng | osr
        print(self.reg[0])
        self.i2c.writeto_mem(self.addr, _QMC5883L_CTRL_REG1, self.reg)


    def dataReady(self):
        print("QMC5883L_dataReady")
        self.i2c.readfrom_mem_into(self.addr, _QMC5883L_STATUS_REG, self.reg)
        return(self.reg[0] & 0x01)


    def readMag(self):
        print("QMC5883L_readMag")
        data = bytearray(6)
        data = self.i2c.readfrom_mem(self.addr, _QMC5883L_X_AXIS, 6)
        x_t = (data[1] << 8) + data[0]
        y_t = (data[3] << 8) + data[2]
        z_t = (data[5] << 8) + data[4]
        return x_t, y_t, z_t
```

Figura 5.24: Funciones setMode(), dataReady() y readMag().

```
85  i2c = I2C(0, I2C.MASTER, baudrate=100000)
86  qmc5883l_m = QMC5883L(i2c, addr=_QMC5883L_I2C_ADDRESS)
87  qmc5883l_m.initialize()
```

Figura 5.25: Inicializaciones para la comunicación con el sensor magnetómetro.

```
91  while(True):
92      qmc5883l_m.setMode(_QMC5883L_Mode_Continuos, _QMC5883L_ODR_200Hz, _QMC5883L_RNG_2G, _QMC5883L_OSR_256)
93      time.sleep_ms(1000)
94      if qmc5883l_m.dataReady():
95          x, y, z = qmc5883l_m.readMag()
96          print(x)
97          print(y)
98          print(z)
99          print("---------")
100         qmc5883l_m.softReset()
101         time.sleep(5)
102     else:
103           print("--------- not qmc5883l_m.dataReady")
104           time.sleep_ms(250)
105
```

Figura 5.26: Ejemplo de uso de la librería QMC5883L.

5.5.3 Codificación de la funcionalidad del nodo final

El siguiente paso tras implementar y declarar las librerías asociadas al sensor de proximidad y al sensor magnetómetro, consiste en establecer la configuración de la tecnología de comunicación utilizada, en este caso LoRa/LoRaWAN. Como se ha comentado en otros apartados, se pretende que el nodo final se conecte a una red LoRaWAN para comunicarse con el nodo *gateway*. Para unirse a esta red se debe realizar un proceso de activación, bien mediante ABP o mediante OTAA. En el caso particular de este TFG, el modo de activación utilizado ha sido OTAA, por lo que se debe realizar una asignación de claves para el dispositivo utilizado. Estas claves deben estar presentes, tanto en la plataforma *The Things Network* (TTN), como en el código del nodo final. Las tres claves necesarias son las siguientes:

- *DevEUI*: Es el identificador del dispositivo en la red LoRaWAN. Este identificador puede ser editado por el usuario y debe copiarse en el campo *DevEUI* de uno de los dispositivos asociados a la aplicación que se encuentra en la plataforma TTN.

- *AppKEY*: Es la clave de la aplicación que genera automáticamente TTN al crear una aplicación en su plataforma.

- *AppEUI*: Es el identificador de la aplicación, es otro campo que genera automáticamente TTN.

En la Figura 5.27 se puede observar que el primer paso para la configuración de la tecnología de comunicación es elegir el modo de funcionamiento, pudiendo ser este, LoRa o LoRaWAN. Seguidamente, se introducen las claves necesarias para la activación OTAA, indicadas anteriormente. En esta misma Figura 5.27, se observa cómo se añaden los tres canales por defecto que se reservan para la comunicación, todos ellos en la misma frecuencia. También se limitan los posibles valores de *Data Rate* que pueden utilizar los canales por defecto de '0' a '5'.

```
########################################Configuración LoRa/LoRaWAN###################################
# Se inicializa LoRa en el modo LoRaWAN
# Se pueden añadir más parámteros como la frecuencia, la potencia de transmisión o el Spreading Factor
lora = LoRa(mode=LoRa.LORAWAN)

# Se crean los parámteros de autenticación para la activación OTAA
dev_eui = binascii.unhexlify('70B3D54997783F1E'.replace(' ','')) #lopy-device-fbt-2
app_key = binascii.unhexlify('0C546F91E25899E5E283A27D5A44BF50'.replace(' ','')) #lopy-device-fbt-1
app_eui = binascii.unhexlify('70B3D57ED0007139'.replace(' ',''))

# Se dejan los 3 canales por defecto en la misma frecuencia. (Se debe hacer antes del join-request)
lora.add_channel(0, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(1, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(2, frequency=868100000, dr_min=0, dr_max=5)
```

Figura 5.27: Modo de funcionamiento LoRa/LoRaWAN y claves de activación OTAA.

Una vez que se tiene configurado el modo de funcionamiento, las claves de activación OTAA, y los canales por defecto, se puede pasar al envío de la solicitud de *Join-request*. Con este mensaje que se envía al nodo *gateway*, se comprueba si el nodo final que envía la solicitud de conexión está configurado en la plataforma TTN. En caso positivo, el nodo final y el nodo *gateway* ya pueden comunicarse de forma continua, mientras que si las claves no coinciden, el nodo final se mantiene enviando solicitudes, aunque no vaya a ser aceptado. Así, tal y como se muestra en la Figura 5.28, el nodo final se mantiene en un bucle hasta que se acepta su solicitud. Cabe destacar que se ha realizado el bucle porque debido a problemas de conexión entre el nodo *gateway* o el servidor, puede que no se

procesen todas las solicitudes de activación, por lo que se reintenta continuamente el establecimiento de la conexión. No obstante, si las claves introducidas en el nodo final y las definidas en la plataforma TTN no coinciden, esta activación nunca llegará a producirse, por lo que la espera infinita del nodo final por la aceptación de solicitud se puede interpretar como un error en el proceso de activación.

```
# Se envía el join-request a la red vía OTAA
lora.join(activation=LoRa.OTAA, auth=(dev_eui, app_eui, app_key), timeout=0, dr=5)

# Se mantiene en espera hasta que se acepte el join.request
while not lora.has_joined():
    time.sleep(2.5)
    print('Not joined yet...')

print('Joined!')

# Se borran todos los canales que no sean por defecto
for i in range(3, 16):
    lora.remove_channel(i)

# Se crea un socket LoRa
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# Se configura el data rate de LoRaWAN
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)
```

Figura 5.28: Join-request, eliminación de canales no prestablecidos y configuración del socket.

En la Figura 5.28 también se puede observar cómo se eliminan los canales que no sean los establecidos por defecto, además de crear un *socket* y configurarlo con un *Data Rate* determinado, en este caso con un valor de 5.

En el desarrollo de la funcionalidad del nodo, se utiliza el *socket* que se acaba de crear y configurar, siendo la manera correcta de utilizarlo la siguiente: si se desea recibir datos procedentes del nodo *gateway*, se debe bloquear el *socket*, mientras que, para poder enviar, se debe desbloquear. El código asociado a las funciones de bloqueo y desbloqueo, es el que se muestra en la Figura 5.29.

```
# Se bloquea el socket
s.setblocking(False)
# Se desbloquea el socket
s.setblocking(True)
```

Figura 5.29: Bloqueo y desbloqueo del socket.

En este caso, como el primer proceso que se va a realizar, según el diagrama que define el funcionamiento del nodo final, es enviar las tramas de inicio, se deja el *socket* desbloqueado.

Por otro lado, en el código desarrollado, se utilizan variables para la gestión de diferentes valores determinantes en el funcionamiento del nodo final. En la Figura 5.30, la Figura 5.31, la Figura 5.32 y la Figura 5.33 se muestran variables relacionadas con diferentes funcionalidades dentro del código, como son: los niveles de batería, la gestión del convertidor ADC, diversos identificadores, la gestión de *Timers*, de los sensores, RTC, y parámetros de configuración, con los valores establecidos en todo caso por defecto.

```
##########################################Variables#################
#Número de lecturas del ADC
numADCreadings = const(100)

#Nivel de batería
batteryLevel = 0
#Nivel de batería binario
binBatteryLevel = 0
binBatteryLevelMSB = 0
binBatteryLevelLSB = 0

#Estado de la batería; 0x00:Nivel adecuado, 0x40:Nivel bajo
batteryStatus = 0x00

#Valor mínimo que debe tener la batería
minCharge = 660
```

Figura 5.30: Variables utilizadas en la gestión del ADC y en los niveles de batería.

En la Figura 5.34 se puede observar la creación y configuración de los objetos asociados al convertidor ADC, al RTC y a los *Timers*. Además, se crea un *bytearray* de 11 bytes para rellenar con el contenido de las tramas.

```
#ID del nodo
id1=0x05

#Contador para el frameCounter
contador = 0

#Variables auxiliares para el cálculo de current hours and minutes
total = 0
totalh = 0
totalm = 0

#Modo nocturno(NM=1) o normal(NM=0)
NM = 0
```

Figura 5.31: Variables para la gestión del identificador del nodo, del tiempo transcurrido y del modo de funcionamiento.

```
#Variables iniciales del RTC
year = 2018
month = 2
day = 22
hour = 15
minutes = 56
seconds = 0
usec = 0

#configuración inicial del nodo final
#versión del firmware
version = 01
#hora a la que arranca el modo nocturno. Tipo:24horas
NMStart = 16
#Duración del modo nocturno. (En horas)
NMPeriod = 6
#Intervalo de tiempo entre dos tramas Keep-Alive en el modo nocturno. 4h --> 14400s
NMKeepAlive = 14400
#Intervalo de tiempo en el que el nodo está "dormido" en el modo nocturno. 5min --> 300s
NMSleepTime = 300
#Intervalo de tiempo entre dos tramas Keep-Alive en el modo normal. 2h --> 7200s
KeepAlive = 60
#Intervalo de tiempo en el que el nodo está "dormido" en el modo normal. 2min --> 120s
SleepTime = 20
```

Figura 5.32: Variables para la gestión del RTC y de los parámetros de configuración.

```
############################Gestión sensores#############################################
#Número de lecturas que se realizan en el arranque del nodo para estabilizar el magnetómetro
contadorMagneto = 3
#FuncionamientoNormal = 0-> Necesita estabilizarse, FuncionamientoNormal = 1-> Medidas fiables
FuncionamientoNormal = 0
#Inicialización variables de los 3 ejes del magnetómetro
x = 0
y = 0
z = 0
#Inicialización variable auxiliar
xprev = 0
#Inicialización de variables de conversión a binario
xprevbin = 0
xprevbinGSB = 0
xprevbinMSB = 0
xprevbinLSB = 0
#Umbrales
umbralProxi = 100 #Sensor de proximidad
umbralMagneto = 10000 #Sensor magnetómetro
#Umbral para detectar cambios en la plaza de aparcamiento
#(Rango threshold: 0-255)*79 =(Rango umbralMagneto: 0-20000)
threshold = 1
#Inicialmente esta variable indica que la plaza está ocupada para que envíe
#un primer InfoFrame al estar libre, que es como se presupone el estado de la plaza.
estadoParking = 1
```

Figura 5.33: Variables para la gestión del magnetómetro.

Como parte de las inicializaciones se debe iniciar la comunicación I^2C con el sensor seleccionado. La selección del sensor que se va a utilizar en el nodo final debe reflejarse en la variable *TipoSensor* que se muestra en la Figura 5.35. En la misma Figura 5.35 se observa la función *tipoSensor()*, que se encarga de inicializar la comunicación I^2C con un sensor u otro dependiendo del valor que tenga la variable *TipoSensor*. Si el valor de esta variable es "1" se iniciará la comunicación I^2C con el sensor de proximidad, mientras que si es "0" se iniciará con el sensor magnetómetro.

Para la gestión de las variables y de las distintas funcionalidades que incorpora el nodo final, se han programado funciones que se encargan de esta tarea.

Así, en la Figura 5.36 se adjunta el código correspondiente al incremento del valor del campo *FrameCounter* utilizado en todas las tramas. Cada vez que se llama a esta función se incrementa el valor de la variable *contador* y se asegura que no supere los 8 bits. Es una función que permite confirmar que no se han producido errores en la transmisión o en la recepción de las tramas, y que han llegado todas. Es decir, si en la recepción de las

tramas llegan dos paquetes seguidos con valores de *FrameCounter* no consecutivos, significa que se ha perdido un paquete.

```
########################################Inicializaciones#######

#Creamos e inicializamos el ADC
adc = ADC()
batt = adc.channel(attn=ADC.ATTN_11DB, pin='P16')
batt.value()

#Creamos objeto rtc y lo inicializamos con un valor determinado
rtc = RTC()
rtc.init((year, month, day, hour, minutes, seconds, usec, 0))
rtc.ntp_sync("pool.ntp.org") #Sincronización

#Inicialización I2C y asociación al magnetómetro QMC5883L
i2c = I2C(0, I2C.MASTER, baudrate=100000)
qmc5883l_m = QMC5883L(i2c, addr=_QMC5883L_I2C_ADDRESS)
qmc5883l_m.initialize()

#Creamos dos objetos Timer
chrono = Timer.Chrono()
k = Timer.Chrono()

#Se inicia la cuenta para ambos
chrono.start()
k.start()

#Se crea un bytearray para la gestión de las tramas
frame = bytearray(11)
```

Figura 5.34: Inicializaciones de diferentes funcionalidades.

```
def tipoSensor(tipo):
    global zx
    global qmc5883l_m
    if(tipo == 1):
     i2c = I2C(0, I2C.MASTER, baudrate=100000)
     zx = ZXSENSOR(i2c, addr=ADDR)

    if(tipo == 0):#Inicialización I2C y asociación al magnetómetro QMC5883L
     i2c = I2C(0, I2C.MASTER, baudrate=100000)
     qmc5883l_m = QMC5883L(i2c, addr=_QMC5883L_I2C_ADDRESS)
     qmc5883l_m.initialize()
```

Figura 5.35: Función tipoSensor().

```
#Función para incrementar el frameCounter
def increase_frameCounter():
    global contador
    contador = (contador + 1) & 0xFF
```

Figura 5.36: Función increase_frameCounter().

Otra de las funciones utilizadas es la que permite actualizar los parámetros de configuración a partir de la recepción de una trama de configuración recibida desde el nodo *gateway*. En la Figura 5.37 se muestra el código esta función, denominada *actualizar_config()*. En ella se observa cómo se decodifica la trama recibida y se actualizan los parámetros de configuración del nodo final. Para actualizar el identificador del nodo, es necesario que el bit 6 del Byte 0 sea '1'. También se observa el cambio de escala del valor *threshold* recibido desde la trama de configuración y que se almacena en la variable *umbralMagneto*. Este cambio de escala se realiza porque el *byte* que almacena en la trama de configuración el valor *threshold* solo puede estar dentro del rango de valores de 0 a 255, y el umbral del magnetómetro es del orden de decenas de miles. Con el cambio de escala el nuevo rango de valores umbrales es de 0 a 20.000.

La siguiente función desarrollada se encarga de cambiar el modo de funcionamiento según la hora del día, tal y como se puede observar en la Figura 5.38. En esta función, denominada *define_mode()*, se evalúa si se encuentra en la franja correspondiente al modo nocturno, o al modo normal. Para ver si se encuentra en la franja nocturna, se compara la hora actual con el valor del parámetro *NMStart*, que indica la hora de comienzo de la franja nocturna, y con la suma de la hora de inicio de este modo y de la duración, que viene determinada por el parámetro *NMPeriod*. Si la hora actual se encuentra entre estos dos valores, el modo de funcionamiento que corresponde es el nocturno, y si no es así, el modo de funcionamiento es el normal.

En la función *clearframe()* de la Figura 5.39 se muestra cómo se elimina el contenido de los *bytearray*, para poder rellenar los bytes correspondientes a los campos de una nueva trama.

```
#Función para actualizar la configuración actual del nodo por los
#valores recibidos de la trama de configuración
def actualizar_config():
    global version
    global NMStart
    global NMPeriod
    global NMKeepAlive
    global NMSleepTime
    global SleepTime
    global KeepAlive
    global id1
    global threshold
    version = bytearr[2]
    NMStart = bytearr[3]
    NMPeriod = bytearr[4]
    NMKeepAlive = bytearr[5]
    NMSleepTime = bytearr[6]
    SleepTime = bytearr[8]
    KeepAlive = bytearr[9]
    threshold = bytearr[10]
    #Cambio de escala
    #De 255 a 20000-----> 10000 de umbral = 128 en un byte
    umbralMagneto = threshold * 79
    if(bytearr[0] == 0x73):
      id1= bytearr[1]
```

Figura 5.37: Función actualizar_config().

```
#Función para comprobar si se debe cambiar de modo (normal o nocturno)
def define_mode():
    global NM
    #Se captura el estado del RTC.
    m = rtc.now()
    #Si se llega a la hora NMStart se pasa al modo nocturno, y no se
    #cambia de modo hasta que se haya cumplido la duración del modo nocturno.
    if((m[3] >= NMStart) and (m[3] <= NMStart + NMPeriod)):
      NM = 1
    #Cuando termina el período establecido para el modo nocturno,
    #se inicia el modo normal
    if((m[3] > NMStart + NMPeriod) and (m[3] < NMStart)):
      NM = 0
```

Figura 5.38: Función define_mode().

```
#Función para borrar los campos de las tramas
def clearframe():
    global frame
    for i in range(0, 11):
     frame[i] = frame[i] & 0x00
```

Figura 5.39: Función clearframe().

La siguiente función que se desarrolló permite realizar lecturas del ADC del dispositivo LoPy para el cálculo del nivel de la batería. En la Figura 5.40 se adjunta el contenido de la función *ADCloopMeanStdDev()*. En ella se puede ver cómo se realizan las lecturas del ADC mediante el Pin 16 del dispositivo LoPy y con una cierta ponderación. Con las medidas recogidas se realiza una media y sobre ese valor se convierte de bits al valor correspondiente en mV con una simple regla de tres. Teniendo en cuenta que el número máximo en decimal que se puede alcanzar es 4095, y que la tensión máxima puede ser de 3700 mV, el valor medio calculado se divide entre 4095 y se multiplica por la tensión máxima, obteniendo de esta manera los valores expresados en mV. Tras calcular el nivel de batería, se puede comparar su valor con el límite establecido para garantizar un funcionamiento adecuado, y comprobar que este se encuentra por encima de dicho límite. En caso de encontrarse por debajo, se actualiza la variable *batteryStatus* al valor 0x40, mientras que si el nivel de batería es adecuado, se deja el valor de esta variable en 0x00. Posteriormente, en la configuración de las tramas, se hace uso de esta variable para informar del estado de la batería.

En la Figura 5.41, en la Figura 5.42 y en la Figura 5.43 se muestra el código de la función *magneto()*, que permite realizar lecturas del sensor magnetómetro y almacenar los valores leídos para compararlos posteriormente en la función *estadoPlaza()*. Como se puede ver en el código adjunto en dichas figuras, se tiene en cuenta que las primeras lecturas del magnetómetro no son fiables, por lo que se descartan realizando nuevas lecturas. Ya en la tercera lectura se considera que el magnetómetro se encuentra estabilizado y por tanto sus valores se corresponden con el entorno en el que se encuentra el sensor. Una vez que se sale del bucle que repite las medidas, es decir se accede siempre directamente a la condición de funcionamiento “normal”.

Con respecto a las medidas, se puede ver cómo se consulta previamente si hay un dato preparado antes de almacenar su valor, o imprimirlo por pantalla.

```
#Función para realizar lecturas del estado de la batería usando ADC
def ADCloopMeanStdDev():
    global batteryLevel
    global batteryStatus
    #Se calcula el nivel de carga de batería
    adc = machine.ADC(0)
    adcread = adc.channel(attn=ADC.ATTN_11DB, pin='P16')
    samplesADC = [0.0]*numADCreadings; meanADC = 0.0
    i = 0
    #Se realizan 100 medidas, y después se calcula la media
    while (i < numADCreadings):
        adcint = adcread()
        samplesADC[i] = adcint
        meanADC += adcint
        i += 1
    meanADC /= numADCreadings
    #Regla de 3 para devolver valores en mV
    batteryLevel = meanADC*3700/4095
    #Se compara el nivel de batería con la carga mínima preestablecida
    if(batteryLevel < minCharge):
        batteryStatus = 0x40 #Nivel bajo de batería
    else:
        batteryStatus = 0x00 #Nivel adecuado
    print("Mean of ADC readings (0-4095) = %15.13f" % meanADC)
    print("Mean of ADC readings (0-3300 mV) = %15.13f" % batteryLevel)
```

Figura 5.40: ADCloopMeanStdDev().

```
383  #Función para realizar las lecturas del magnetómetro
384  def magneto():
385      global contadorMagneto
386      global FuncionamientoNormal
387      global x
388      global qmc5883l_m
389      #Las primeras lecturas suelen dar valores fuera de rango, por lo que se
390      #descartan. Solo se entra en esta condición en el arranque del nodo
391      if(FuncionamientoNormal == 0):
392          #Bucle al que se entra 3 veces, es decir en 3 lecturas,
393          #los valores medidos ya son estables
394          while(contadorMagneto > 0):
395           #Se configura el modo continuo del magnetómetro
396           qmc5883l_m.setMode(_QMC5883L_Mode_Continuos, _QMC5883L_ODR_200Hz,
397           _QMC5883L_RNG_2G, _QMC5883L_OSR_256)
398           time.sleep_ms(1000)
399           #Se descuenta el contador de entrada al bucle
400           contadorMagneto = contadorMagneto - 1
401           #Si ya se han realizado las 3 lecturas, se activa el flag de
402           #funcionamiento normal
403           if(contadorMagneto == 0):
404             FuncionamientoNormal = 1
```

Figura 5.41: Función magneto().

```
405             #Si hay un dato listo se procede a la lectura
406             if qmc5883l_m.dataReady():
407               x, y, z = qmc5883l_m.readMag()
408               print(x)
409               print(y)
410               print(z)
411               print("---------")
412               qmc5883l_m.softReset()
413               time.sleep(5)
414             else:
415               print("--------- not qmc5883l_m.dataReady")
416               time.sleep_ms(250)
```

Figura 5.42: Función magneto()_2.

```
417       #Cuando el flag de funcionamiento normal ya está activado
418       #se entiende que las medidas ya son estables
419       if(FuncionamientoNormal == 1):
420            qmc5883l_m.setMode(_QMC5883L_Mode_Continuos, _QMC5883L_ODR_200Hz,
421            _QMC5883L_RNG_2G, _QMC5883L_OSR_256)
422            time.sleep_ms(1000)
423            if qmc5883l_m.dataReady():
424              x, y, z = qmc5883l_m.readMag()
425              print(x)
426              print(y)
427              print(z)
428              print("---------")
429              qmc5883l_m.softReset()
430              time.sleep(5)
431            else:
432              print("--------- not qmc5883l_m.dataReady")
433              time.sleep_ms(250)
```

Figura 5.43: Función magneto()_3.

En relación con el procesamiento de las medidas del sensor magnetómetro en la función *magneto()*, la función *estadoPlaza()*, se encarga de evaluar el estado de la plaza de aparcamiento, es decir, determina si la plaza está libre u ocupada. El código de esta función se muestra en la Figura 5.44 y la Figura 5.45. La comprobación del estado de la plaza de aparcamiento se establece a partir de la comparación entre el valor leído en la función *magneto()* y el valor umbral que se encuentra almacenado en la variable *umbralMagneto*. Si el valor leído es inferior al umbral se interpreta que la plaza de aparcamiento está "Ocupado", mientras que si el valor leído es superior al umbral se entiende que la plaza de aparcamiento se encuentra en estado "Libre". Una vez que se determina el estado de la plaza se evalúa si se debe enviar una trama de tipo *Info Frame* con el estado determinado. Esta trama de *Info Frame* se enviará solo en caso de que el estado haya cambiado. De esta manera se evita enviar tramas de información constantemente cuando el estado de la plaza no ha cambiado.

Tras enviar la trama de información se almacena el valor leído por el magnetómetro en la variable auxiliar *xprev*. Este valor se convierte de sistema decimal a sistema hexadecimal para poder enviar su valor en la trama de tipo *Info Frame*. Se necesitan tres bytes para la transmisión de este valor.

```
435  #Función para evaluar el estado de la plaza de aparcamiento
436  def estadoPlaza():
437      global x
438      global xprev
439      global frame
440      global xprevbin
441      global xprevbinGSB
442      global xprevbinMSB
443      global xprevbinLSB
444      global umbralMagneto
445
446      print(x)
447      print(xprev)
448      if(x < umbralMagneto):
449          print('plaza ocupada')
450          if(estadoParking == 0):
451            frame[0]=0x80
452            estadoParking = 1
453      if(x >= umbralMagneto):
454          print('plaza libre')
455          if(estadoParking == 1):
456            frame[0]=0x00
457            estadoParking = 0
```

Figura 5.44: Función estadoPlaza().

```
458      #Se transfiere el valor medido a una variable auxiliar para enviarlo
459      #en la InfoFrame
460      xprev = x
461      #Se pasa el valor de la x a binario(maximo valor 66000-> 17 bits max)
462      xprevbin = int(xprev)
463      xprevbinGSB = (xprevbin >> 16) & 0xFF #Terceros 8 bits
464      xprevbinMSB = (xprevbin >> 8) & 0xFF #Segundos 8 bits
465      xprevbinLSB = (xprevbin) & 0xFF #Primeros 8 bits
466      frame[4] = frame[4] | (xprevbinLSB | 0x00)
467      frame[3] = frame[3] | (xprevbinMSB | 0x00)
468      frame[2] = frame[2] | (xprevbinGSB | 0x00)
469      trama('info')
```

Figura 5.45: Función estadoPlaza()_2.

El procedimiento para determinar el estado de la plaza de aparcamiento utilizando el sensor de proximidad es similar, pero con algunas puntualizaciones que se deben comentar. En la Figura 5.46 y en la Figura 5.47 se muestra el código de la función

proximity(), que es la función que se encarga de determinar el estado de la plaza de aparcamiento y de enviar el valor medido por el sensor cuando se hace uso del sensor de proximidad.

```
352  #Función para sensor de proximidad
353  def proximity():
354      global frame
355      global zx
356      global estadoParking
357      global z
358      global x
359      global umbralProxi
360      #Si se puede realizar la lectura se recoge el valor
361      #de la coordenada Z para enviarlo en la Info Frame
362      if(zx.position_available()):
363         z, x = zx.get_position()
364         print("z: {}, x: {}".format(z,x))
365         frame[2] = frame[2] | (z | 0x00)
```

Figura 5.46: Función proximity().

```
366      #Si el valor de la coord. Z es superior al umbral, la plaza está ocupada
367      if(z >= umbralProxi):
368          print("ocupado")
369          #Si la plaza ya estaba ocupada no se vuelve a enviar un InfoFrame
370          if(estadoParking == 0):
371           frame[0]= frame[0] | 0x80
372           trama('info')
373           estadoParking = 1
374      #Si el valor de la coord. Z es inferior al umbral, la plaza está libre
375      if(z < umbralProxi):
376          print("libre")
377          #Si la plaza ya estaba libre no se vuelve a enviar un InfoFrame
378          if(estadoParking == 1):
379           frame[0]= frame[0] | 0x00
380           trama('info')
381           estadoParking = 0
```

Figura 5.47: Función proximity()_2.

En la función *proximity()* se comprueba inicialmente si hay un objeto reflector que permita al sensor de proximidad realizar medidas, si es así se realiza la medida y se almacena en el Byte 2 de la trama de tipo *Info Frame* el valor medido. Seguidamente, se utiliza ese mismo valor para determinar si la plaza se encuentra ocupada o libre. En este

caso se evalúa si el valor medido es superior o inferior al umbral establecido para el sensor de proximidad (*umbralProxi*). Si el valor medido es superior, se entiende que la plaza se encuentra en estado "Ocupado", mientras que, si el valor medido es inferior al umbral, se interpreta que la plaza está "Libre". Al igual que en la función *estadoPlaza()*, solo se envían tramas de tipo *Info Frame* si el estado recién determinado es diferente al anterior.

En este punto, ya se tienen las funciones que permiten determinar el estado de la plaza y conocer el valor medido por el sensor para los dos sensores estudiados y caracterizados en la plataforma HW/SW propuesta. Se recuerda que según el valor que tenga la variable *TipoSensor*, se utilizará un sensor u otro. Se comentó con anterioridad que con la función *tipoSensor()* se inicializaba la comunicación I^2C con el sensor especificado, pues bien, con la función *lecturaSensor()* se llama a una de las funciones de lectura del sensor. Dependiendo una vez más de la variable *TipoSensor*, se decide si se llama a las funciones asociadas al sensor magnetómetro (*magneto()* y *estadoPlaza()*) o a las del sensor de proximidad (*proximity()*). En la Figura 5.58 se muestra el código de la función *lecturaSensor()*.

```
471  def lecturaSensor(tipo):
472      global zx
473      global qmc5883l_m
474      if(tipo == 1):#Realizar lectura con el sensor de proximidad
475       proximity()
476
477      if(tipo == 0):#Realizar lectura con el magnetómetro
478       magneto()
479       estadoPlaza()
```

Figura 5.48: Función lecturaSensor().

En la siguiente función se codifica el proceso de formación de las tramas. El código de la función *formarTrama()* se adjunta en la Figura 5.49, la Figura 5.50 y la Figura 5.51. Así, en la Figura 5.49 se muestra el código común a todas las tramas con las que trabaja esta función. Estas tramas son: *Info Frame*, *Keep-Alive Frame*, *Start Frame 1* y *Start Frame 2*. Todas las tramas llevan en el Byte 1 el valor del campo *frameCounter* que corresponda, además de contar en el Byte 0 con un bit que indica el estado de carga de la batería,

además del identificador del nodo final, que se encuentra siempre en el Byte 5. El resto de Bytes se rellenan con un valor u otro dependiendo del formato de cada tipo de trama. Por ejemplo, en el Byte 0, los primeros 4 bits se dedican a identificar el tipo de trama, por lo que como se observa en la Figura 5.49, la Figura 5.50 y la Figura 5.51, se realiza una operación 'OR' con el Byte 0 para incluir el tipo de trama, mientras se evita la modificación de otros bits ya asignados.

```
#Función para rellenar las tramas con el contenido apropiado
def formarTrama(tipo):
    global frame

    #Se incrementa el valor del FrameCounter y se guarda en el Byte 1.
    increase_frameCounter()
    frame[1] = contador

    #Se guarda ID del nodo final en el Byte 5.
    frame[5] = id1

    #Consulta del estado de la batería y se almacena el valor 0x00(nivel adecuado de carga)
    #o 0x40(nivel bajo de carga) en el Byte 0.
    ADCloopMeanStdDev()
    frame[0] = frame[0] | batteryStatus
```

Figura 5.49: Función formarTrama().

```
#Según el tipo de trama que se pase por parámetro en la función se rellena
#un tipo de trama u otro
if(tipo == 'info'):
    frame[0] = frame[0] | 0x00 # Identificador del tipo de trama: Info Frame

if(tipo == 'keep'):
    frame[0] = frame[0] | 0x01 # Identificador del tipo de trama: Keep-Alive Frame
    total = chrono.read() #Se realiza una lectura del tiempo transcurrido
    #desde que se inició el funcionamiento del nodo
    if(total >= 3600): #Si el tiempo transcurrido supera los 3600 segundos,
    #se guardan el número de horas transcurridas en la variable totalh
     totalh = total/3600
    else:   #Si no, se deja este valor a 0.
     totalh = 0
    if(total >=60): #Si el tiempo transcurrido supera los 60 segundos,
    #se guardan los minutos transcurridos en la variable totalm
     totalm = total/60
    else:   #Si no, se deja este valor a 0.
     totalm = 0
    #Se guardan las horas y los minutos transcurridos en los Bytes 2 y 3 respectivamente.
    frame[2] = int(totalh)
    frame[3] = int(totalm)
```

Figura 5.50: Función formarTrama()_2.

```
if(tipo == 'start1'):
    frame[0] = frame[0] | 0x04  # Identificador del tipo de trama: Start Frame 1
    #Se convierte el nivel de batería a un número binario
    binBatteryLevel = int(batteryLevel)
    #Se recogen los bits más significativos de ese número binario
    binBatteryLevelMSB = (binBatteryLevel >> 8) & 0xFF
    #Se recogen los bits menos significativos de ese número binario
    binBatteryLevelLSB = (binBatteryLevel) & 0xFF
    #Se guardan los bits más significativos y los menos significativos en los
    #Bytes 9 y 10 respectivamente.
    frame[9] =  binBatteryLevelMSB
    frame[10] = binBatteryLevelLSB

if(tipo == 'start2'):
    frame[0] = frame[0] | 0x05 # Identificador del tipo de trama: Start Frame 2
    #En la Start Frame 2 se informa d ela configuración inicial del nodo.
    frame[2] = version
    frame[3] = NMStart
    frame[4] = NMPeriod
    frame[6] = NMKeepAlive
    frame[7] = NMSleepTime
    frame[8] = SleepTime
    frame[9] = KeepAlive
```

Figura 5.51: Función formarTrama()_3.

En la Figura 5.50 se puede observar el proceso realizado para obtener el tiempo transcurrido desde que se inició el funcionamiento del nodo final. En primer lugar, se realiza una lectura del estado del T*imer*, y posteriormente se evalúa el número de horas y minutos transcurridos según el valor de la lectura. Por último, se transfieren los valores a los Bytes 2 y 3.

En la Figura 5.51 se muestra el contenido principal de las tramas de inicio. En la trama de tipo *Start Frame 1* se puede ver cómo se convierte el nivel de batería a su correspondiente valor binario y se divide este valor en dos Bytes, con los bits más significativos en el Byte 9, y los menos significativos en el Byte 10. Por otra parte, en la trama de tipo *Start Frame 2*, se almacenan en los Bytes todos los parámetros de configuración. De esta manera se informa al usuario de la configuración inicial que utiliza el nodo final.

La función mostrada en la Figura 5.52 permite recoger varias funciones y ejecutarlas una detrás de otra, para poder dejar el código más limpio y ordenado. Las funciones que se incluyen en *trama()* son: *formarTrama()*, *send()* y *clearFrame()*. Así, con esta función se

empaqueta la trama, se envía, y se elimina el contenido del *bytearray* para poder volver a utilizarlo.

Una vez se han presentado todas las funciones que constituyen el código correspondiente a la implementación de la funcionalidad del nodo final en la plataforma HW/SW desarrollada en este TFG, se puede empezar a comentar el código en líneas más generales, y siguiendo la idea del diagrama de flujo presentado.

```
#Función que engloba otras funciones con el objetivo de simplificar el loop infinito
def trama(tipo):
    global frame
    if(tipo == 'info'):
     formarTrama('info')
    if(tipo == 'keep'):
     formarTrama('keep')
    if(tipo == 'daily'):
     formarTrama('daily')
    if(tipo == 'start1'):
     formarTrama('start1')
    if(tipo == 'start2'):
     formarTrama('start2')
    #Se visualiza la trama por pantalla
    print("- RECEIVED data = {}".format(binascii.hexlify(frame)))
    #Se envía la trama
    s.send(frame)
    #Se borran los campos de la trama(Se limpia el bytearray)
    clearframe()
```

Figura 5.52: Función trama().

Así, el funcionamiento del nodo final comienza con el envío de las tramas de inicio y con la inicialización de la comunicación I^2C con el sensor correspondiente. Posteriormente, se entra en un bucle infinito dónde se evalúan diferentes situaciones y se actúa en consecuencia. El inicio del funcionamiento del nodo final se muestra en la Figura 5.53. En la Figura 5.53 se observa también que, dentro del bucle infinito, se define el modo de funcionamiento, y además se desbloquea el *socket* para poder transmitir.

En la Figura 5.54 se muestra la lectura del *timer* que permite comparar el tiempo de *Keep-Alive* establecido, con el tiempo transcurrido. Como se comentó anteriormente, dependiendo del modo de funcionamiento se evalúa la condición con un tiempo de *Keep-*

Alive correspondiente al modo nocturno o al modo normal. En caso de cumplirse el tiempo establecido, se reinicia el *timer* y se envía una trama de tipo *Keep-Alive*.

```
562  ########Inicio del funcionamiento del nodo#########################
563  #En el inicio del nodo se envían las dos tramas de inicio
564  trama('start1')
565  trama('start2')
566
567  #Se inicializan los sensores
568  tipoSensor(TipoSensor)
569  #loop infinito
570  while 1:
571   #Se define en qué modo se está actualmente
572   define_mode()
573
574   # Se desbloquea el socket para poder transmitir
575   s.setblocking(True)
576
577   #Se lee el estado del sensor y se envia la infoframe oportuna
578   lecturaSensor(TipoSensor)
```

Figura 5.53: Arranque del funcionamiento del nodo.

```
#Se realiza una acptura del timer 'k'
lap = k.read()

#Cada 'KeepAlive' segundos se manda una trama de keep alive en modo normal
if(lap >= KeepAlive and NM == 0):
 #se reinicia la cuenta y se envía la trama de Keep-Alive
 k.reset()
 trama('keep')
#Cada 'KeepAlive' segundos se manda una trama de keep alive en modo nocturno
if(lap >= NMKeepAlive and NM == 1):
 #Se reincia la cuenta y se envía la trama de Keep-Alive
 k.reset()
 trama('keep')
```

Figura 5.54: Comprobación de períodos Keep-Alive.

Tras cubrir todos los casos de envío de trama, se contempla la posibilidad de recepción de una trama de configuración, para lo cual se bloquea el *socket*, y se recibe la trama. La trama recibida se almacena en un *bytearray* y se filtra para ver si se corresponde con una

trama de configuración, realizándose también un filtrado por si la trama recibida no es para el nodo en cuestión, a través del identificador. Si el nodo es el receptor esperado, se actualiza la configuración del nodo final. El proceso explicado de muestra en la Figura 5.55.

```
#Recepción de la trama de configuración
#Se bloquea el socket para poder recibir
s.setblocking(False)
#Se realiza la recepción
data = s.recv(64)
#Se almacenan los datos en un bytearray
bytearr = bytearray(data)
#Se imprime el bytearray recibido en la pantalla
print("- RECEIVED data from TTN = {}".format(binascii.hexlify(bytearr)))

#Si el bytearray tiene contenido se analiza la trama recibida
if(len(data) > 0):
 #Se comprueba que la trama recibida es de configuración
 if((bytearr[0] == 0x73) or (bytearr[0] == 0x33)): #Código-filtro = 0x33 en el Byte 0.
 #0x33--> Campo asignarID ='no', 0x73--> Campo asignarID ='si'
 #Se analiza si el mensaje es para este nodo: variable ID1
  if(bytearr[7] == id1): #Si la trama recibida es para este nodo,
  #se actualizan los parámetros configuradores del nodo
   actualizar_config()
```

Figura 5.55: Recepción de trama de configuración.

Finalmente, como último paso del código desarrollado, se comprueba el modo de funcionamiento actual, y en consecuencia se realiza una espera por un tiempo mayor (en caso del modo nocturno) o menor (en caso del modo normal). Durante este tiempo, el nodo no ejecuta ninguna acción, y por tanto el consumo de batería es muy bajo. En la Figura 5.56 se adjunta el código relacionado.

```
#Según el modo de funcionamiento el tiempo de espera(dormido)
# del nodo será un valor u otro
if(NM == 1): #Modo nocturno
 time.sleep(NMSleepTime)
else: #Modo normal
 time.sleep(SleepTime)
```

Figura 5.56: Tiempo de Sleep del nodo final.

Al finalizar el tiempo de espera establecido en cada caso, se vuelve al comienzo del bucle infinito y se vuelve a ejecutar el mismo código continuamente.

Capítulo 6. Nanogateway

El siguiente elemento de la solución, que se encuentra en el nivel inmediatamente superior al del nodo final según la arquitectura de la plataforma HW/SW planteada, es el *gateway*. El nodo *gateway* es el encargado de gestionar la comunicación con los nodos finales y con la plataforma *The Things Network*. La comunicación con los nodos finales se realiza mediante la tecnología LoRa/LoRaWAN, mientras que la comunicación con la plataforma TTN se establece vía TCP/IP. Por lo tanto, a diferencia de los nodos finales, el dispositivo que actúa como *gateway* se ubica en un punto estratégico para que pueda tener acceso a comunicación TCP/IP, que en este caso concreto se materializa en el uso de tecnología *WiFi*. Es decir, debe estar ubicado a una distancia adecuada de los nodos finales, que como se indicó en capítulos anteriores, en el caso de un par de kilómetros la comunicación es totalmente viable, aun variando el *Spreading Factor* entre todos los valores posibles. Además, debe estar a unos pocos metros de un *router* con acceso a Internet con el fin de poder establecer la comunicación TCP/IP con la plataforma TTN. En definitiva, el *gateway* debe ubicarse en un punto donde ambos tipos de comunicación puedan darse sin ningún problema.

El dispositivo utilizado para implementar la funcionalidad del *gateway* es el mismo que se usa como nodo final, el dispositivo LoPy de Pycom. Se recuerda que este dispositivo cuenta con tecnología *WiFi*, *Bluetooth* y LoRa, por lo que sus características se adaptan perfectamente a las necesidades de un *gateway* en una solución como la planteada en el

presente TFG. Se utilizará la tecnología *WiFi* para la conexión a Internet, y la tecnología LoRa para la comunicación con el nodo final.

No obstante, el *gateway* de esta solución es del tipo denominado como *nanogateway* o *gateway* monocanal.

Se denomina *nanogateway* a un *gateway* de un único canal o "*single-channel gateway*". Los *gateways* de canal único solo pueden recibir datos por un canal, y con un único *Spreading Factor* al mismo tiempo, mientras que los *gateways* "completos" pueden recibir por 8 canales diferentes y con 6 valores posibles de *Spreading Factor* simultáneamente [53]. Algunos *gateways* de canal único pueden "saltar" entre frecuencias y valores de S*preading Factor* para simular un *gateway* "completo", pero aun así se mantienen por debajo del 2% de la capacidad real de un *gateway* "completo".

Es importante resaltar que utilidades como *Adaptative Data Rate* no se pueden utilizar con estos *nanogateways*, ya que requieren de múltiples canales para realizar las adaptaciones a las necesidades reales de la red [53].

La principal ventaja que se tiene en los *nanogateways* es su coste, que resulta mucho más asequible, económicamente hablando, que los *gateways* "completos". En resumen, conociendo las limitaciones a nivel de canales y de *Spreading Factor* en las recepciones simultáneas, se puede hacer uso de este tipo de *gateway* sin generar ningún conflicto en la aplicación concreta propuesta en el presente TFG.

Como se ha comentado anteriormente, se utilizará el dispositivo LoPy para la implementación de la funcionalidad del *nanogateway*, que está certificado por LoRaWAN, aunque como nodo, no como *gateway*. Por lo tanto, teniendo claro que no es la solución ideal para una gran red de dispositivos, se utiliza el dispositivo LoPy como *nanogateway* con el fin de realizar pruebas experimentales, y especialmente para pequeñas aplicaciones en las que se controlan las frecuencias utilizadas por los nodos, como por ejemplo en los nodos LoPy. Así, el *nanogateway* queda como se muestra en la Figura 6.1.

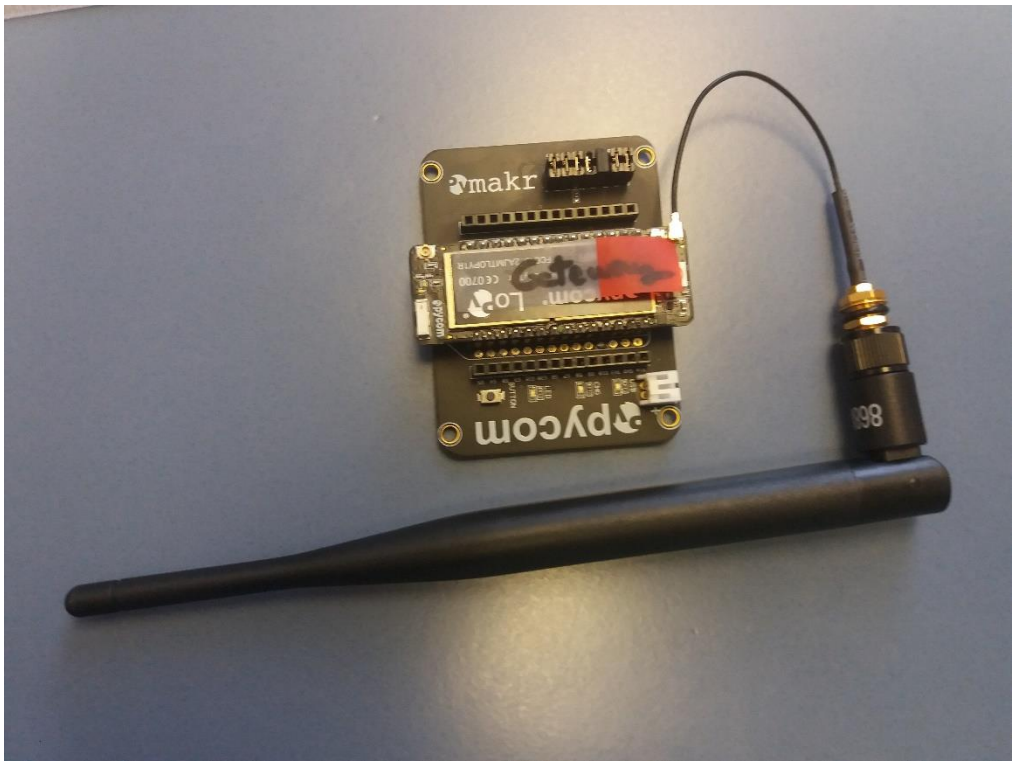


Figura 6.1: Dispositivo LoPy utilizado como nanogateway.

De este modo, se justifica el uso de este dispositivo de Pycom como *nanogateway* por el bajo coste que represente en comparación a un *gateway* "completo", y porque la aplicación propuesta se encuentra dentro de los límites de un *gateway* de canal único.

Pycom, en su documentación [54] facilita el código de un *nanogateway* de referencia, que incorpora todas las funciones requeridas por este, y en el que, en principio, únicamente hay que configurar los parámetros asociados a la comunicación *WiFi* (*Service Set Identifier* (SSID) y contraseña). El código actualizado se puede consultar en *Github* [55].

En el caso concreto del *nanogateway* utilizado en el presente TFG, consta de 3 archivos en *MicroPython*:

- `Main.py`: Inicialización del funcionamiento del *nanogateway*.
- `Nanogateway.py`: Código que codifica la funcionalidad del *nanogateway*.

- `Config.py`: Destinado a la configuración de las opciones del *nanogateway*.

6.1 Archivo: `nanogateway.py`

Se comienza por el análisis del fichero *nanogateway.py*, pues los otros dos archivos tienen dependencias con este fichero. Las librerías utilizadas por este archivo son las que se muestran en la Figura 6.2.

```
""" LoPy LoRaWAN Nano Gateway. Can be used for both EU868 and US915. """

import errno
import machine
import ubinascii
import binascii
import ujson
import uos
import usocket
import utime
import _thread
from micropython import const
from network import LoRa
from network import WLAN
from machine import Timer
```

Figura 6.2: Librerías utilizadas en nanogateway.py.

Seguidamente, en la Figura 6.3 se adjuntan las constantes utilizadas en el código. Como se puede observar en la Figura 6.3, se incluyen constantes asociadas a diferentes mensajes de error, mostrándose su uso en las funciones definidas.

En esta parte dedicada a la declaración de constantes, también se incluyen estructuras de metadatos, que como se ve en la Figura 6.4, se inicializan a un valor por defecto ('0' en la mayoría de los casos). La estructura de la Figura 6.4, STAT_PK, se rellena con campos y valores relacionados con la estadística actual del *nanogateway*, por lo que incluye campos como el tiempo transcurrido desde que se inició su funcionamiento, su ubicación, etc. También se inicializa la estructura RX_PK, cuyos campos y valores se relacionan directamente con el estado de la recepción de paquetes. Tanto la estructura RX_PK como

TX_ACK_PK se adjuntan en la Figura 6.5. La estructura TX_ACK_PK contiene información sobre el estado de la transmisión realizada. Su único campo es el de error, por lo que está orientado hacia el aviso de errores en la transmisión. Posteriormente, en las funciones se hará uso de estas estructuras y se rellenará su contenido con los valores oportunos.

```
PROTOCOL_VERSION = const(2)

PUSH_DATA = const(0)
PUSH_ACK = const(1)
PULL_DATA = const(2)
PULL_ACK = const(4)
PULL_RESP = const(3)

TX_ERR_NONE = 'NONE'
TX_ERR_TOO_LATE = 'TOO_LATE'
TX_ERR_TOO_EARLY = 'TOO_EARLY'
TX_ERR_COLLISION_PACKET = 'COLLISION_PACKET'
TX_ERR_COLLISION_BEACON = 'COLLISION_BEACON'
TX_ERR_TX_FREQ = 'TX_FREQ'
TX_ERR_TX_POWER = 'TX_POWER'
TX_ERR_GPS_UNLOCKED = 'GPS_UNLOCKED'

UDP_THREAD_CYCLE_MS = const(10)
```

Figura 6.3: Variables utilizadas en nanogateway.py.

En la Figura 6.6 se crea la clase NanoGateway, que será utilizada en el archivo *main.py*. Dentro de esta clase se definen todas las funciones que permiten que el dispositivo LoPy se comporte como *nanogateway*. El resto del código de este archivo, por tanto, son funciones que se describirán a continuación. En la Figura 6.6 se muestra un fragmento de la función *__init__()*.

```
STAT_PK = {
    'stat': {
        'time': '',
        'lati': 0,
        'long': 0,
        'alti': 0,
        'rxnb': 0,
        'rxok': 0,
        'rxfw': 0,
        'ackr': 100.0,
        'dwnb': 0,
        'txnb': 0
    }
}
```

Figura 6.4: Estructura STAT_PK.

```
RX_PK = {
    'rxpk': [{
        'time': '',
        'tmst': 0,
        'chan': 0,
        'rfch': 0,
        'freq': 0,
        'stat': 1,
        'modu': 'LORA',
        'datr': '',
        'codr': '4/5',
        'rssi': 0,
        'lsnr': 0,
        'size': 0,
        'data': ''
    }]
}

TX_ACK_PK = {
    'txpk_ack': {
        'error': ''
    }
}
```

Figura 6.5: Estructuras RX_PK y TX_ACK_PK.

```
class NanoGateway:
    """
    Nano gateway class, set up by default for use with TTN, but can be configured
    for any other network supporting the Semtech Packet Forwarder.
    Only required configuration is wifi_ssid and wifi_password which are used for
    connecting to the Internet.
    """

    def __init__(self, id, frequency, datarate, ssid, password, server, port, ntp_server='pool.ntp.org', ntp_period=3600):
        self.id = id
        self.server = server
        self.port = port

        self.frequency = frequency
        self.datarate = datarate

        self.ssid = ssid
        self.password = password

        self.ntp_server = ntp_server
        self.ntp_period = ntp_period
```

Figura 6.6: Creación de la clase NanoGateway y de la función __init__().

La configuración del *gateway* se realiza a partir de los parámetros que se pasan a la función *__init__()*. Estos parámetros, como se verá más adelante, se obtienen del archivo *config.py*. En esta función simplemente se recogen los valores de los parámetros de configuración y se asignan al objeto creado de la clase NanoGateway.

Además de volcar estos parámetros de configuración, también se inician valores como el *Spreading Factor* o el ancho de banda, entre otros, tal y como se muestra en la Figura 6.7. Estos dos valores, por ejemplo, se recogen a partir del parámetro de configuración "*datarate*", que como se verá en el archivo *config.py*, incluye en un *String*, tanto el *Spreading Factor* como el ancho de banda. Para extraer la información del *String* se hace uso de dos funciones, que se presentan en la Figura 6.8, denominadas *_dr_to_sf()* y *_dr_to_bw()*. Además, en la misma Figura 6.8 se tiene la conversión inversa a partir de la función *_sf_bw_to_dr().*

El resto de las inicializaciones están relacionadas con las comunicaciones *WiFi* y LoRa, pero en esta función no se realiza ninguna acción determinante en su funcionamiento, ya que todas las inicializaciones se mantienen en '*None*'. Por último, se asocia el RTC creado con el RTC propio del objeto NanoGateway.

```
97          self.ntp_period = ntp_period
98
99          self.server_ip = None
100
101         self.rxnb = 0
102         self.rxok = 0
103         self.rxfw = 0
104         self.dwnb = 0
105         self.txnb = 0
106
107         self.sf = self._dr_to_sf(self.datarate)
108         self.bw = self._dr_to_bw(self.datarate)
109
110         self.stat_alarm = None
111         self.pull_alarm = None
112         self.uplink_alarm = None
113
114         self.wlan = None
115         self.sock = None
116         self.udp_stop = False
117         self.udp_lock = _thread.allocate_lock()
118
119         self.lora = None
120         self.lora_sock = None
121
122         self.rtc = machine.RTC()
```

Figura 6.7: Función __init__()_2.

La siguiente función que se analiza es *start()*. No obstante, con antelación se debe comentar brevemente la existencia de la función *_log()* que se encarga de crear mensajes, darles formato, e imprimirlos por pantalla. El código se adjunta en la Figura 6.9.

El código de la Figura 6.10 y la Figura 6.11 muestra la declaración de la función *start()* y su contenido. Lo primero que se hace es imprimir por pantalla un mensaje de inicio con el identificador del *gateway*, haciendo uso de la función *_log()*. Seguidamente, se configura la conexión *WiFi*, para lo cual, se utiliza la función *_connect_to_WiFi()*, cuyo código se adjunta en la Figura 6.12. Una vez configurado y conectado a la red *WiFi*, se sincroniza el RTC con el servidor NTP (servidor horario).

```
def _dr_to_sf(self, dr):
    sf = dr[2:4]
    if sf[1] not in '0123456789':
        sf = sf[:1]
    return int(sf)

def _dr_to_bw(self, dr):
    bw = dr[-5:]
    if bw == 'BW125':
        return LoRa.BW_125KHZ
    elif bw == 'BW250':
        return LoRa.BW_250KHZ
    else:
        return LoRa.BW_500KHZ

def _sf_bw_to_dr(self, sf, bw):
    dr = 'SF' + str(sf)
    if bw == LoRa.BW_125KHZ:
        return dr + 'BW125'
    elif bw == LoRa.BW_250KHZ:
        return dr + 'BW250'
    else:
        return dr + 'BW500'
```

Figura 6.8: Conversiones Datarate-SF/BW.

```
    def _log(self, message, *args):
        """
        Outputs a log message to stdout.
        """

        print('[{:>10.3f}] {}'.format(
            utime.ticks_ms() / 1000,
            str(message).format(*args)
))
```

Figura 6.9: Función _log().

El siguiente paso consiste en acceder al servidor IP, crear un socket UDP *(User Datagram Protocol)*, y dejarlo listo para su uso. Posteriormente, se envía inmediatamente el primer paquete STAT_PK. Para rellenar la estructura de este paquete se utiliza la función

_make_stat_packet(), mientras que para enviar el paquete se hace uso de la función *_push_data()*. Ambas se definen en la Figura 6.13 y la Figura 6.14, respectivamente.

También se crean las interrupciones, haciendo uso de las funciones de alarma del *Timer*. Así, cuando se cumple el tiempo establecido (60 segundos para la primera alarma y 25 para la segunda), y se llama a la función *_push_data()* en el primer caso, y a la función *_pull_data()* en el segundo. El código de la función *_pull_data()* se adjunta en la Figura 6.14.

En la parte final de la función, se comienza un *thread* de recepción UDP haciendo uso de la función *_udp_thread(),* que se detalla en la Figura 6.15, la Figura 6.16 y la Figura 6.17, y se configura LoRa en el modo LoRa (no LoRaWAN), y con los parámetros de configuración indicados, el *socket* asociado.

Por último, si se produce un evento de tipo *TX_PACKET_EVENT* o *RX_PACKET_EVENT,* se llama a la función *_lora_cb()*, cuyo código se adjunta en la Figura 6.18, y que se encarga de dar respuesta al evento acontecido.

Se debe comentar, además, que en una de las llamadas a la función *_log()* se hace uso de la función *_freq_to_float()*, que como su nombre indica convierte el parámetro que se le pasa (una frecuencia) en una variable de tipo *float*, con el fin de poder trabajar con mayor precisión. En la Figura 6.19 se adjunta el código correspondiente a la implementación de dicha función.

En la función *_connect_to_WiFi()* de la Figura 6.12 se ve cómo se introducen el SSID y la contraseña para poder establecer la conexión *WiFi*. A continuación, se comprueba si se está conectado, y cuando lo está, se imprime un mensaje por pantalla indicando esta situación.

En la función *_make_stat_packet()* de la Figura 6.13 se rellena la estructura STAT_PK con los valores actualizados. Se observa cómo para el primer campo, se requiere hacer una lectura del RTC. El resto de los valores se consultan directamente al objeto NanoGateway creado.

```
124     def start(self):
125         """
126         Starts the LoRaWAN nano gateway.
127         """
128 
129         self._log('Starting LoRaWAN nano gateway with id: {}', self.id)
130 
131         # setup WiFi as a station and connect
132         self.wlan = WLAN(mode=WLAN.STA)
133         self._connect_to_wifi()
134 
135         # get a time sync
136         self._log('Syncing time with {} ...', self.ntp_server)
137         self.rtc.ntp_sync(self.ntp_server, update_period=self.ntp_period)
138         while not self.rtc.synced():
139             utime.sleep_ms(50)
140         self._log("RTC NTP sync complete")
141 
142         # get the server IP and create an UDP socket
143         self.server_ip = usocket.getaddrinfo(self.server, self.port)[0][-1]
144         self._log('Opening UDP socket to {} ({}) port {}...', self.server, self.server_ip[0], self.server_ip[1])
145         self.sock = usocket.socket(usocket.AF_INET, usocket.SOCK_DGRAM, usocket.IPPROTO_UDP)
146         self.sock.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
147         self.sock.setblocking(False)
```

Figura 6.10: Función start().

```
150         self._push_data(self._make_stat_packet())
151         # create the alarms
152         self.stat_alarm = Timer.Alarm(handler=lambda t: self._push_data(self._make_stat_packet()), s=60, periodic=True)
153         self.pull_alarm = Timer.Alarm(handler=lambda u: self._pull_data(), s=25, periodic=True)
154 
155         # start the UDP receive thread
156         self.udp_stop = False
157         _thread.start_new_thread(self._udp_thread, ())
158 
159         # initialize the LoRa radio in LORA mode
160         self._log('Setting up the LoRa radio at {:.1f} Mhz using {}', self._freq_to_float(self.frequency), self.datarate)
161         self.lora = LoRa(
162             mode=LoRa.LORA,
163             frequency=self.frequency,
164             bandwidth=self.bw,
165             sf=self.sf,
166             preamble=8,
167             coding_rate=LoRa.CODING_4_5,
168             tx_iq=True
169         )
170         # create a raw LoRa socket
171         self.lora_sock = usocket.socket(usocket.AF_LORA, usocket.SOCK_RAW)
172         self.lora_sock.setblocking(False)
173         self.lora_tx_done = False
174 
175         self.lora.callback(trigger=(LoRa.RX_PACKET_EVENT | LoRa.TX_PACKET_EVENT), handler=self._lora_cb)
176         self._log('LoRaWAN nano gateway online')
```

Figura 6.11: Función start()_2.

```
def _connect_to_wifi(self):
    self.wlan.connect(self.ssid, auth=(None, self.password))
    while not self.wlan.isconnected():
        utime.sleep_ms(50)
    self._log('WiFi connected to: {}', self.ssid)
```

Figura 6.12: Función _connect_to_WiFi().

En la Figura 6.14 se encuentra el código de las funciones *_push_data()* y *_pull_data()*. La primera función se encarga de enviar los datos que se le pasan por parámetro con el *socket* UDP hacia el servidor IP. Mientras que a la segunda función no se le pasa ningún parámetro, y, por tanto, no envía más que el identificador del *gateway*.

```
def _make_stat_packet(self):
    now = self.rtc.now()
    STAT_PK["stat"]["time"] = "%d-%02d-%02d %02d:%02d:%02d GMT" % (now[0], now[1], now[2], now[3], now[4], now[5])
    STAT_PK["stat"]["rxnb"] = self.rxnb
    STAT_PK["stat"]["rxok"] = self.rxok
    STAT_PK["stat"]["rxfw"] = self.rxfw
    STAT_PK["stat"]["dwnb"] = self.dwnb
    STAT_PK["stat"]["txnb"] = self.txnb
    return ujson.dumps(STAT_PK)
```

Figura 6.13: Función _make_stat_packet().

```
def _push_data(self, data):
    token = uos.urandom(2)
    packet = bytes([PROTOCOL_VERSION]) + token + bytes([PUSH_DATA]) + ubinascii.unhexlify(self.id) + data
    with self.udp_lock:
        try:
            self.sock.sendto(packet, self.server_ip)
            self._log('push_data packet: {}'.format(binascii.hexlify(packet)))
        except Exception as ex:
            self._log('Failed to push uplink packet to server: {}', ex)

def _pull_data(self):
    token = uos.urandom(2)
    packet = bytes([PROTOCOL_VERSION]) + token + bytes([PULL_DATA]) + ubinascii.unhexlify(self.id)
    with self.udp_lock:
        try:
            self.sock.sendto(packet, self.server_ip)
            self._log('pull_data packet: {}'.format(binascii.hexlify(packet)))
        except Exception as ex:
            self._log('Failed to pull downlink packets from server: {}', ex)
```

Figura 6.14: Función _push_data() y _pull_data().

En la Figura 6.18 se adjunta el código de la función *_lora_cb()*. En esta función, según el evento que se produzca, se lleva a cabo un tipo de acciones u otras. Así, si el evento es de tipo RX_PACKET_EVENT, se incrementan variables asociadas al número de paquetes recibidos, se recibe el paquete y se consulta el estado de la recepción del paquete. Además, los datos recibidos se pasan como parámetro a la función *_make_node_packet()*, que como se muestra en la Figura 6.20, rellena los datos de la estructura de tipo RX_PK. Con la estructura formada, se llama a la función *_push_data()* para que la envíe al servidor IP.

En caso de que el evento sea de tipo TX_PACKET_EVENT, se incrementa la variable que contabiliza el número de transmisiones realizadas, y se inicia la comunicación LoRa con los parámetros configurados.

En la función *_udp_thread()* de la Figura 6.15, la Figura 6.16 y la Figura 6.17, se reciben los datos del servidor y se realizan diferentes acciones sobre ellos. Lo primero que se hace es analizar el tipo de dato recibido, de manera que si es un *Acknowledgement* (ACK) se imprime por pantalla un mensaje que indica que los datos han sido enviados y recibidos correctamente, mientras que si es de tipo PULL_RESP, significa que se ha recibido otro tipo de dato que no es de confirmación.

```
360     def _udp_thread(self):
361         """
362         UDP thread, reads data from the server and handles it.
363         """
364
365         while not self.udp_stop:
366             try:
367                 data, src = self.sock.recvfrom(1024)
368                 _token = data[1:3]
369                 _type = data[3]
370                 if _type == PUSH_ACK:
371                     self._log("Push ack")
372                 elif _type == PULL_ACK:
373                     self._log("Pull ack")
```

Figura 6.15: Función _udp_thread().

```
372                 elif _type == PULL_ACK:
373                     self._log("Pull ack")
374                 elif _type == PULL_RESP:
375                     self.dwnb += 1
376                     ack_error = TX_ERR_NONE
377                     tx_pk = ujson.loads(data[4:])
378                     tmst = tx_pk["txpk"]["tmst"]
379                     t_us = tmst - utime.ticks_us() - 12500
380                     if t_us < 0:
381                         t_us += 0xFFFFFFFF
382                     if t_us < 20000000:
383                         self.uplink_alarm = Timer.Alarm(
384                             handler=lambda x: self._send_down_link(
385                                 ubinascii.a2b_base64(tx_pk["txpk"]["data"]),
386                                 tx_pk["txpk"]["tmst"] - 50, tx_pk["txpk"]["datr"],
387                                 int(tx_pk["txpk"]["freq"] * 1000000)
388                             ),
389                             us=t_us
390                         )
391                     else:
392                         ack_error = TX_ERR_TOO_LATE
393                         self._log('Downlink timestamp error!, t_us: {}', t_us)
394                     self._ack_pull_rsp(_token, ack_error)
395                     self._log("Pull rsp")
```

Figura 6.16: Función _udp_thread()_2.

Ese dato se recoge y se almacena temporalmente, creándose una interrupción para enviar un mensaje *downlink* hacia el nodo final. Dentro de la creación de esa interrupción se hace uso de la función *_send_down_link(),* que se especifica en la Figura 6.21, como acción a ejecutar cuando se produzca la interrupción.

Finalmente, si alguno de los procesos descritos falla, se muestran y envían mensajes de error. El último paso consiste en cerrar el *socket* y finalizar el *thread* UDP.

Como se puede observar en la Figura 6.17, los mensajes de error se envían haciendo uso de la función *_ack_pull_rsp()*, que se muestra en la Figura 6.22.

Por otra parte, en la Figura 6.19 se muestra la función *_freq_to_float(),* que se encarga de transformar la frecuencia que se pasa por parámetro, en una variable de tipo *float* con un mecanismo que evita imprecisiones.

En la función *_make_node_packet()* de la Figura 6.20 se rellena la estructura RX_PK con todos los valores relacionados con la recepción de un paquete.

```
391                         else:
392                             ack_error = TX_ERR_TOO_LATE
393                             self._log('Downlink timestamp error!, t_us: {}', t_us)
394                         self._ack_pull_rsp(_token, ack_error)
395                         self._log("Pull rsp")
396                 except usocket.timeout:
397                     pass
398                 except OSError as ex:
399                     if ex.errno != errno.EAGAIN:
400                         self._log('UDP recv OSError Exception: {}', ex)
401                 except Exception as ex:
402                     self._log('UDP recv Exception: {}', ex)
403 
404                 # wait before trying to receive again
405                 utime.sleep_ms(UDP_THREAD_CYCLE_MS)
406 
407             # we are to close the socket
408             self.sock.close()
409             self.udp_stop = False
410             self._log('UDP thread stopped')
```

Figura 6.17: Función _udp_thread()_3.

```
def _lora_cb(self, lora):
    """
    LoRa radio events callback handler.
    """

    events = lora.events()
    if events & LoRa.RX_PACKET_EVENT:
        self._log('*** LoRa.RX_PACKET_EVENT')
        self.rxnb += 1
        self.rxok += 1
        rx_data = self.lora_sock.recv(256)
        stats = lora.stats()
        packet = self._make_node_packet(rx_data, self.rtc.now(), stats.rx_timestamp,
        stats.sfrx, self.bw, stats.rssi, stats.snr)
        self._push_data(packet)
        self._log('Received packet: {}', packet)
        self._log('Received packet rx_data: {}'.format(binascii.hexlify(rx_data)))
        self.rxfw += 1
    if events & LoRa.TX_PACKET_EVENT:
        self._log('*** LoRa.TX_PACKET_EVENT')
        self.txnb += 1
        lora.init(mode=LoRa.LORA,frequency=self.frequency,bandwidth=self.bw,
            sf=self.sf,preamble=8,coding_rate=LoRa.CODING_4_5,tx_iq=True)
```

Figura 6.18: Función _lora_cb().

```
def _freq_to_float(self, frequency):
    """
    MicroPython has some inprecision when doing large float division.
    To counter this, this method first does integer division until we
    reach the decimal breaking point. This doesn't completely elimate
    the issue in all cases, but it does help for a number of commonly
    used frequencies.
    """

    divider = 6
    while divider > 0 and frequency % 10 == 0:
        frequency = frequency // 10
        divider -= 1
    if divider > 0:
        frequency = frequency / (10 ** divider)
    return frequency
```

Figura 6.19: Función _freq_to_float().

```
def _make_node_packet(self, rx_data, rx_time, tmst, sf, bw, rssi, snr):
    RX_PK["rxpk"][0]["time"] = "%d-%02d-%02dT%02d:%02d:%02d.%dZ" % (rx_time[0],
    rx_time[1], rx_time[2], rx_time[3], rx_time[4], rx_time[5], rx_time[6])
    RX_PK["rxpk"][0]["tmst"] = tmst
    RX_PK["rxpk"][0]["freq"] = self._freq_to_float(self.frequency)
    RX_PK["rxpk"][0]["datr"] = self._sf_bw_to_dr(sf, bw)
    RX_PK["rxpk"][0]["rssi"] = rssi
    RX_PK["rxpk"][0]["lsnr"] = snr
    RX_PK["rxpk"][0]["data"] = ubinascii.b2a_base64(rx_data)[:-1]
    RX_PK["rxpk"][0]["size"] = len(rx_data)
    return ujson.dumps(RX_PK)
```

Figura 6.20: Función _make_node_packet().

En la Figura 6.21 se muestra la función *_send_down_link(),* que se encarga de enviar datos al nodo final. Para llevar a cabo esta tarea, en primer lugar, inicializa la comunicación LoRa con los parámetros configurados, y posteriormente envía los datos.

En la función *_ack_pull_rsp(),* mostrada en la Figura 6.22, se rellena la estructura TX_ACK_PK con el error que se ha producido, y se envía al servidor IP.

```
def _send_down_link(self, data, tmst, datarate, frequency):
    """
    Transmits a downlink message over LoRa.
    """

    self.lora.init(
        mode=LoRa.LORA,
        frequency=frequency,
        bandwidth=self._dr_to_bw(datarate),
        sf=self._dr_to_sf(datarate),
        preamble=8,
        coding_rate=LoRa.CODING_4_5,
        tx_iq=True
        )
    while utime.ticks_us() < tmst:
        pass
    self.lora_sock.send(data)
    self._log(
        '!!! Sent downlink packet scheduled on {:.3f}, at {:.1f} Mhz using {}: {}',
        tmst / 1000000,
        self._freq_to_float(frequency),
        datarate,
        data
    )
```

Figura 6.21: Función _send_down_link().

```
def _ack_pull_rsp(self, token, error):
    TX_ACK_PK["txpk_ack"]["error"] = error
    resp = ujson.dumps(TX_ACK_PK)
    packet = bytes([PROTOCOL_VERSION]) + token + bytes([PULL_ACK]) + ubinascii.unhexlify(self.id) + resp
    with self.udp_lock:
        try:
            self.sock.sendto(packet, self.server_ip)
        except Exception as ex:
            self._log('PULL RSP ACK exception: {}', ex)
```

Figura 6.22: Función _ack_pull_rsp().

Tras haber descrito dos de las funciones principales, *__init__()* y *start()*, a continuación, se explica una tercera función principal, denominada *stop()*. En la Figura 6.23 se adjunta el código correspondiente a esta función.

En la función *stop()* de la Figura 6.23 se detienen todos los procesos que se están ejecutando. Se indica la parada de LoRa, con la opción de *Sleep*, se cancelan todas las interrupciones, se detiene el sincronismo del RTC, se indica al *thread* UDP que se detenga, y se deshabilita la conexión *WiFi*.

Con esta función se finaliza el análisis del archivo *nanogateway.py*

```
def stop(self):
    """
    Stops the LoRaWAN nano gateway.
    """

    self._log('Stopping...')

    # send the LoRa radio to sleep
    self.lora.callback(trigger=None, handler=None)
    self.lora.power_mode(LoRa.SLEEP)

    # stop the NTP sync
    self.rtc.ntp_sync(None)

    # cancel all the alarms
    self.stat_alarm.cancel()
    self.pull_alarm.cancel()

    # signal the UDP thread to stop
    self.udp_stop = True
    while self.udp_stop:
        utime.sleep_ms(50)

    # disable WLAN
    self.wlan.disconnect()
    self.wlan.deinit()
```

Figura 6.23: Función stop().

6.2 Archivo: `config.py`

En este apartado se procede a comentar el archivo *config.py*, cuyo código se muestra en la Figura 6.24. En la Figura 6.24 se pueden observar los parámetros de configuración que se pueden editar en función de las particularidades de cada *nanogateway*. Es fundamental modificar las variables WiFi_SSID y WiFi_PASS que vienen por defecto para poder establecer la conexión *WiFi* con la red que se desee. Por otra parte, también es importante introducir en la variable GATEWAY_ID el identificador del *gateway* registrado en la plataforma *The Things Network*. Además, los datos se envían desde el *gateway* hacia

el servidor, por lo que se debe indicar a qué servidor se enviarán los datos. La dirección del servidor, que se recoge en la variable SERVER, pertenece a TTN.

```
""" LoPy LoRaWAN Nano Gateway configuration options """

GATEWAY_ID = '70B3D54999D4C26B'

SERVER = 'router.eu.thethings.network'
PORT = 1700

NTP = "pool.ntp.org"
NTP_PERIOD_S = 3600

WIFI_SSID = 'ULPGC'
WIFI_PASS = ''

LORA_FREQUENCY = 868100000
LORA_GW_DR = "SF7BW125" # DR_5
LORA_NODE_DR = 5
```

Figura 6.24: config.py.

6.3 Archivo: `main.py`

Por último, en la Figura 6.25 se adjunta el código del archivo *main.py*, en el que se crea un objeto de la clase NanoGateway y se le asignan los parámetros de configuración establecidos en el archivo *config.py*. Seguidamente, se llama a la función *start()* para iniciar así el funcionamiento del *nanogateway*.

```
""" LoPy LoRaWAN Nano Gateway example usage """

import config
from nanogateway import NanoGateway

if __name__ == '__main__':
    nanogw = NanoGateway(
        id=config.GATEWAY_ID,
        frequency=config.LORA_FREQUENCY,
        datarate=config.LORA_GW_DR,
        ssid=config.WIFI_SSID,
        password=config.WIFI_PASS,
        server=config.SERVER,
        port=config.PORT,
        ntp_server=config.NTP,
        ntp_period=config.NTP_PERIOD_S
        )

    nanogw.start()
    nanogw._log('You may now press ENTER to enter the REPL')
input()
```

Figura 6.25: main.py.

Capítulo 7. The Things Network - Integración: HTTP

En el presente TFG, se ha elegido TTN como plataforma para desarrollar la aplicación que permitirá gestionar la información recibida/enviada hacia los nodos finales. En esta plataforma se cuenta con un foro dedicado exclusivamente a IoT, código abierto y consulta de otros desarrollos ya completados, así como la posibilidad de ubicar geográficamente un *gateway*, tutoriales, integraciones y una consola. Si se consulta su portal web [56] se visualiza una interfaz como la que se muestra en la Figura 7.1.

Para poder trabajar en esta plataforma se debe crear en primer lugar, una cuenta de usuario. Con la cuenta creada, se puede acceder a todas las funcionalidades de la plataforma, siendo las más utilizadas en el presente TFG la consola, para el desarrollo de la solución, y el foro, para la resolución de diferentes dudas e inconvenientes que han ido surgiendo en el desarrollo de la aplicación.

Si se inicia la consola, aparece la opción de acceder a los *gateways* registrados o a las aplicaciones, tal y como se muestra en la Figura 7.2. El primer paso, consiste en registrar el *gateway*, por lo que se selecciona la opción GATEWAYS. Una vez dentro de esta sección, se da la opción de registrar un nuevo *gateway*.

Figura 7.1: Portal web The Things Network.

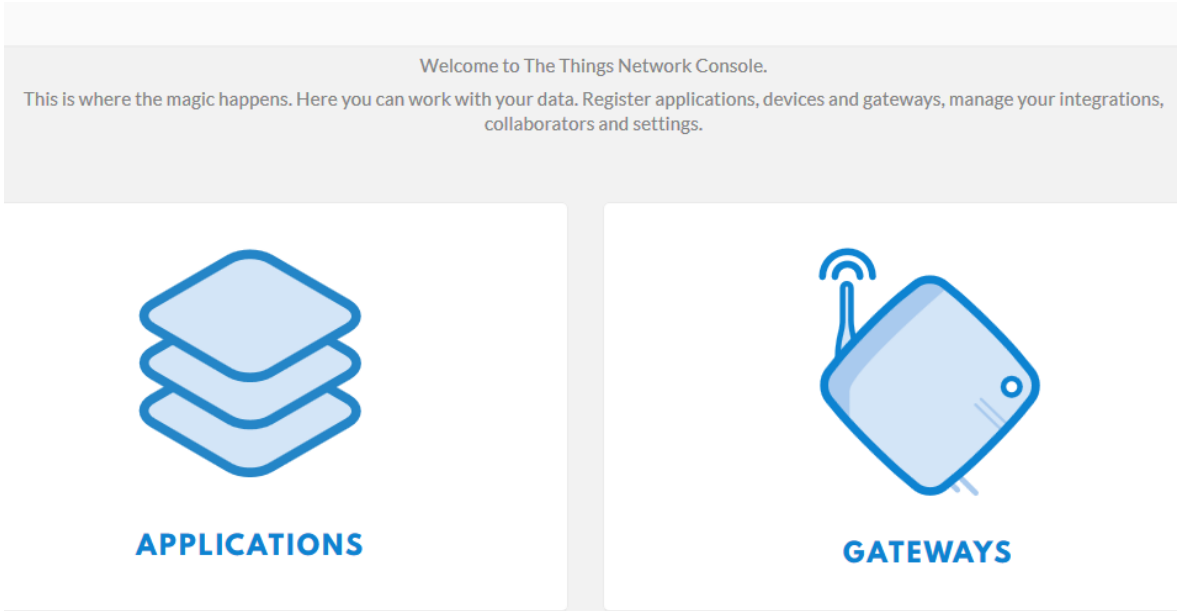


Figura 7.2: Consola de TTN.

En el registro del *gateway* se debe introducir, en primer lugar, el ID del *gateway*, configurado en el archivo *config.py* del *nanogateway*. También se debe seleccionar el plan de frecuencia adecuado, que para el caso de esta aplicación debe ser la frecuencia europea de 868 MHz. También es conveniente señalar si la antena estará ubicada dentro de un habitáculo (*indoor*) o en el exterior (*outdoor*). Además, se puede indicar la ubicación física del *gateway* a partir de un mapa. Con el proceso de registro completado

se pueden consultar las características del *gateway* en el espacio "*Gateway Overview*", que presenta la disposición que se muestra en la Figura 7.3.

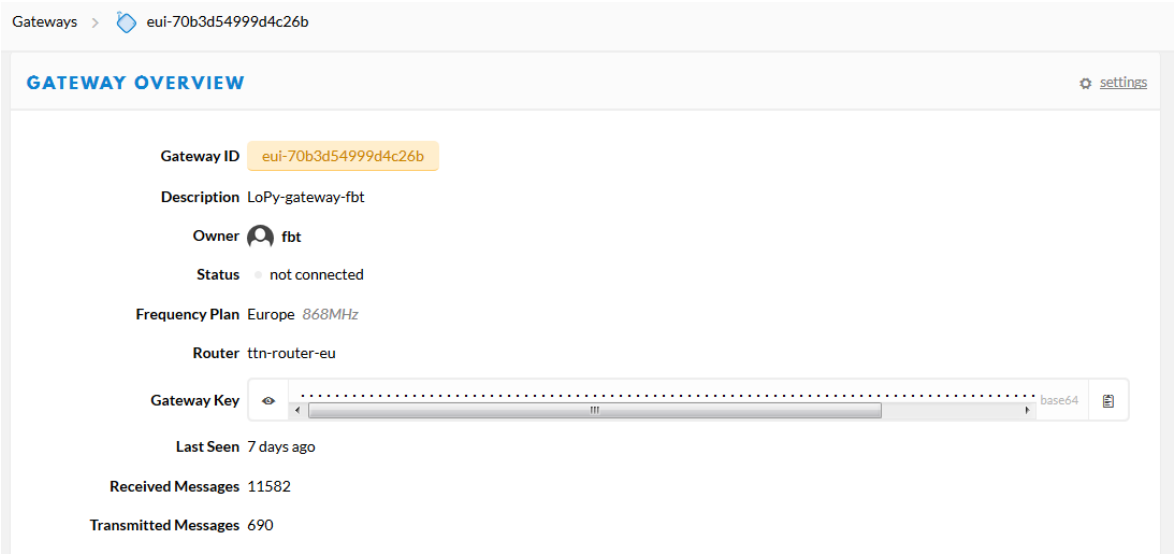


Figura 7.3: Gateway Overview.

Teniendo el *gateway* registrado, ya se puede proceder a la creación de una aplicación en la plataforma TTN.

Los pasos que seguir son similares a los del caso del *gateway*. Así, en primer lugar, se debe registrar la aplicación. En este registro se requiere de un ID de la aplicación, la selección del gestor, que para el caso del presente TFG es el europeo, y el EUI asociado a la aplicación, que autogenerará la propia plataforma TTN. Introduciendo los datos y pulsando en registrar, finaliza el proceso de creación de la aplicación.

Con la aplicación creada se puede acceder al campo denominado "*Application Overview*", que se muestra en la Figura 7.4 y en la Figura 7.5. En ellas se describen las principales características de la aplicación, como su ID, su EUI (que será necesario utilizar en el nodo final), su clave de acceso, los colaboradores que pueden editar la aplicación, y los dispositivos asociados en la aplicación.

Si se accede a los dispositivos, se pueden visualizar aquellos que se tengan registrados, además de disponer de la posibilidad de registrar nuevos dispositivos. El registro de un dispositivo es similar al del *gateway* y al de la aplicación.

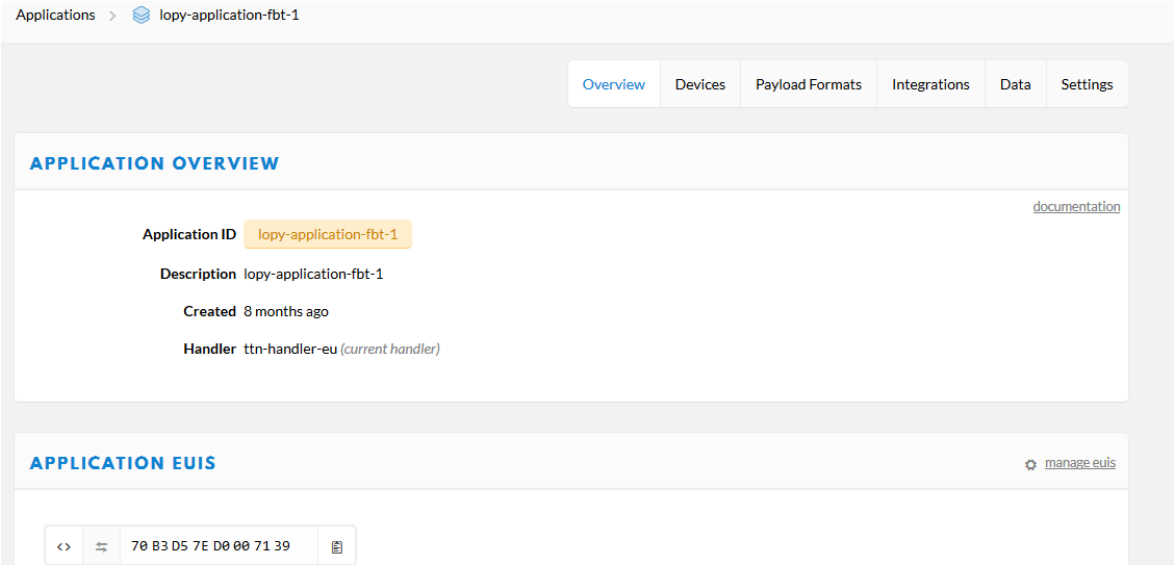


Figura 7.4: Application Overview.

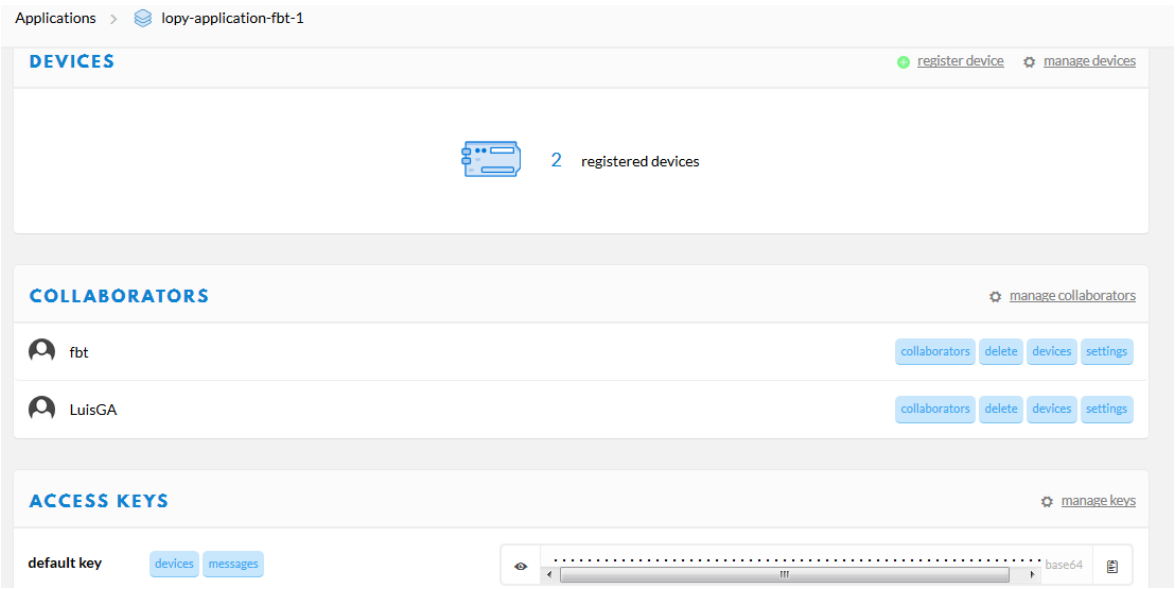


Figura 7.5: Application Overview_2.

Así, en el registro de un dispositivo se requiere la introducción de un ID, el EUI del dispositivo (que debe coincidir con el *dev_EUI* del nodo final) y el EUI de la aplicación. El ID del dispositivo se introduce manualmente por el usuario y el EUI de la aplicación se genera automáticamente en la creación de ésta. Por otra parte, el EUI del dispositivo se obtiene ejecutando la siguiente sentencia en el dispositivo que se desea registrar.

```
binascii.hexlify(network.LoRa().mac())
```

Con el registro del dispositivo se genera una clave específica para el dispositivo, denominada *app_key*, que debe incluirse en el código del nodo final. Si se consultan las características de uno de los dispositivos registrados, se encuentra una interfaz como la mostrada en la Figura 7.6 y en la Figura 7.7. En estas figuras se puede ver también el tipo de activación LoRaWAN que se utiliza para el dispositivo, que para el caso del presente TFG será OTAA. También se generan automáticamente los campos *Device Adress*, *Network Session Key* y *App Session Key*.

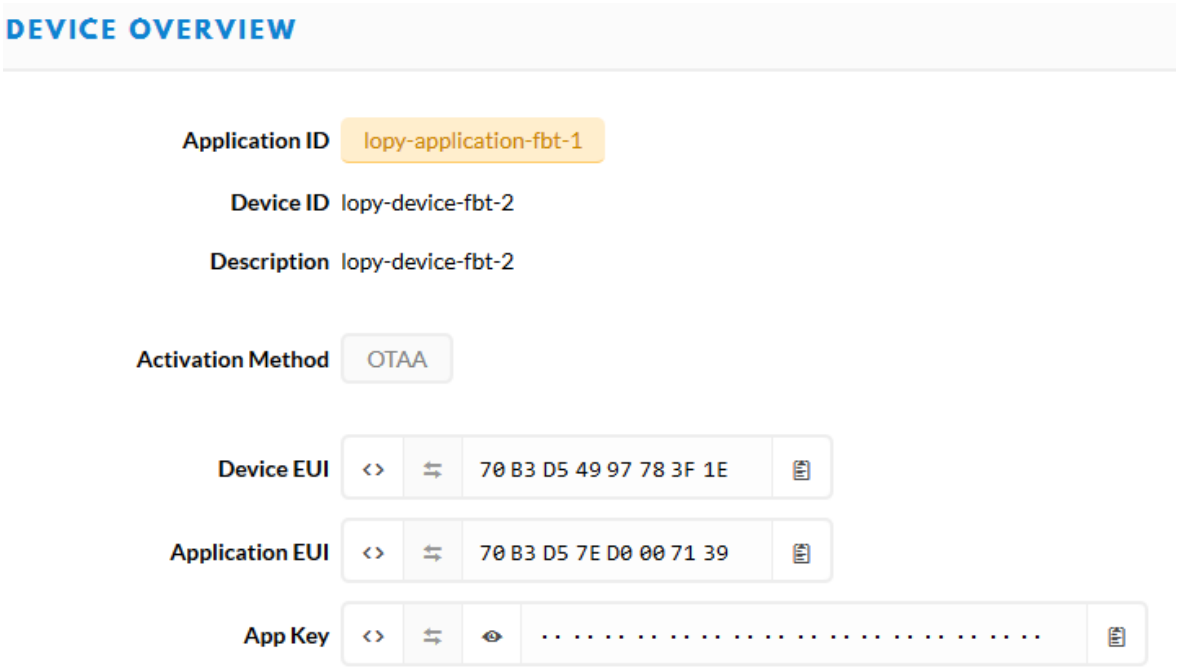


Figura 7.6: Device Overview.

Además, en esta misma pestaña de la consola se da la opción de realizar una transmisión *Downlink* hacia el nodo final. Esta opción se presenta como se muestra en la Figura 7.8.

Por otra parte, en la Figura 7.9 se adjunta la funcionalidad asociada a la simulación de un mensaje de tipo *Uplink* hacia la aplicación. Esta utilidad resulta muy interesante cuando se necesita probar el *Decoder* desarrollado en las *Payload Functions*, que se presentará a continuación.

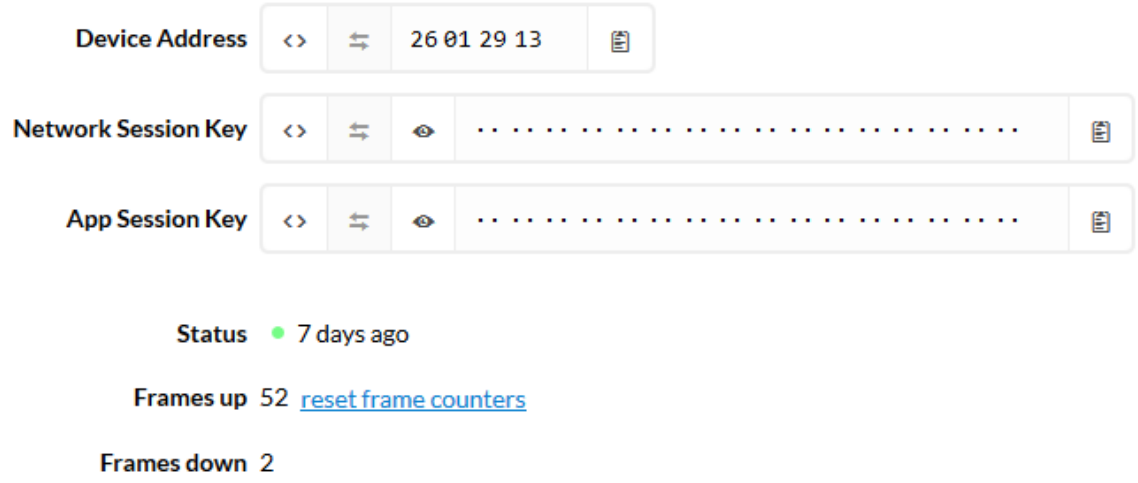


Figura 7.7: Device Overview_2.

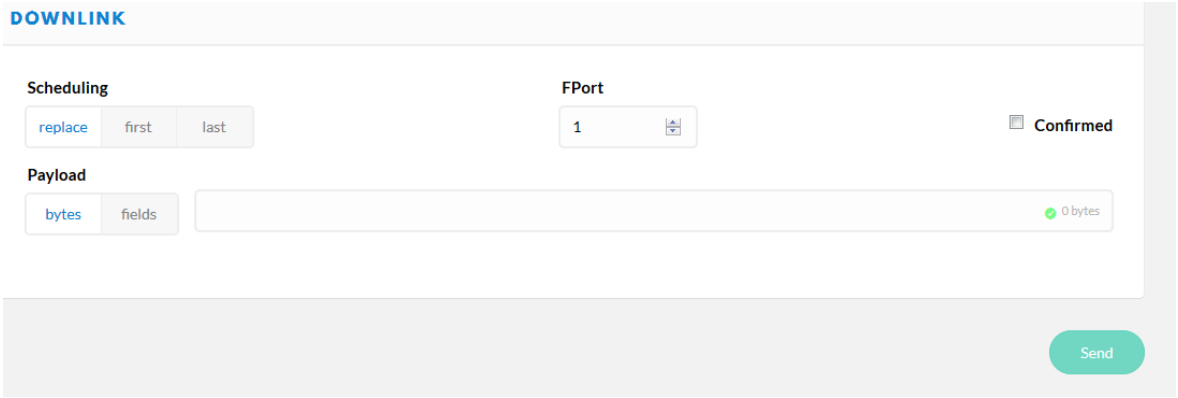


Figura 7.8: Downlink.

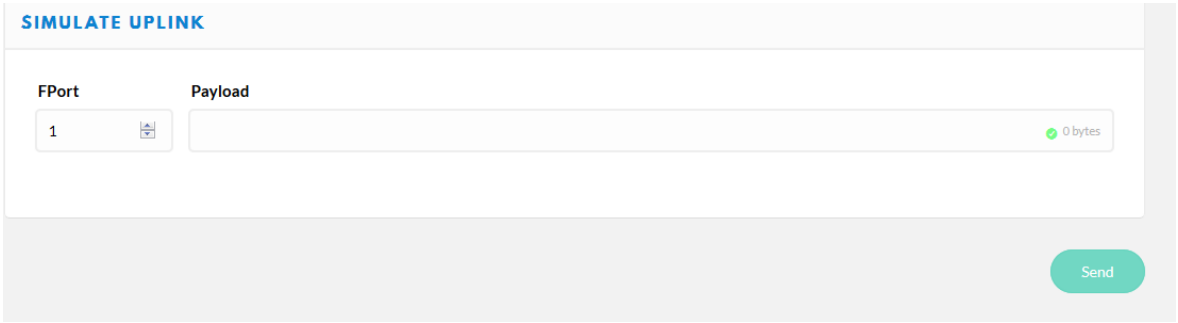


Figura 7.9: Simulate Uplink.

7.1 Payload Functions

Para evitar el trato directo con bytes y poder visualizar el significado de las informaciones transmitidas, se desarrollan las denominadas *Payload Functions*. Estas funciones se incluyen en la pestaña *Payload Formats*, dentro de la aplicación creada en la plataforma TTN, ya que son específicas para cada aplicación.

Las funciones desarrolladas para la aplicación del presente TFG son el decodificador o *decoder*, que se utiliza para convertir los bytes en los valores asociados a determinados campos (o *fields*), y el codificador o *encoder*, que se utiliza para convertir los valores de cada campo en los bytes correspondientes. De esta manera, el *decoder* se utilizará cuando se reciba una trama desde el nodo final, y el *encoder* se utilizará cuando se envíe una trama de configuración desde la integración, o desde la aplicación *Java* desarrollada.

La función *Decoder* permite extraer a partir de la información de cada campo mostrar los bytes recibidos por la aplicación desde el nodo final, a través del *nanogateway*. Hay campos asignados directamente a ciertos bits o bytes que no dependen del tipo de trama recibida, por lo que estos son los primeros en extraerse, como se muestra en la Figura 7.10.

El resto de los campos deben rellenarse en función del tipo de trama que se haya recibido. Para ello se hace uso de la estructura *switch*, como se muestra también en la Figura 7.10, con el fin de poder determinar qué tipo de trama es la que se está recibiendo. Como se explicó en el capítulo correspondiente a la implementación de la funcionalidad del nodo final, las tramas incluyen un identificador de tipo de trama en el Byte 0, y de ahí es donde se extrae la información que determina el tipo de trama. A continuación, se debe evaluar para cada trama si hay que actualizar campos o no. Por ejemplo, para el tipo de trama *Info Frame*, se debe comprobar un bit del Byte 0 para ver si la plaza de aparcamiento está libre u ocupada. Según el valor de este bit, se actualiza el campo *ParkingSlotStatus* con el valor “Libre” u “Ocupado”. También se reciben los valores de las medidas realizadas por los sensores en los Bytes 2, 3 y 4. El Byte 2 contiene o el valor medido por el sensor de proximidad o el valor más significativo, dependiendo del sensor que esté utilizando el nodo final para realizar las medidas. Como el valor medido

por el sensor magnetómetro se almacena en tres Bytes, tras su recepción se reconstruye el valor original de la medida en la variable *xmagneto*.

En las tramas de tipo *Keep-Alive*, la información que se tiene que extraer de los Bytes es el tiempo transcurrido desde que se inició el funcionamiento del nodo final, que se encuentran separados en horas y minutos. Por otra parte, en las tramas de tipo *Start Frame 1*, el campo recuperado a partir de los Bytes 9 y 10 se corresponde con el nivel de la batería, tal y como se muestra en la Figura 7.11.

```
function Decoder(bytes, port) {

// Decode an uplink message from a buffer
// (array) of bytes to an object of fields.

 var batteryState = (bytes[0] & 0x40);
 var tipoTrama = (bytes[0] & 0x0F);
 var FrameCounter = bytes[1];
 var IDnodo = bytes[5];
 var tipo;
 var ParkingSlotStatus;
 var error;

 switch (tipoTrama){

  case (0x00):
   tipo = 'Info';

   if (bytes[0] && 0x80){
    ParkingSlotStatus = 'Ocupado';
   }else{
    ParkingSlotStatus = 'Libre';
   }

   var zproxi = bytes[2];

   var xmagneto1 = bytes[4];
   var xmagneto2 = bytes[3];
   var xmagneto3 = bytes[2];
   var xmagneto = (xmagneto3*65536) + (xmagneto2*256) + (xmagneto1);

   break;

  case (0x01):
   tipo = 'KeepAlive';
   var currentHours = bytes[2];
   var currentMinutes = bytes[3];
   break;
```

Figura 7.10: Función Decoder.

En La Figura 7.11 se puede ver que para las tramas de tipo *Start Frame 2* se extraen múltiples valores de los bytes, ya que prácticamente cada uno de ellos contiene el valor de un campo.

```
40   case (0x04):
41    tipo = 'Start1';
42    var batteryLevel = (bytes[9] << 8) | bytes[10];
43    break;
44
45   case (0x05):
46    tipo = 'Start2';
47    var version = bytes[2];
48    var NMStart = bytes[3];
49    var NMPeriod = bytes[4];
50    var NMKeepAlive = bytes[6];
51    var NMSleepTime = bytes[7];
52    var SleepTime = bytes[8];
53    var KeepAlive = bytes[9];
54    break;
55
56   default:
57
58    error = 'Tipo de trama errónea';
59
60  }
```

Figura 7.11: Función Decoder_2.

```
62  return{
63   tipo: tipo,
64   batteryState: batteryState,
65   FrameCounter: FrameCounter,
66   IDnodo: IDnodo,
67   ParkingSlotStatus: ParkingSlotStatus,
68   currentHours: currentHours,
69   currentMinutes: currentMinutes,
70   batteryLevel: batteryLevel,
71   version: version,
72   error: error,
73   NMStart: NMStart,
74   NMPeriod: NMPeriod,
75   NMKeepAlive: NMKeepAlive,
76   NMSleepTime: NMSleepTime,
77   SleepTime: SleepTime,
78   KeepAlive: KeepAlive,
79   zproxi: zproxi,
80   xmagneto: xmagneto
81   }
82 }
```

Figura 7.12: Función Decoder_3.

En caso de que la trama recibida no se corresponda con ninguna de las enunciadas anteriormente, no se extrae ningún valor más.

Por último, y como se observa en la Figura 7.12, se devuelven las variables en las que se han almacenado los valores extraídos de los bytes recibidos.

Con respecto a la función *Encoder*, en ella se realiza la conversión de los valores de cada campo a bytes, con el fin de poder enviar una trama de configuración al nodo final, y que este pueda interpretarla correctamente.

Para ello se crea un vector de 11 bytes que será rellenado con los valores de los campos que se tengan en la trama de configuración. Como se puede ver en la Figura 7.13, hay valores que se pueden introducir directamente en los bytes correspondientes, mientras que otros dependen de si un campo contiene un "si" o un "no", como es el caso del campo "*asignarID*".

```
1  function Encoder(object, port) {
2
3   // Encode downlink messages sent as
4   // object to an array or buffer of bytes.
5
6   var bytes = [00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00];
7
8   if (object.asignarid === "si"){
9     bytes[0] = 0x73;
10  }else if (object.asignarid === "no"){
11    bytes[0] = 0x33;
12  }
13
14  bytes[1] = object.nodeid;
15  bytes[2] = object.firmwareversion;
16  bytes[3] = object.nmstart;
17  bytes[4] = object.nmperiod;
18  bytes[5] = object.nmsleeptime;
19  bytes[6] = object.nmkeepalive;
20  bytes[7] = object.id2config;
21  bytes[8] = object.sleeptime;
22  bytes[9] = object.keepalive;
23  bytes[10] = object.threshold;
24
25 return bytes;
26
27 }
```

Figura 7.13: Función Encoder.

Una vez que se han rellenado todos los bytes de la trama de configuración, la función devuelve el *array* de *bytes* con su contenido actualizado.

Para validar inicialmente el funcionamiento de la función *Encoder*, se puede enviar una sentencia *JavaScript Object Notation* (JSON) como la que se puede recibir, tanto desde la integración, como desde la aplicación *Java*. Para esta prueba se utiliza la utilidad *Downlink* que se comentó anteriormente. Una sentencia JSON válida podría ser la que se muestra en la Figura 7.14.


```
{ "asignarid": "no", "broadcast": "si", "id2config":2, "nodeid":2, "firmwareversion":1, "nmstart":45,
"nmperiod":30, "nmsleeptime":15, "nmkeepalive":25, "sleeptime":2, "keepalive":5, "threhold":4000}
```


Figura 7.14: Sentencia JSON.

En esta sentencia se pueden observar todos los campos con sus respectivos valores, que se pasarán a los bytes en la función E*ncoder*.

Una vez realizada la configuración, se concluye el proceso de preparación de la plataforma TTN para la interacción descendente directa con el *nanogateway*, e indirecta con el nodo final. Así, la comunicación desde la plataforma TTN con los elementos de niveles inferiores de la arquitectura presentada está establecida, pero en el otro sentido, en el sentido ascendente, todavía se debe interactuar con la integración y con la aplicación *Java*.

7.2 Integración HTTP

En este apartado se analiza la interacción con la integración, ya que está muy ligada a la plataforma TTN. Se ha realizado la integración con el fin de mostrar en una interfaz la información recibida desde el nodo final sin necesidad de tener acceso a la cuenta de TTN creada en la plataforma. Es decir, a partir de la integración HTTP desarrollada, cualquier usuario que posea la dirección web de la integración puede visualizar la información que envía el nodo final.

Si se va a la pestaña *Integrations* dentro de la aplicación creada en TTN, se visualiza una interfaz como la que se muestra en la Figura 7.15.

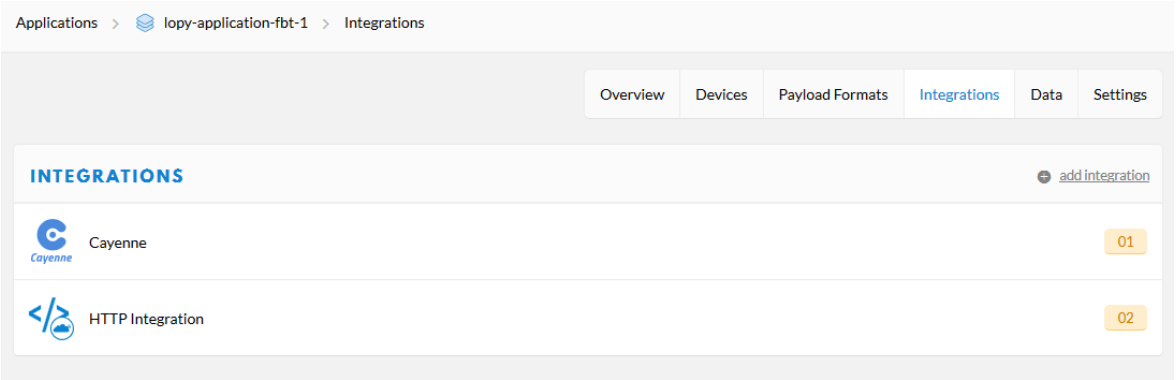


Figura 7.15: Integración en TTN.

Al crear la integración HTTP se pide una *Uniform Resource Locator* (URL). Esta URL se puede generar en el enlace [57] http://requestbin.fullcontact.com/, cuya interfaz se muestra en la Figura 7.16.

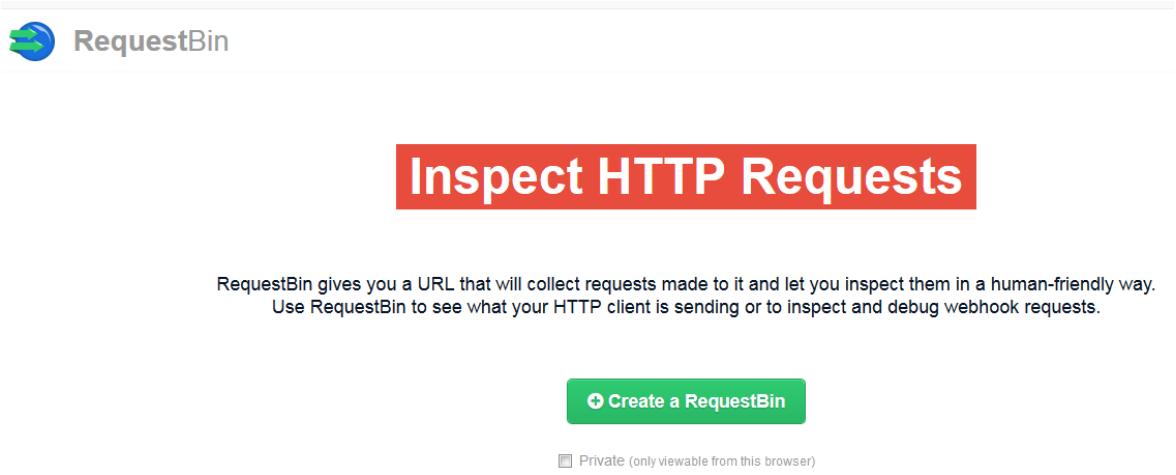


Figura 7.16: RequestBin.

La URL generada se introduce en la configuración de la integración en la plataforma TTN y se selecciona el método POST HTTP. Con esta configuración es suficiente para poder visualizar las tramas recibidas en la URL generada, tal y como se puede observar en la Figura 7.17. Así, en esta figura se muestran la cabecera del POST HTTP y el *Raw_Body*. En la zona *Raw_Body* es donde se pueden visualizar los campos, con sus respectivos valores. Es decir, se muestran los datos enviados desde el nodo final en el POST de la URL generada.

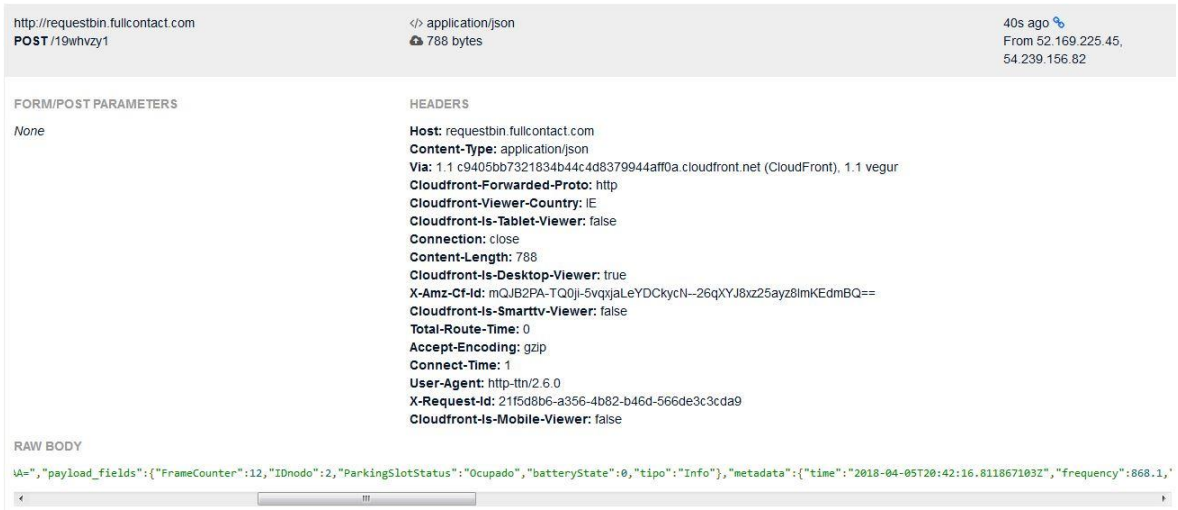


Figura 7.17: Visualización de trama en la Integración HTTP.

Para realizar una transmisión *Downlink* es necesaria la instalación de la herramienta *curl* [58]. Esta herramienta es un *software* consistente en una biblioteca y un intérprete de comandos orientado a la transferencia de archivos. Haciendo esto e introduciendo desde CMD u otra ventana de comandos, la sentencia mostrada en la Figura 7.18, se consigue enviar un *Downlink* a TTN. Como se puede ver en la sentencia, hay una URL que es la que se extrae del último elemento del *Raw_Body*. Para editar la información enviada se debe modificar el valor del campo "*payload_raw*".

```
➜  ~ curl -X POST --data '{"dev_id":"node","payload_raw":"AQE="}' https://integrations.thethingsnetwork.or
g/ttn-eu/api/v2/down/howto-videos/test\?key\=ttn-account-v2.EYXJx_RkdpD3hZZS3_gwXPX84OC1vQACtdnIxmQ0rsY
```

Figura 7.18: Transmisión downlink asociada a la integración HTTP.

Capítulo 8. Aplicación Java

En los apartados anteriores se ha presentado el procedimiento seguido para establecer la comunicación entre el nodo final y el servidor, además de la visualización de los datos asociados a la aplicación desarrollada. En este capítulo se pretende explicar el desarrollo de la aplicación *Java* diseñada para crear una interfaz más visual e intuitiva en la que se muestre la información de estado del nodo, así como la posibilidad de envío de mensajes *Downlink*.

8.1 Creación y configuración del archivo *App.java*

Para la creación de la aplicación y su integración en TTN se deben seguir unos pasos previos que permitirán crear el proyecto *Java* incorporando todas las utilidades necesarias para su programación. A continuación, se describe el proceso previo que se debe realizar para poder empezar a programar la aplicación *Java*.

El objetivo principal de la aplicación desarrollada es comunicarse con la plataforma *The Things Network* para intercambiar información, ya sea en vía ascendente o descendente. Para crear una aplicación que se conecte a TTN, se siguen los pasos que se indican en la documentación de la propia plataforma, y que se puede consultar en la referencia [59].

El primer paso consiste en asegurarse de que se dispone de *Java* instalado en el PC en el que se vaya a desarrollar la aplicación, y que esta versión sea al menos la versión 8. Si no

se tiene instalado *Java*, o la versión es inferior a la indicada, se puede descargar o actualizar desde la página oficial de *Java* [60].

El siguiente paso consiste en instalar *Apache Maven* [61], un gestor de proyectos *software*. Para ello se puede consultar la guía de la referencia [62] en caso de tener un sistema operativo Windows, como en el caso particular del presente TFG.

El procedimiento para instalar *Maven* se inicia con la descarga del archivo comprimido del *software* desde el enlace de la referencia [63]. Seguidamente se descomprime en la carpeta en la que se desea instalar el software. En la guía para la instalación de *Maven* se hace referencia a que se debe disponer de Windows 8 para el correcto funcionamiento del gestor de proyectos *software*, aunque se ha probado con Windows 7 y funciona perfectamente. Sin embargo, las versiones de *Apache Maven* y *Java Development Kit* (JDK) indicadas si se deben respetar con el fin de evitar problemas de funcionamiento.

El siguiente paso para instalar *Maven*, es acceder al menú de variables de entorno del sistema operativo. Para el caso de Windows 7 se debe seguir la siguiente ruta:

"*Panel de control -> Sistema -> Configuración avanzada del sistema -> Variables de entorno*"

En esta pantalla se debe comprobar que se encuentra definida la variable de entorno `JAVA_HOME` con la ubicación del archivo JDK (En la carpeta de *Java*). También se deben crear las variables `M2_HOME` y `MAVEN_HOME` con la ruta de la carpeta en la que se descomprimió el archivo comprimido *Maven*. Además de todo esto, en la variable *Path* hay que añadir la siguiente instrucción:

```
%M2_HOME%\bin
```

Para verificar que todo el proceso de instalación se ha realizado correctamente, se debe abrir una ventana de comandos y ejecutar la siguiente sentencia:

```
mvn –versión
```

El resultado debe ser similar al que se muestra en el paso 5 de la guía de la referencia [62].

Una vez hecho esto, se puede pasar a la creación de la aplicación. Para ello se debe acceder a una ventana de comandos y se debe introducir la siguiente sentencia:

```
cd C:\Program Files\apache-maven-3.5.3 -
DgroupId=org.thethingsnetwork.samples.mqtt -DartifactId=mqtt -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Seguidamente, se debe ejecutar la sentencia:

```
cd mqtt
```

Al hacer esto, se encuentra en la ruta "*C:\Program Files\apache-maven-3.5.3\mqtt*" un archivo denominado *pom.xml* que se debe completar con el código que se muestra en la Figura 8.1 y en la Figura 8.2.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.thethingsnetwork.samples.mqtt</groupId>
  <artifactId>mqtt</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>mqtt</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.thethingsnetwork</groupId>
      <artifactId>data-mqtt</artifactId>
      <version>2.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20180130</version>
    </dependency>
    <dependency>
      <groupId>org.apache.ant</groupId>
      <artifactId>ant</artifactId>
      <version>1.8.1</version>
      <type>jar</type>
    </dependency>
    <dependency>
```

Figura 8.1: POM.xml.

Algunas de las dependencias de la Figura 8.1 y de la Figura 8.2 no son prioritarias, pero pueden facilitar acciones en la edición de la aplicación. Es importante comprobar que las versiones de las dependencias son las adecuadas, como, por ejemplo, la dependencia *thethingsnetwork*, en la que es fundamental revisar que al menos se supere la versión 2.1.1. También se debe comprobar que dentro de las propiedades que se ven en las líneas

44-48, se cuenta con la versión 1.8. De todas formas, en la referencia [64] se incluye un archivo *pom.xml* que cumple a la perfección los requisitos que se acaban de indicar.

```
33      <dependency>
34        <groupId>com.fasterxml.jackson.core</groupId>
35        <artifactId>jackson-annotations</artifactId>
36        <version>2.9.0</version>
37      </dependency>
38      <dependency>
39        <groupId>org.jdesktop</groupId>
40        <artifactId>beansbinding</artifactId>
41        <version>1.2.1</version>
42      </dependency>
43    </dependencies>
44    <properties>
45          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
46          <maven.compiler.source>1.8</maven.compiler.source>
47          <maven.compiler.target>1.8</maven.compiler.target>
48      </properties>
49  </project>
```

Figura 8.2: POM.xml_1.

Tras editar el archivo *pom.xml*, se ejecuta en una ventana de comandos la siguiente instrucción para comprobar que todo marcha correctamente. Al hacerlo, aparecerá un mensaje en pantalla que certifica que se ha completado la creación de la aplicación.

```
mvn clean compile
```

Una vez que se tiene la aplicación creada, puede abrirse la aplicación con un editor y se puede empezar a programar. En el presente TFG, el entorno utilizado para la programación de la aplicación *Java* es el *Integrated Development Environment* (*IDE*) *Netbeans* [65], un entorno en el que se ha comprobado que es completamente viable la edición y ejecución del archivo creado en los pasos anteriores. No obstante, para establecer correctamente la comunicación entre TTN y la aplicación, hay que seguir una serie de pasos adicionales.

En primer lugar, se deben incluir en el archivo *App.java* las librerías asociadas a la interacción con el entorno TTN. Las librerías que se deben importar se muestran en la Figura 8.3.

```
import org.thethingsnetwork.data.common.Connection;
import org.thethingsnetwork.data.common.messages.ActivationMessage;
import org.thethingsnetwork.data.common.messages.DataMessage;
import org.thethingsnetwork.data.common.messages.DownlinkMessage;
import org.thethingsnetwork.data.common.messages.RawMessage;
import org.thethingsnetwork.data.common.messages.UplinkMessage;
import org.thethingsnetwork.data.mqtt.Client;
```

Figura 8.3: Librerías para la interacción Java-TTN.

A continuación, se debe ir a la aplicación que se creó en TTN y consultar la región de trabajo, el identificador de la aplicación (*AppID*), y la clave de acceso a dicha aplicación (*AppKey*).

Una vez que se tenga esta información, se debe adjuntar el siguiente código, adaptado a cada caso, en la función *main()* del archivo *App.java*, tal y como se muestra en la Figura 8.4.

```
public static void main(String[] args) throws Exception {
    //Parámetros para establecer la comunicación con la aplicación TTN.
    String region = "eu";
    String appId = "lopy-application-fbt-1";
    String accessKey = "ttn-account-v2.kT8E4eIsBqp9u7iQ2eD0OwBN8h9un5Tg6EWKYoDTx8Y";
    //Se crea el cliente con los parámetros establecidos.
    Client client = new Client(region, appId, accessKey);
```

Figura 8.4: Creación del objeto cliente en App.java.

Como se observa en la Figura 8.4, con los parámetros introducidos se crea un objeto de la clase *Client*. Con este objeto se podrán gestionar las funciones asociadas a la comunicación con la plataforma TTN.

Otro apunte importante que se debe comentar antes de empezar con la adecuación de la aplicación a las características que requiere la solución del presente TFG, es que al final de la función *main()* se debe llamar al método *start()*, que normalmente se deja como último método dentro de la función *main()*, y que inicia el funcionamiento del cliente *Java*. La llamada a la función *start()* se visualiza en la Figura 8.5.

```
105
106             client.onActivation((String _devId, ActivationMessage _data) ->
107                     System.out.println("Activation: " + _devId + ", data: " + _data.getDevAddr()));
108
109             client.onError((Throwable _error) ->
110                     System.err.println("error: " + _error.getMessage()));
111
112             client.onConnected((Connection _client) ->
113                     System.out.println("connected !"));
114
115             client.start();
116
117         }
118
119     }
```

Figura 8.5: Llamada a la función start() en App.java.

Para ejecutar la aplicación se tienen dos opciones, o bien se ejecuta directamente con las utilidades que aporta *Netbeans*, o bien se ejecuta mediante la siguiente sentencia en una ventana de comandos:

```
mvn clean compile exec:java
-Dexec.mainClass="org.thethingsnetwork.samples.mqtt.App"
```

A partir de este punto, ya se puede proceder a la programación de la aplicación *Java* para adecuarla a las necesidades de la solución del presente TFG.

8.2 Desarrollo de la aplicación *Java*

La aplicación desarrollada consta de 4 archivos *.java*. A continuación, se comenta brevemente cada uno de ellos y qué función tienen dentro de la aplicación completa.

- `App.java`: Archivo creado a partir de las indicaciones del apartado anterior. En este archivo establece la comunicación ascendente y descendente con el servidor de la plataforma *The Things Network*. Incluye todas las funciones que permiten establecer dicha comunicación.

- `VentanaP.java`: Archivo que contiene la ventana principal de la interfaz de la aplicación. Desde esta ventana principal se da la opción de abrir las otras dos ventanas que constituyen la interfaz de la aplicación, correspondiente a la ventana de configuración, y la ventana de estado.

- `StatusPanel.java`: Archivo que contiene la ventana que permite visualizar el estado de la plaza de aparcamiento asociada al nodo final.

- `ConfigurationPanel.java`: Archivo que contiene la ventana que permite configurar los parámetros del nodo final.

Como se vio en el aparatado de creación y configuración del archivo *App.java*, se deben incluir en este *script* las librerías que contienen las funciones que permiten interactuar con la plataforma TTN. En la Figura 8.6 se muestra la lista de librerías utilizadas.

En la última versión de código desarrollado, la relación de librerías utilizadas es la que se muestra en la Figura 8.6. Las librerías de TTN que se incluyen en esta relación son las que contienen las funciones de conexión, mensajes de activación, mensajes de datos, mensajes tipo *Downlink* y *Uplink*, y todas las funciones de la clase *Client*. Además de estas librerías de TTN, también se puede ver en la Figura 8.6 que se hace uso de las librerías *Map* y *HashMap*, que permiten la gestión de variables que sean de ese tipo.

```
1
2   package org.thethingsnetwork.samples.mqtt;
3
4
5   import java.util.HashMap;
6   import java.util.Map;
7   import org.thethingsnetwork.data.common.Connection;
8   import org.thethingsnetwork.data.common.messages.ActivationMessage;
9   import org.thethingsnetwork.data.common.messages.DataMessage;
10  import org.thethingsnetwork.data.common.messages.DownlinkMessage;
11  import org.thethingsnetwork.data.common.messages.UplinkMessage;
12  import org.thethingsnetwork.data.mqtt.Client;
13
```

Figura 8.6: Librerías utilizadas en App.java.

En la Figura 8.7 se muestra la clase principal del archivo, correspondiente a la clase *App*. Dentro de esta clase se desarrolla todo el código orientado a establecer la comunicación con la plataforma TTN. Lo primero que se hace, como se puede observar en la Figura 8.7, es declarar variables propias de la clase, como son *MapDef* y *floresMain*. También se crea una clase denominada *Example* en la que las variables declaradas se utilizarán en otros archivos *java* a modo de variables globales. Como se puede deducir de los nombres de las

variables, estas se corresponden con parámetros de configuración y con valores de estado del nodo final.

Haciendo uso de las variables declaradas, se crea el constructor de la clase *App*, que tiene como parámetro una variable de tipo *String* y otra variable de tipo *HashMap*. El constructor creado se muestra en la Figura 8.8.

Por otra parte, antes de comenzar a analizar la función *main()* del archivo *App.java* se debe explicar la función *imprimir(),* cuyo código se adjunta en la Figura 8.9. Esta función se desarrolló con el objetivo de visualizar en pantalla ciertos datos que se van recogiendo desde la plataforma TTN. Es decir, es una función útil para la depuración del código, o bien para asegurar que este cumple con la funcionalidad establecida. En esta función se recogen datos de la variable *map2* y se almacenan en variables locales para mostrar su valor por pantalla. Como se verá en las funciones que permiten la comunicación con la plataforma TTN, la variable *map2* es la que contiene toda la información de las tramas recibidas desde TTN, y es por ello por lo que se extrae la información para mostrarla a partir de esta variable. También se puede ver en la línea 55, que se realiza una transferencia de información desde la variable global *map2* a la variable *mapDef*, al igual que en la línea 56 se extrae un valor concreto de un campo específico a partir de la misma variable (*map2*). Esto se hace para comprobar que la creación de variables y la recepción de la información recogida desde la plataforma TTN son adecuadas.

Ahora ya se puede proceder al análisis de la función principal de este archivo, la función *main()*. Lo primero que se debe codificar en la función principal, es la creación de un cliente TTN. Para la creación de este cliente es necesario incluir la región de trabajo, el identificador de la aplicación de TTN, y su clave asociada. Con estos parámetros se puede crear el cliente, tal y como se muestra en la Figura 8.10.

Lo siguiente que se crea es una clase denominada *Response*. En esta clase se crean las variables que van a contener la información del envío tipo *Downlink*. La declaración de esta clase y su contenido se adjunta en la Figura 8.11. Como se puede ver en la Figura 8.11, las variables creadas se corresponden con los parámetros de configuración del nodo final. Esto se debe a que en el mensaje de tipo *Downlink* se envía la trama de configuración, que tendrá los valores que se introduzcan en la ventana de configuración

del archivo `ConfigurationPanel.java`. En este archivo se recogen los datos introducidos por el usuario y se almacenan en las variables globales creadas en el archivo `App.java` que se está analizando, mientras que en la función *Response()*, dentro de su clase, se transfiere el valor de esas variables globales a las variables creadas en la propia clase *Response*. De esta manera, y como se verá en el análisis de la función *onMessage()*, se envía la información que almacenan las variables de la clase *Response*, en mensajes de tipo *Downlink*.

```
19  public class App {
20
21      //Variables globales
22      public static class Example {
23          public static Map<String, Object> map2 = new HashMap<String, Object>();
24          public static int contador = 0;
25          public static String ParkingSlotStatus;
26          public static byte[] byteprueba;
27          public static String asignarID;
28          public static int ID2config;
29          public static int nodeID;
30          public static int firmwareVersion;
31          public static int NMStart;
32          public static int NMPeriod;
33          public static int NMSleepTime;
34          public static int NMKeepAlive;
35          public static int SleepTime;
36          public static int KeepAlive;
37          public static int threshold;
38          public static boolean flagConfig;
39      }
40
41      static Map<String, Object> mapDef;
42      static String floresMain;
```

Figura 8.7: Clase principal de App.java y variables declaradas.

```
45      public App(String floresMain, Map<String, Object> mapDef ){
46
47          this.floresMain = floresMain;
48          this.mapDef = mapDef;
49      }
```

Figura 8.8: Constructor de App.java.

```
51      //Método para la extracción y visualización de los datos de cada trama.
52      public static void imprimir(){
53              String ParkingSlotStatus = (String) Example.map2.get("ParkingSlotStatus");
54              String tipo = (String) Example.map2.get("tipo");
55              mapDef = App.Example.map2;
56              floresMain = (String) App.Example.map2.get("ParkingSlotStatus");
57              System.out.println(ParkingSlotStatus);
58              System.out.println("Status:" + floresMain);
59              System.out.println(tipo);
60          }
61
```

Figura 8.9: Función imprimir() de App.java.

```
62      public static void main(String[] args) throws Exception {
63          //Parámetros para establecer la comunicación con la aplicación TTN.
64          String region = "eu";
65          String appId = "lopy-application-fbt-1";
66          String accessKey = "ttn-account-v2.kT8E4eIsBqp9u7iQ2eD0OwBN8h9un5Tg6EWKYoDTx8Y";
67          //Se crea el cliente con los parámetros establecidos.
68          Client client = new Client(region, appId, accessKey);
```

Figura 8.10: Función main() y creación del cliente TTN.

```
69          //Clase y método para responder cuando se reciba una trama.
70          class Response {
71
72              private String asignarid;
73              private int firmwareversion, id2config, nodeid, nmstart, nmperiod;
74              private int nmsleeptime, nmkeepalive, sleeptime, keepalive, threshold;
75              public Response() throws Exception {
76
77                  asignarid = Example.asignarID;
78                  id2config = Example.ID2config;
79                    nodeid = Example.nodeID;
80                firmwareversion= Example.firmwareVersion;
81                    nmstart = Example.NMStart;
82                   nmperiod = Example.NMPeriod;
83                 nmsleeptime= Example.NMSleepTime;
84                nmkeepalive = Example.NMKeepAlive;
85                  sleeptime = Example.SleepTime;
86                  keepalive = Example.KeepAlive;
87                   threshold = Example.threshold;
88              }
89          }
```

Figura 8.11: Clase Response en App.java.

Explicada la clase *Response*, se puede proceder al análisis de las funciones que interactúan directamente con la plataforma TTN. Estas funciones son las que se muestran en la Figura 8.12 y en la Figura 8.13.

Así, en la Figura 8.12 se llama a la función *onMessage()* sobre el objeto cliente creado con anterioridad. En este caso, se utiliza esta función para enviar un mensaje de tipo *Downlink*. En la documentación de la tecnología LoRa/LoRaWAN se estudió que el envío

de información al nodo final se realizaba a través de ventanas de recepción que se daban tras una transmisión ascendente desde el propio nodo. Sobre la base de esta teoría, al recibir un mensaje ascendente del nodo final, se envía inmediatamente un mensaje descendente para aprovechar dicha ventana de recepción. Como se observa en la Figura 8.12, al recibir un paquete que contenga el campo *tipo*, se crea un mensaje de tipo *Downlink* con el contenido que genera la función *Response()* explicada con anterioridad. Sin embargo, este mensaje solo se envía en caso de que el *flag flagConfig* esté activado. Este *flag* se activa al pulsar un botón en la ventana creada en el archivo `ConfigurationPanel.java`. De esta manera, solo se envía el mensaje *Downlink* cuando el usuario pulse dicho botón en la interfaz. Por lo tanto, si el *flag* está activado, se llama a la función *send()* para enviar el mensaje creado, y automáticamente se desactiva el *flag* para que no siga enviando mensajes cada vez que reciba un paquete que incluya el campo *tipo*. Todo este funcionamiento se realiza dentro de una estructura *try-catch*, y en caso de error se muestra un mensaje que indique esa situación.

```
93          //Si se recibe una trama desde TTN que incluya el field "tipo" y el flag
94          //de configuración está activado, se responde con un mensaje downlink
95          //que se visualiza en TTN(Data)
96          client.onMessage(null, "tipo", (String _devId, DataMessage _data) -> {
97              try {
98                  DownlinkMessage response = new DownlinkMessage(0, new Response());
99                  //Comprobamos si en ConfigurationPanel se ha pulsado el botón enviar
100                 if(Example.flagConfig == true){
101                  client.send(_devId, response);
102                  //Se desactiva el flag
103                  Example.flagConfig = false;
104                 }
105             } catch (Exception ex) {
106                 System.out.println("Response failed: " + ex.getMessage());
107             }
108         });
```

Figura 8.12: Uso de la función onMessage() para el envío de mensaje tipo Downlink.

En la Figura 8.13 se vuelve a hacer uso de la función *onMessage()*, pero esta vez se recoge el parámetro de la función y se transforma en un mensaje de tipo *Uplink*, para posteriormente extraer del mensaje los campos recibidos y sus valores asociados. Esta información se almacena en la variable global *map2*. Tras transferir los datos a esta variable, se llama a la función *imprimir()*, que, como se explicó anteriormente, permite visualizar el valor de los campos recibidos en el mensaje *Uplink*.

El resto de funciones que se muestran en la Figura 8.13 son *onActivation()*, *onError()* y *onConnected()*, que se encargan de mostrar mensajes de activación, de error y de conexión establecida, respectivamente. También se puede ver en la zona inferior de la Figura 8.13 la llamada a la función *start()*, que inicia el funcionamiento del objeto *Client* creado.

Con la llamada a estas funciones dentro de la función principal, *main()*, finaliza el código del archivo `App.java`. A continuación, se explica el código correspondiente al archivo `VentanaP.java`.

Como se indicó anteriormente, el archivo `VentanaP.java` crea una interfaz que permite al usuario acceder a la pantalla de lectura del estado de la plaza de aparcamiento asociada al nodo final o a la ventana de configuración del nodo final. La interfaz desarrollada se muestra en la Figura 8.14.

Para crear la interfaz de la Figura 8.14 se han utilizado los elementos que se muestran en la Figura 8.15, correspondientes a un panel que contiene a los otros 3 elementos, 2 botones que permiten seleccionar al usuario la opción deseada, y una etiqueta con el nombre de los desarrolladores.

```
110             client.onMessage((String devId, DataMessage data) -> {
111                 UplinkMessage message = (UplinkMessage) data;
112                 Map<String, Object> map = message.getPayloadFields();
113                 Example.map2 = map;
114                 //Se llama al método imprimir cuando se recibe un mensaje.
115                 imprimir();
116             });
117 
118             client.onActivation((String _devId, ActivationMessage _data) ->
119                     System.out.println("Activation: " + _devId + ", data: " + _data.getDevAddr()));
120 
121             client.onError((Throwable _error) ->
122                     System.err.println("error: " + _error.getMessage()));
123 
124             client.onConnected((Connection _client) ->
125                     System.out.println("connected !"));
126 
127             client.start();
128 
129         }
130 
131     }
```

Figura 8.13: Función OnMessage() para la recepción de mensaje tipo Uplink, función onActivation(), onError(), onConnected() y start().

En la Figura 8.16 se comienza a analizar el código de este archivo *java*. En esta figura se aprecia la declaración de la clase `VentanaP`, su constructor, y el código que se autogenera en el editor *Netbeans* al añadir los elementos indicados en la Figura 8.15.

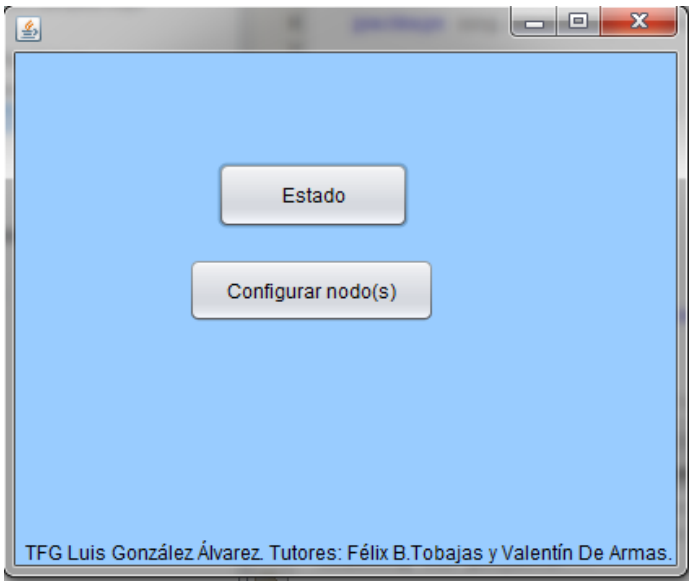


Figura 8.14: Interfaz generada en el archivo VentanaP.java

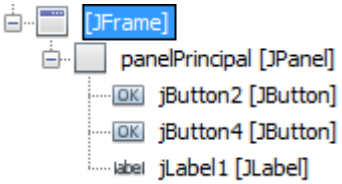


Figura 8.15: Elementos de la interfaz VentanaP.

```
6   package org.thethingsnetwork.samples.mqtt;
7
8
9   import java.util.HashMap;
10  import java.util.Map;
11
12  /**...4 lines */
16  public class VentanaP extends javax.swing.JFrame {
17
18      /** Creates new form VentanaP ...3 lines */
21      public VentanaP() {
22          initComponents();
23      }
24
25      /** This method is called from within the constructor to initialize the form
30      @SuppressWarnings("unchecked")
31      Generated Code
```

Figura 8.16: Clase VentanaP.

Así, en esta interfaz se disponen 2 botones, cada uno de ellos redirigiendo al usuario a una ventana secundaria diferente. El elemento `jButton2`, que se asocia al texto *Configurar nodo(s)* de la Figura 8.14, al ser pulsado hace visible la ventana de configuración que lanza el archivo `ConfigurationPanel.java`. Para ello, en primer lugar, se crea un objeto de la clase `ConfigurationPanel` y, posteriormente, se llama a la función *setVisible()*, que permite mostrar u ocultar un panel determinado. Para el elemento `jButton4`, asociado al texto *Estado* de la Figura 8.14, el procedimiento es exactamente el mismo que para el elemento `jButton2`, solo que en esta ocasión el objeto creado es de la clase `StatusPanel`. Con este código, que se adjunta en la Figura 8.17, se pueden visualizar ambas ventanas.

```
102     private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
103         // TODO add your handling code here:
104         ConfigurationPanel panelConfig = new ConfigurationPanel();
105         panelConfig.setVisible(true);
106     }
107
108     private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
109         // TODO add your handling code here:
110         StatusPanel panelEstado = new StatusPanel();
111         panelEstado.setVisible(true);
112     }
```

Figura 8.17: Acciones de los botones de VentanaP.java.

Por último, en la función *main()* del archivo `VentanaP.java`, que se muestra en la Figura 8.18, se puede ver cómo se crea un objeto de la clase `App` del archivo `App.java` y se llama a la función *main()* de dicha clase. Al hacer esto, se puede ejecutar el archivo `VentanaP.java` y en el inicio de esta función se ejecuta también la comunicación con la plataforma TTN, que se había programado en la función principal del archivo `App.java`. En la Figura 8.14 se muestra la visualización de la propia interfaz creada en el archivo `VentanaP.java`.

Por otro lado, la interfaz principal generada por el archivo `StatusPanel.java` es la que se muestra en la Figura 8.19, y los elementos que forman dicha interfaz se pueden visualizar en la Figura 8.20.

En la Figura 8.20 aparece la lista de elementos que constituyen la interfaz del panel de estado de la plaza de aparcamiento asociada al nodo final. Los elementos se resumen en

un panel que contiene el resto de los elementos, etiquetas para introducir texto en la interfaz, un panel para indicar el estado de la plaza de aparcamiento, un botón para iniciar la visualización del estado, y una etiqueta adicional para señalar el nombre de los desarrolladores.

En el código del archivo `StatusPanel.java` se gestionan utilidades como la edición de los colores de los paneles, o la gestión de *Timers*. Para poder hacer uso de ellas es necesario incorporar las librerías adecuadas, que se muestran en la Figura 8.21.

```
    public static void main(String args[]) throws Exception {
        /* Set the Nimbus look and feel */
        Look and feel setting code (optional)

        String rosa = new String();
        Map<String, Object> mapDef = new HashMap<String, Object>();
        String[] hola = {""};
        App otraClase = new App(rosa,mapDef);
        otraClase.main(hola);

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new VentanaP().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton jButton2;
    private javax.swing.JButton jButton4;
    private javax.swing.JLabel jLabel1;
    public static javax.swing.JPanel panelPrincipal;
    // End of variables declaration
}
```

Figura 8.18: Función main() del archivo VentanaP.java.

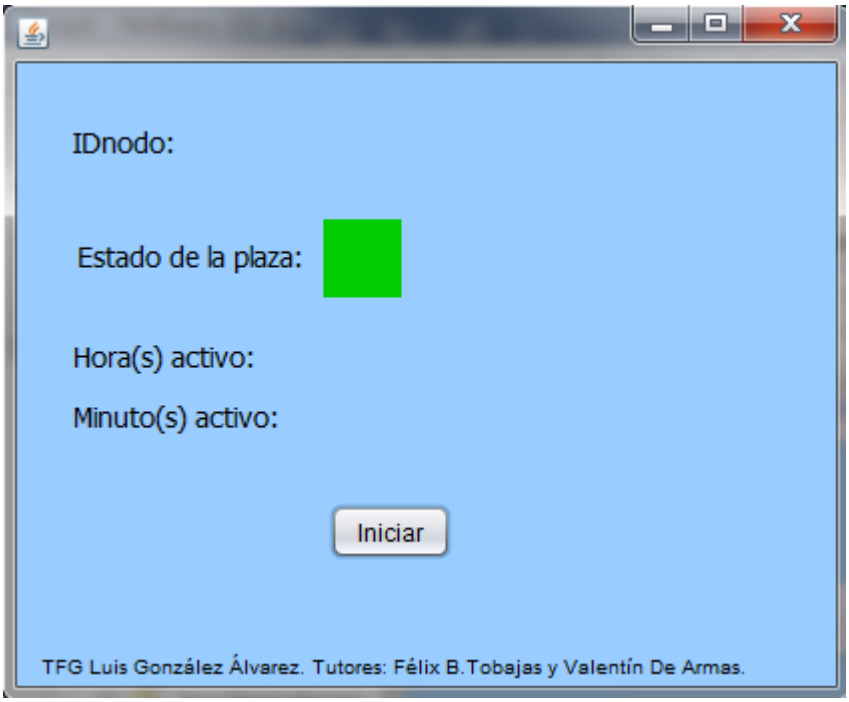


Figura 8.19: Interfaz generada por el archivo StatusPanel.java.

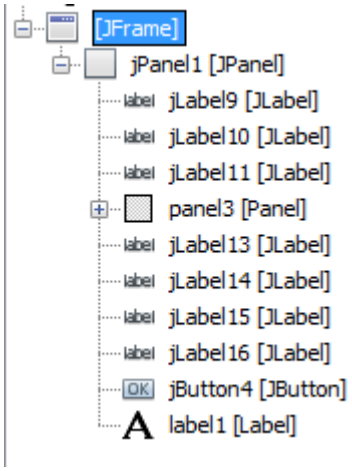


Figura 8.20: Elementos de la interfaz generada por el archivo StatusPanel.java.

```
6   package org.thethingsnetwork.samples.mqtt;
7   import java.awt.Color;
8   import java.awt.event.ActionEvent;
9   import java.awt.event.ActionListener;
10  import javax.swing.JLabel;
11  import javax.swing.Timer;
12
13  /**...4 lines */
17  public class StatusPanel extends javax.swing.JFrame {
18
19      /** Creates new form StatusPanel ...3 lines */
22      public StatusPanel() {
23          initComponents();
24      }
25
26      public static class Example3 {
27       public static int contador = 0;
28       public static int IDnodoFijo = -2;
29       public static int currentHours = -1;
30       public static int currentMinutes = -1;
31      }
```

Figura 8.21: Librerías y clases del archivo StatusPanel.java.

En la Figura 8.21 también se puede ver la declaración de la clase *StatusPanel*, así como su constructor, en el que se inicializan los componentes. También se ha creado una clase, denominada *Example3*, para la gestión de variables globales. El uso de estas variables se presentará más adelante, por ahora solo se inicializan.

En la Figura 8.22 se declara la clase *TimerToLabel*, que permite asociar un tiempo de refresco a las etiquetas. Se hará uso de esta clase y sus funcionalidades para refrescar automáticamente el valor de una etiqueta cuando se reciba información desde la plataforma TTN. Al constructor se le pasa por parámetros un tiempo de actualización y el nombre de la etiqueta. En la función *init()* se inicializa la cuenta del *Timer*.

Cuando se cumple el tiempo establecido en el parámetro *delay* que se le pasa al constructor *TimerToLabel()*, se ejecuta la acción que se codificó en la función *actionPerformed()*. En este código, que se adjunta en la Figura 8.23 y en la Figura 8.24, se declaran e inicializan dos variables locales: `IDnodo` y `ParkingSlotStatus`. Seguidamente, se desarrolla una estructura que permite establecer el ID del nodo. Se ha codificado esta funcionalidad porque la aplicación es mono-nodal, es decir, que solo se muestran datos de un único nodo final, concretamente del primero del que se reciban sus datos. Por lo tanto, en esta estructura se extrae el ID del nodo desde la variable global *map2* del archivo `App.java`, y se establece el valor obtenido a la variable local declarada anteriormente, `IDnodo`. Para el desarrollo de esta estructura se ha utilizado

un contador, de manera que solo se permita entrar una vez a la opción que permite establecer el identificador del nodo final.

```
33  private class TimerToLabel implements ActionListener {
34
35      private Timer timer;
36      private final JLabel label;
37      private final int delay;
38
39      public TimerToLabel(final int delay, final JLabel label) {
40          this.delay = delay;
41          this.label = label;
42      }
43
44      public void init() {
45          timer = new Timer(delay, this);
46          timer.start();
47      }
```

Figura 8.22: Clase TimerToLabel, constructor e inicialización.

Con el identificador del nodo final establecido, se codifica una sentencia *if* que filtre los mensajes que provienen de ese nodo y descarte el resto automáticamente.

En caso de que el nodo esté enviando información, se extraen de la variable global `map2` todos los campos necesarios para mostrar el estado del nodo.

Como se ha visto a lo largo del presente TFG, existen diferentes tipos de tramas, y cada tipo de trama contiene campos diferentes, por lo tanto, aquellos campos que no sean fijos en todos los paquetes se extraen solo en caso de haber recibido el tipo de trama al que correspondan, evitando así posibles errores de operación. Como se puede ver en la Figura 8.23, los campos recogidos son: `IDnodo`, `tipo`, `ParkingSlotStatus`, `currentHours` y `currentMinutes`. Todos estos valores se muestran en la interfaz, salvo el campo `tipo`, que se utiliza únicamente para discriminar las tramas recibidas.

```
49      @Override
        public void actionPerformed(final ActionEvent e) {
51          String ParkingSlotStatus = "";
            int IDnodo = -1;
53
54          //Aplicación mononodal: El primer nodo en conectar es el único
55          //del que se decodifica su información.
56          //Recogemos el valor del nodo
57          IDnodo =  (int) App.Example.map2.get("IDnodo");
58          if(Example3.contador == 0){
59          //Si es un ID válido, será mayor que 0.
60           if(IDnodo >= 1){
61              //Fijamos el ID al valor obtenido
62              Example3.IDnodoFijo = IDnodo;
63              Example3.contador++;
64           }
65          }
66          //Sólo se recogen datos de las tramas que envie el nodo con ID igual al
67          //guardado en la variable IDnodoFijo.
68          if(IDnodo == Example3.IDnodoFijo){
69          String tipo = (String) App.Example.map2.get("tipo");
70          if("Info".equals(tipo)){
71           ParkingSlotStatus = (String) App.Example.map2.get("ParkingSlotStatus");
72          }else if("KeepAlive".equals(tipo)){
73           Example3.currentHours = (int) App.Example.map2.get("currentHours");
74           Example3.currentMinutes = (int) App.Example.map2.get("currentMinutes");
75          }
```

Figura 8.23: Acción que se realiza al cumplirse el tiempo del timer.

En la Figura 8.24 se muestra la funcionalidad que permite modificar el color del panel que indica el estado de la plaza de aparcamiento. Si el estado de la plaza es “Libre” el panel se muestra en color verde, mientras que si el estado es “Ocupado” el panel adquiere el color rojo.

Por último, se edita el texto de las etiquetas asociadas al ID del nodo, a las horas transcurridas y a los minutos transcurridos. Estas etiquetas se actualizan con la función *updateUI()* para poder ser visualizadas en la interfaz con sus nuevos valores.

En la Figura 8.25 se muestra el código de acción ante un evento en el botón `jButton4`, asociado al texto *Iniciar* en la Figura 8.19. Al presionar este botón se crea un objeto de la clase `TimerToLabel` y se inicia la cuenta del *Timer*. Para que se produzca un evento al finalizar la cuenta del T*imer*, basta con introducir un tiempo de *delay* en milisegundos y una etiqueta utilizada en la interfaz, como por ejemplo *jLabel10*.

```
76
77              if("Libre".equals(ParkingSlotStatus)){
78                  panel3.setBackground(Color.green);
79              }else if("Ocupado".equals(ParkingSlotStatus)){
80                  panel3.setBackground(Color.red);
81              }
82
83              jLabel10.setText("" + Example3.IDnodoFijo);
84              jLabel14.setText("" + Example3.currentHours);
85              jLabel16.setText("" + Example3.currentMinutes);
86              }
87
88              jLabel10.updateUI();
89              jLabel14.updateUI();
90              jLabel16.updateUI();
91          }
92      }
```

Figura 8.24: Acción que se realiza al cumplirse el tiempo del timer_2.

```
98      @SuppressWarnings("unchecked")
99      Generated Code
298
299     private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
300
301     }
302
303     private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
304         // TODO add your handling code here:
305        final TimerToLabel timer = new TimerToLabel(1000, jLabel10);
306        timer.init();
307     }
```

Figura 8.25: Acción ante un evento en el botón jButton4.

Finalmente, en la Figura 8.26 se adjunta el código de la función principal *main()* del archivo `StatusPanel.java`. En esta función se hace visible la interfaz creada.

Por su parte, en el archivo `ConfigurationPanel.java`. Se crea una interfaz que permite al usuario configurar el nodo final desde esta aplicación *Java*. La interfaz que genera este archivo se muestra en la Figura 8.27.

En la Figura 8.28 se adjunta la lista de los elementos que forman la interfaz mostrada en la Figura 8.27.

```
312     public static void main(String args[]) {
313         /* Set the Nimbus look and feel */
314         Look and feel setting code (optional)
335
336         /* Create and display the form */
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
339                 new StatusPanel().setVisible(true);
340             }
341         });
342     }
```

Figura 8.26: Función main() del archivo StatusPanel.java.

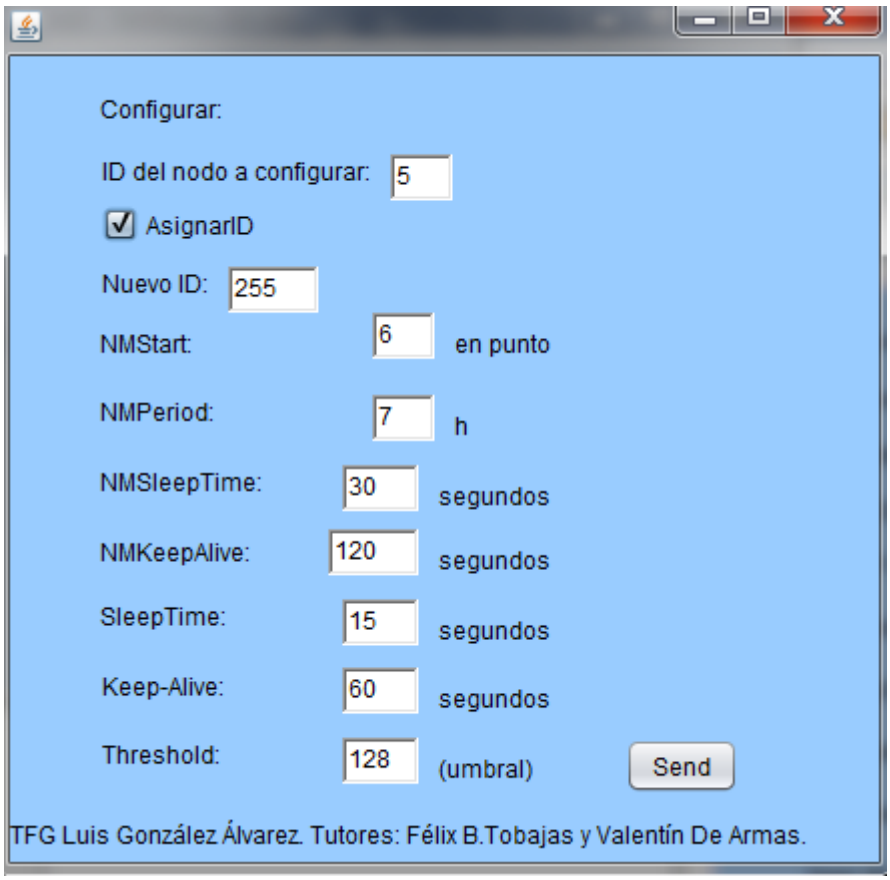


Figura 8.27: Interfaz generada por el archivo ConfigurationPanel.java.

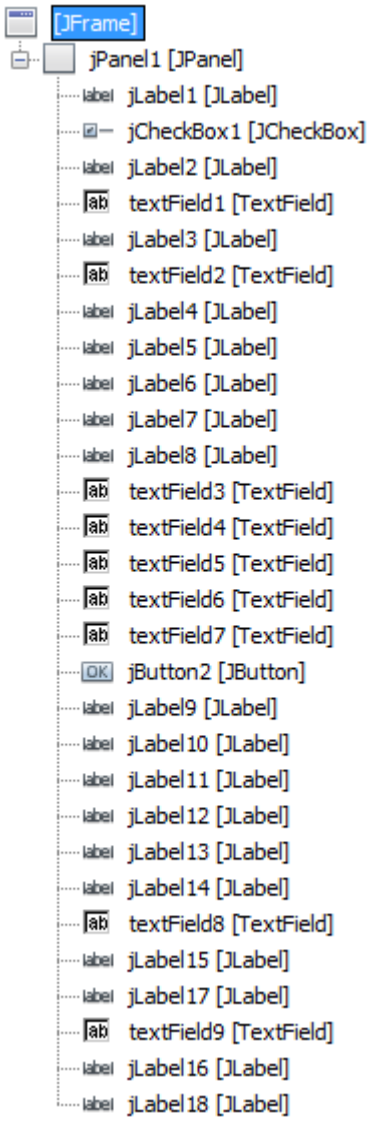


Figura 8.28: Elementos de la interfaz del archivo ConfigurationPanel.java.

Los elementos utilizados son los mismos que en los archivos `VentanaP.java` y `StatusPanel.java`, a excepción de los elementos *checkBox* y *textField*, que permiten habilitar/deshabilitar una opción e introducir valores numéricos, respectivamente.

En la Figura 8.29 se puede ver la creación de la clase *ConfigurationPanel*, con su correspondiente constructor y con la clase *Example2*, que se utiliza para la gestión de variables globales. Las variables globales declaradas coinciden con los parámetros de configuración del nodo final.

```
12    public class ConfigurationPanel extends javax.swing.JFrame {
13
14        //Variable global
15        public static class Example2 {
16         public static String asignarID;
17         public static int ID2config;
18         public static int nodeID;
19         public static int firmwareVersion;
20         public static int NMStart;
21         public static int NMPeriod;
22         public static int NMSleepTime;
23         public static int NMKeepAlive;
24         public static int SleepTime;
25         public static int KeepAlive;
26         public static int threshold;
27        }
28        /**
29         * Creates new form ConfigurationPanel
30         */
31        public ConfigurationPanel() {
32            initComponents();
33        }
```

Figura 8.29: Clase ConfigurationPanel, constructor y clase Example2.

Las variables declaradas en la Figura 8.29 se actualizan con los valores introducidos por el usuario a través del código que se muestra en la Figura 8.30 y en la Figura 8.31. En esta función se llevan a cabo ciertas acciones cuando se produce un evento sobre el botón `jButton2`. Este botón está asociado al texto "Send" de la Figura 8.27, y cuando se pulsa, se activa inmediatamente el *flag flagConfig*, que da permiso para enviar un mensaje tipo *Downlink* en el `archivoApp.java`. También se incrementa el valor de la variable *firmwareVersion*, que indica el número de configuraciones que el usuario ha realizado.

Después de tratar estas dos variables, se recogen de los elementos *textField* los valores introducidos por el usuario y se almacenan en su correspondiente variable global. Tras almacenarlas, como se muestra en la Figura 8.31, se indican por pantalla los valores almacenados de todos ellos, con lo que se muestra la configuración completa realizada por el usuario.

En la Figura 8.32 se indica cómo se transfieren los valores de las variables de la clase *Example2* a las variables globales del archivo `App.java`, que es donde se pueden enviar

dichos parámetros en un mensaje *Downlink* hacia la plataforma TTN, y, por tanto, hacia el nodo final.

La siguiente función, cuyo código se adjunta en la Figura 8.32, realiza unas acciones específicas en caso de que se produzca un evento sobre el elemento `jCheckBox1`. Si el elemento está seleccionado, se cambia el valor de la variable *asignarID* a "si", mientras que, si no está seleccionado, el valor de esta variable pasa a ser "no".

```
311     private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
312         // TODO add your handling code here:
313         //Al enviar una trama, se activa un flag para que en app.java se pueda
314         //enviar la trama de configuración.
315         App.Example.flagConfig = true;
316         //Una nueva confuguración, por lo tanto se aumenta el número de
317         //la versión del firmware.
318         App.Example.firmwareVersion++;
319 
320         Example2.ID2config = Integer.parseInt(textField8.getText());
321         Example2.nodeID = Integer.parseInt(textField1.getText());
322         Example2.NMStart = Integer.parseInt(textField2.getText());
323         Example2.NMPeriod = Integer.parseInt(textField3.getText());
324         Example2.NMSleepTime = Integer.parseInt(textField4.getText());
325         Example2.NMKeepAlive = Integer.parseInt(textField5.getText());
326         Example2.SleepTime = Integer.parseInt(textField6.getText());
327         Example2.KeepAlive = Integer.parseInt(textField7.getText());
328         Example2.threshold = Integer.parseInt(textField9.getText());
329         System.out.println( "AsignarID: " + Example2.asignarID + ", ID2config: 
330                 + Example2.ID2config  + ", nodeID: " + Example2.nodeID +
331                 ", NMStart: " + Example2.NMStart + ", NMPeriod: " +
332                 Example2.NMPeriod + ", NMSleepTime: " + Example2.NMSleepTime +
333                 ", NMKepAlive: " + Example2.NMKeepAlive +", NMSleepTime: " +
334                 Example2.SleepTime + ", NMKeepAlive: " + Example2.KeepAlive +
335                 ", Threhold: " + Example2.threshold);
```

Figura 8.30: Acción que se ejecuta ante un evento en el botón jButton2.

```
336 
337         App.Example.asignarID = Example2.asignarID;
338         App.Example.ID2config = Example2.ID2config;
339         App.Example.nodeID = Example2.nodeID;
340         App.Example.NMStart = Example2.NMStart;
341         App.Example.NMPeriod = Example2.NMPeriod;
342         App.Example.NMSleepTime = Example2.NMSleepTime;
343         App.Example.NMKeepAlive = Example2.NMKeepAlive;
344         App.Example.SleepTime = Example2.SleepTime;
345         App.Example.KeepAlive = Example2.KeepAlive;
346         App.Example.threshold = Example2.threshold;
347 
348     }
```

Figura 8.31: Acción que se ejecuta ante un evento en el botón jButton2_2.

```
394     private void jCheckBox1ActionPerformed(java.awt.event.ActionEvent evt) {
395         // TODO add your handling code here:
396         if(jCheckBox1.isSelected()){
397         Example2.asignarID = "si";
398         }else{
399         Example2.asignarID = "no";
400         }
401     }
```

Figura 8.32: Acción ante un evento en el elemento jCheckBox1.

Por último, en la Figura 8.34 se adjunta la función *main()* del archivo `ConfigurationPanel.java` en la que se hace visible la interfaz.

```
    public static void main(String args[]) throws Exception {
415     /* Set the Nimbus look and feel */
416     Look and feel setting code (optional)
437
438     /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
441             new ConfigurationPanel().setVisible(true);
442         }
443     });
444 }
```

Figura 8.33: Función main() del archivo ConfigurationPanel.java.

Capítulo 9. Validación de la plataforma HW/SW

Con el fin de validar la integración HW/SW desarrollada en el presente TFG, se realizaron varias pruebas experimentales de funcionamiento. Estas pruebas de campo han tenido lugar en el aparcamiento que se ubica enfrente del pabellón A del Edificio de Electrónica y Telecomunicación de la ULPGC, señalado en la Figura 9.1. La prueba definitiva y que se adjunta como vídeo en el presente TFG, se realizó exactamente donde indica la cruz representada en la Figura 9.1.

En la prueba de validación se hace uso de la plataforma completa de la solución desarrollada en el presente TFG. Así, en primer lugar, se ubica físicamente el nodo final, que se fija en una posición de la plaza de estacionamiento. El nodo final se comunicará mediante tecnología LoRa/LoRaWAN con el *nanogateway*, que a su vez se comunica con el servidor de la plataforma TTN vía *WiFi*. En esta prueba, el *nanogateway* se encuentra por comodidad a unos pocos metros del nodo final. Haciendo uso también de conexión *WiFi*, se realiza la transferencia de información entre la plataforma TTN y la aplicación *Java*. La conexión *WiFi* que se ha usado durante la validación de la aplicación desarrollada ha sido la propia conexión *WiFi* que ofrece la universidad, cuyo SSID es `ULPGC´ y carece de contraseña.

Figura 9.1: Aparcamiento utilizado en la prueba de validación.

El procedimiento seguido para realizar la prueba de funcionamiento final comienza por la preparación del proceso experimental. En esta preparación se realizan las siguientes tareas:

- Colocación del nodo final en la plaza de estacionamiento. Se posiciona el nodo final en el suelo a una distancia aproximadamente equidistante de las marcas que delimitan la plaza. El nodo final se alimenta mediante una batería y desde que ésta se encuentra conectada a la placa *Expansion Board* del dispositivo LoPy, se inicia el funcionamiento del nodo final.

- Inicio de funcionamiento del *nanogateway*. El dispositivo que actúa como *nanogateway* se conecta vía USB a un ordenador portátil con el fin de alimentarlo.

- Inicio de la aplicación *Java* en el ordenador portátil. Se inicia la aplicación, mostrándose la interfaz que genera el archivo `VentanaP.java`, y al pulsar en la opción de esta interfaz que indica 'Estado', se abre la interfaz asociada al archivo `StatusPanel.java`. Cuando aparece esta ventana

se debe pulsar el botón 'Iniciar' para que comiencen a mostrarse en la pantalla los datos recibidos desde la plataforma TTN.

Con este procedimiento de preparación realizado se puede dar comienzo a la prueba de validación.

Inicialmente, la plaza de estacionamiento se encuentra desocupada, por lo que en la aplicación *Java* debe reflejarse esta situación. En la Figura 9.2 se muestra el panel verde que indica que el estado de la plaza es 'Libre'.

Figura 9.2: Estado inicial de la plaza y de la aplicación Java.

Seguidamente, al aparcar un vehículo en la plaza, el nodo final transmitirá el cambio de estado, y en la aplicación *Java* se muestra el mismo panel de la Figura 9.2, pero esta vez

en rojo, indicando que la plaza está ocupada. Para que se produzca esta situación, el nodo final envía una trama de tipo *Info Frame* al *gateway*, y este a su vez retransmite la trama a la plataforma TTN, donde se decodifica y se envía la información a la aplicación *Java*, que interpreta la información recibida y modifica el panel según el contenido de dicha información. El resultado se visualiza en la Figura 9.3. Además, la modificación de los campos temporales de *Horas transcurridas* y *Minutos transcurridos* se actualizan del mismo modo, pero a través de las tramas de tipo *Keep-Alive*. La frecuencia con la que se reciben este tipo de tramas depende de los parámetros de configuración del nodo final. Para la prueba de validación se utilizaron tiempos de *Sleep* de 10 segundos y de *Keep-Alive* de 30 segundos.

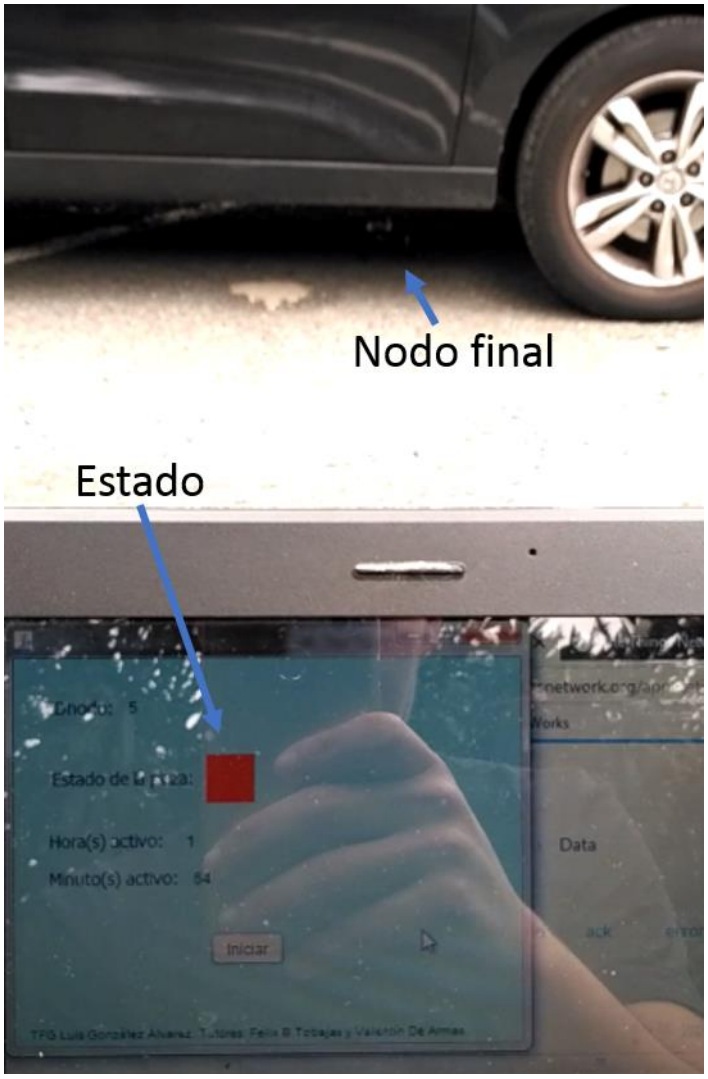


Figura 9.3: Estado de la plaza y de la aplicación Java con un vehículo en la plaza de estacionamiento.

La frecuencia utilizada en el envío de tramas para esta prueba es elevada para agilizar el proceso, pero los parámetros recomendados se indican en apartados anteriores.

Al retirarse el vehículo de la plaza de estacionamiento, se actualiza el panel de la aplicación *Java* y pasa a tener color verde de nuevo, indicando que la plaza vuelve al estado "Libre", tal y como se muestra en la Figura 9.4.

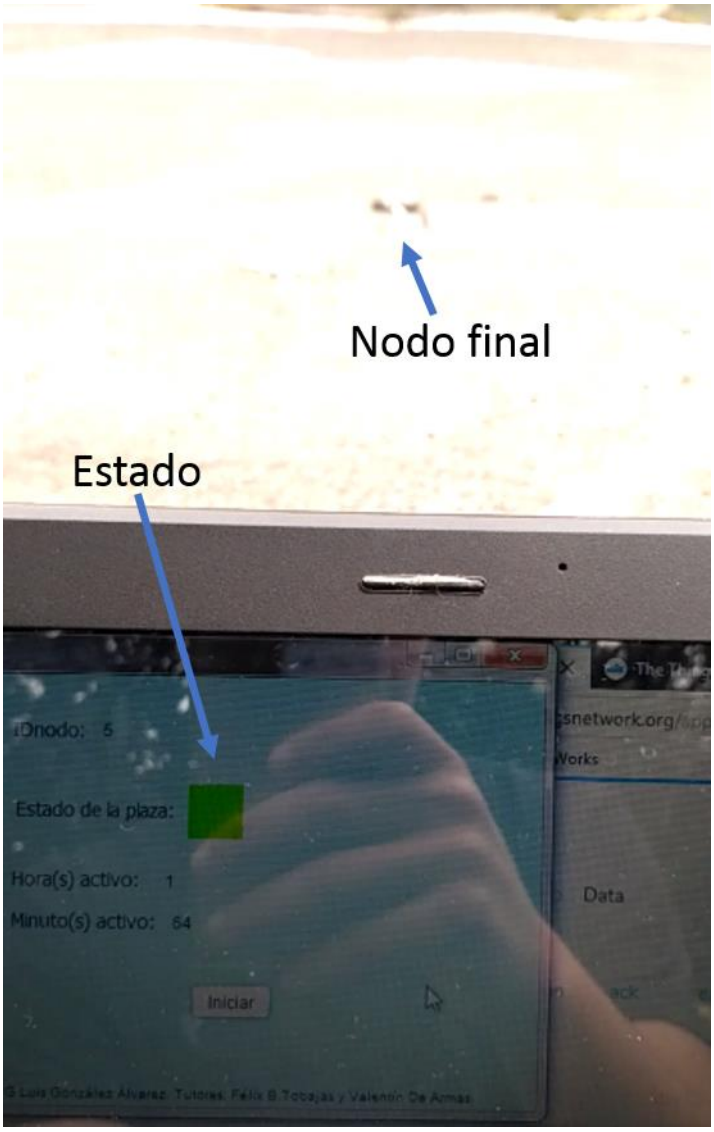


Figura 9.4: Estado final de la plaza y de la aplicación Java al retirarse el vehículo de la plaza de estacionamiento.

Con esta comprobación del estado de la plaza finaliza la prueba de campo realizada. En esta prueba se ha podido validar el correcto funcionamiento de la plataforma HW/SW completa, desde el nodo final hasta la aplicación *Java*, pasando por el *nanogateway* y la plataforma TTN.

También se ha realizado una prueba experimental en el sentido descendente de la comunicación. De esta manera, desde la aplicación *Java* se ha determinado una configuración del nodo y se ha comprobado que el nodo se ha configurado con los parámetros indicados.

En primer lugar, se inicia el funcionamiento del nodo final y del *nanogateway*. También se abre la pestaña *Data* de la aplicación creada en TTN para visualizar las tramas que envía el nodo final y poder comprobar los cambios en los parámetros tras realizar la configuración. En la Figura 9.5 se adjunta la captura de TTN antes de enviar la trama de configuración desde la aplicación *Java*. En esta captura se puede observar que el ID del nodo final es 5. Se destaca este campo sobre los demás porque es el más simple de visualizar y, de esta manera, demostrar de manera sencilla e intuitiva el funcionamiento de la comunicación descendente.

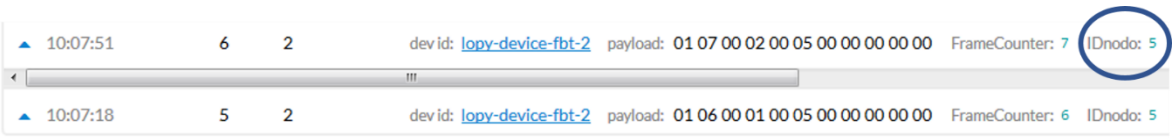


Figura 9.5: Tramas previas a la configuración del nodo final.

Seguidamente, se realiza la configuración deseada desde la interfaz que crea el archivo `ConfigurationPanel.java`. En esta prueba concreta se introdujeron los valores que se muestran en la Figura 9.6.

Al pulsar el botón "Send" de la interfaz de la Figura 9.6, se envía la trama de configuración a TTN. Cuando TTN recibe esta trama, muestra en la parte izquierda de esta una flecha con sentido descendente en color gris como la que se puede ver en la Figura 9.7. En este momento, TTN está esperando a que se reciba alguna trama desde el nodo final para poder enviar la trama de configuración que se encuentra a la espera de la apertura de una ventana de recepción por parte del nodo final. Cuando envía la trama al nodo final, se visualiza una trama con la flecha descendente de color azul, tal y como se observa en la Figura 9.7.

Cuando ha llegado la trama de configuración al nodo, este se configura con los nuevos parámetros y sigue su funcionamiento normal. Uno de los parámetros configurados ha sido el ID del nodo final, que se determinó al valor 255. En la Figura 9.8 se puede observar

cómo el ID del nodo ya ha cambiado en las tramas que este envía y que se visualizan en TTN.

Con la finalización de esta prueba experimental se concluye la fase de validación de la plataforma HW/SW desarrollada.

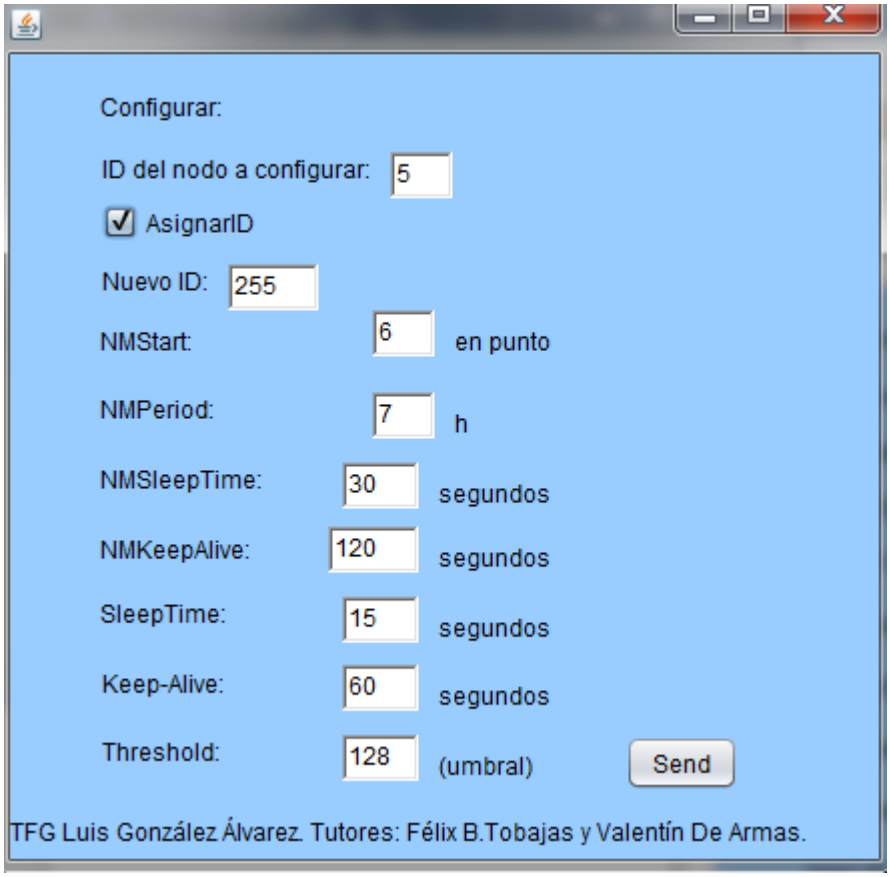


Figura 9.6: Configuración realizada en App.java.

Figura 9.7: Tramas Downlink en TTN.

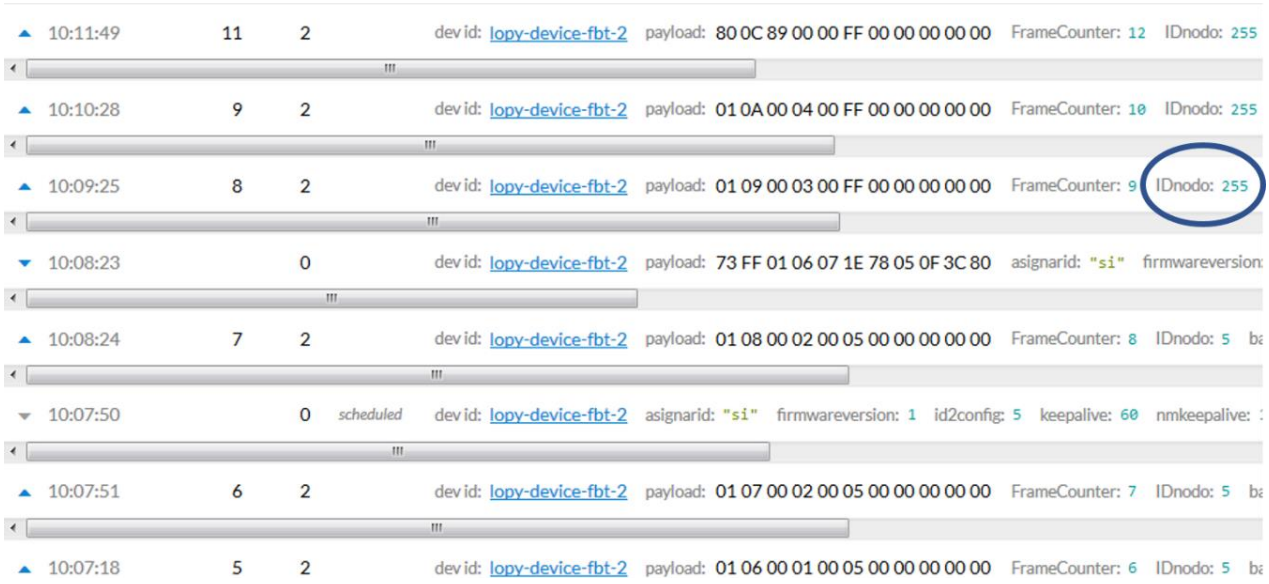


Figura 9.8: Nodo final configurado.

Capítulo 10. Conclusiones y líneas futuras

10.1 Conclusiones

Tras validar la plataforma HW/SW desarrollada, se puede afirmar que se han cumplido los objetivos establecidos en el presente Trabajo Fin de Grado. Se ha estudiado y aplicado a un caso práctico la tecnología de comunicación LoRa/LoRaWAN. Otro objetivo cumplido ha sido el de la caracterización de los sensores y los dispositivos utilizados. También se ha desarrollado el código necesario para cumplir con las funcionalidades establecidas en el nodo final y en el *nanogateway*. Además, se ha configurado la plataforma TTN para la aplicación objetivo de este TFG, así como la integración HTTP. Por último, se ha desarrollado una aplicación *Java* para mejorar la visualización de la información transmitida/recibida.

En resumen, al haber cumplido todos los objetivos se logra la detección del estado de la plaza de estacionamiento y la visualización de dicho estado en la plataforma TTN, en la integración HTTP, y en la aplicación *Java*.

Con el desarrollo de esta solución se logran grandes beneficios, tanto sociales como económicos. Una aplicación *Smart Parking* como la realizada permite ahorrar tiempo de circulación en busca de aparcamiento a los usuarios, disminuyendo de esta manera su

estrés y nerviosismo al volante. Como consecuencia de todo esto se reduce el número de estacionamientos ilegales y la congestión en las carreteras. La disminución del tiempo de circulación de los vehículos también produce un beneficio medioambiental, pues se reduce la emisión de gases contaminantes desde los vehículos. En definitiva, los beneficios de una solución como la desarrollada son múltiples.

La plataforma HW/SW diseñada cuenta con varias propiedades y características que ofrecen al usuario una solución completa, inalámbrica y configurable. El desarrollo de la solución se ha realizado en todos los elementos de la arquitectura, caracterizando cada elemento que interviene en ella, desde el nodo final hasta la aplicación *Java*. El hecho de conocer cada elemento permite mejorar la solución sin generar dependencias entre diferentes bloques de la plataforma HW/SW.

Además, la solución desarrollada es inalámbrica de largo alcance (al estar basada en la tecnología LoRa/LoRaWAN). Esto permite adoptar la propuesta en diferentes situaciones como, por ejemplo, en grandes zonas habilitadas para el estacionamiento, donde un sistema similar pero cableado supondría un coste y un impacto visual importante. Que la solución sea de este tipo favorece claramente la instalación, y el mantenimiento, además de la disminución del impacto visual.

Una de las características más importantes de la solución propuesta es que es configurable. Es decir, el usuario puede configurar la solución con los parámetros que más se adecuen a las necesidades. No obstante, inicialmente la solución inicia su funcionamiento con parámetros óptimos con el fin de obtener un consumo de potencia bajo.

Otra propiedad característica de la solución propuesta es que es escalable. La aplicación se ha dejado desarrollada para soportar comunicaciones con más de un nodo final simultáneamente. La aplicación *Java* es mononodo, pero escalable, es decir con pocas modificaciones se podría obtener una aplicación multinodo. De hecho, la aplicación *Java* recibe datos de otros nodos finales, aunque filtra la información que no pertenece al primer nodo registrado. Por lo tanto, acondicionando levemente la solución se puede obtener un modelo multimodo.

A continuación, se exponen las conclusiones obtenidas al finalizar la realización del Trabajo Fin de Grado, desde la perspectiva de la experiencia adquirida en su desarrollo:

- El estudio y caracterización de los sensores es un factor clave para el éxito de una solución de este tipo. Se han tenido diversos problemas de comportamiento de los sensores dependiendo del entorno en el que se ubicasen. Por lo tanto, se afirma que es indispensable realizar un estudio de comportamiento de los sensores en la zona exacta donde se va a instalar el nodo final, antes de comenzar a codificar su funcionalidad.

- El hecho de que no existieran librerías de los sensores desarrollados en lenguaje *MicroPython* para su integración con los dispositivos de Pycom ha hecho que se tengan que desarrollar dichas librerías a partir de las especificaciones de los sensores facilitadas por los fabricantes.

- Los dispositivos LoPy de Pycom funcionan perfectamente en aplicaciones IoT como la desarrollada. Además, un aspecto favorable de la selección de estos dispositivos ha sido que cuenta con una gran cantidad de documentación en la página oficial de la empresa de Pycom, así como en su foro.

- El uso de un *Nanogateway* en lugar de un *gateway* completo reduce el coste de la solución significativamente pero limita sus posibilidades de comunicación, y prescinde de utilidades como por ejemplo el ADR de LoRaWAN, que aumentan la eficiencia de la red.

- La plataforma TTN facilita la creación de aplicaciones IoT y permite visualizar cómodamente el flujo de datos de cada *gateway* o aplicación registrada. Además, incorpora una documentación adicional que facilita su uso.

- Se ha realizado la integración HTTP pero se ha visto que el resultado no es muy visual para el usuario, por lo que, terminado su desarrollo, se ha optado por realizar también de la integración de una aplicación *Java*.

- El desarrollo de una aplicación *Java* que se comunique con el servidor de la plataforma TTN no es un proceso inmediato. Supone la realización previa de una serie de acciones que permiten crear inicialmente la aplicación, cuyas acciones dependen del sistema operativo que se utilice y de las versiones de *Maven* y *Java* instaladas. A partir de ahí, con la ayuda de un editor, como ha sido en este caso *Netbeans*, se puede completar la aplicación *Java* sin mayor dificultad.

10.2 Líneas futuras

El trabajo realizado supone la combinación de conceptos relacionados con IoT y *Smart Parking*. A partir de esta combinación se ha ideado una solución que se ha visto reflejada de manera práctica en un prototipo de la solución. No obstante, no deja de ser la aplicación de una serie de conceptos teóricos sobre varios dispositivos, y bajo una perspectiva de investigación básica. Por lo tanto, se puede pensar más adelante en esta aplicación como una plataforma HW/SW real para un *parking* completo con esta tecnología aplicada a todas sus plazas de estacionamiento, y considerándola ya como una posible opción de negocio.

A continuación, se describen las ideas principales que se considera que se pueden desarrollar como continuación de este Trabajo Fin de Grado:

- Actualmente, la plataforma TTN carece de una opción para transmitir un mismo paquete desde una aplicación a todos los dispositivos conectados mediante LoRa/LoRaWAN a un determinado *gateway*. Una posible línea de futuro es el desarrollo de una funcionalidad en TTN que permita esta opción, o bien desarrollar esa opción de forma independiente a la plataforma TTN.

- La aplicación *Java* desarrollada está ideada para recibir la información de un único nodo, de hecho, cuenta con un filtro en el que solo reciben paquetes del primer nodo que envía información. Otra línea de desarrollo puede ser la creación de una aplicación *Java* que muestre el estado de todas las plazas de estacionamiento, es decir que sea una aplicación multinodo. Bajo esta idea se puede desarrollar en profundidad la aplicación, y mostrar un mapa real del *parking* y permitir la visualización del estado de cada plaza simultáneamente.

- El nodo final desarrollado no cuenta con una cubierta de protección adecuada como para desempeñar su función sin peligrar su integridad en condiciones de intemperie. El desarrollo de un sistema de protección para el nodo final que permita mantenerlo en perfecto estado y sin disminuir su funcionalidad, se plantea como otra línea posible de desarrollo.

Bibliografía

[1] E. Martín, “Las ‘smart cities’ tienen que ver con los ciudadanos, no con las tecnologías,” *Expansión-Economía Digital*, 15-Dec-2017.

[2] ONU, “Más de la mitad de la población vive en áreas urbanas y seguirá creciendo | ONU DAES | Naciones Unidas Departamento de Asuntos Económicos y Sociales,” Jul-2014. [Online]. Available: http://www.un.org/es/development/desa/news/population/world-urbanization-prospects-2014.html. [Accessed: 26-May-2018].

[3] C. Martínez and E. Cotelo, “Internet de las Cosas: ‘Magnitud de dispositivos a conectar supera cualquier cosa que conozcamos’, dice experto - Radiomundo En Perspectiva,” *En Perspectiva*, 2016. [Online]. Available: https://www.enperspectiva.net/en-perspectiva-programa/entrevistas/internet-de-las-cosas-magnitud-de-dispositivos-a-conectar-supera-cualquier-cosa-que-conozcamos-dice-experto/. [Accessed: 26-May-2018].

[4] E. Press, “Expertos estiman que en 2020 habrá 50.000 millones de dispositivos conectados a Internet.”

[5] AHK, “Bosch simplifica y automatiza la búsqueda de aparcamiento,” 2017. [Online]. Available: http://www.ahk.es/es/comunicacion/noticias/noticias/artikel/bosch-simplifica-y-automatiza-la-busqueda-de-aparcamiento/?no_cache=1&cHash=2b200b978297673a8a2cbdc9515fc2b7. [Accessed: 26-May-2018].

[6] Orange, “LoRa Device Developer Guide,” p. 42, 2016.

[7] UNDESA, *World Urbanization Prospects*. 2014.

[8] G. Rozas, “Efectos psicosociales , ciudad y calidad de vida P,” *Interv. Psicosoc.*, vol. 11, pp. 229–243, 2002.

[9] M. Bouskela, M. Casseb, S. Bassi, C. Luca De, and M. Facchina, “La ruta hacia las Smart Cities,” p. 148, 2016.

[10] B. Cohen, “The Top 10 Smart Cities On The Planet,” *Co.Design*, 2012. [Online]. Available: https://www.fastcodesign.com/1679127/the-top-10-smart-cities-on-the-planet. [Accessed: 26-May-2018].

[11] LOGITEK, “Smart City áreas: Mobility.” [Online]. Available: http://www.creatingsmartcities.es/ambitossmart-mobility.php. [Accessed: 26-May-2018].

[12] E.B, “El transporte es responsable del 41% de las emisiones contaminantes en Madrid | El Boletin,” *El Boletín*, Jul-2017.

[13] DESIC S.L, “GuaguasLPA - Aplicaciones en Google Play.” 2017.

[14] M. Castro, “Smart Parking: Qué es y Cómo nos Beneficia — SITT,” *SITT*, 2017. [Online]. Available: https://www.sittycia.com/blog/2017/4/28/smart-parking-qu-es-y-cmo-nos-beneficia. [Accessed: 26-May-2018].

[15] Statista, “Internet de las cosas: dispositivos interconectados en el mundo 2020 | Estadística,” 2018. [Online]. Available: https://es.statista.com/estadisticas/638100/internet-de-las-cosas-numero-de-dispositivos-conectados-en-todo-el-mundo--2020/. [Accessed: 26-May-2018].

[16] SmartSantander, “SmartSantander.” [Online]. Available: http://www.smartsantander.eu/. [Accessed: 26-May-2018].

[17] “Noruega prueba farolas inteligentes que se encienden al paso de los vehículos,” *La Vanguardia*, Jan-2018.

[18] ECUBE Labs, “Soluciones inteligentes de gestión de residuos | Ecube Labs.” [Online]. Available: https://www.ecubelabs.com/es/. [Accessed: 26-May-2018].

[19] Smart Parking, “Car Parking Solutions.” [Online]. Available: https://www.smartparking.com/. [Accessed: 26-May-2018].

[20] i+D3, “3SCPark: IoT Smart parking para la ciudad inteligente.” [Online]. Available: https://imasdetres.com/3scpark-smartcity-de-aparcamiento/. [Accessed: 26-May-2018].

[21] eSMARTCITY.es, “Una urbanización inteligente en Líbano que se centra en

garantizar la calidad del aire ESMARTCITY," 2017. [Online]. Available: https://www.esmartcity.es/2017/12/05/urbanizacion-inteligente-libano-se-centra-garantizar-calidad-del-aire. [Accessed: 26-May-2018].

[22] InteliLIGHT, "Street Lighting Remote Management." [Online]. Available: https://intelilight.eu/. [Accessed: 26-May-2018].

[23] B. Cendón, "Las redes más usadas en IoT (Sigfox, LoRa, NB-IoT, LTE Cat M)," 2017. [Online]. Available: http://www.bcendon.com/las-redes-mas-usadas-en-el-iot/. [Accessed: 26-May-2018].

[24] 330ohms, "¿Qué es SigFox y cómo funciona?," 2017. [Online]. Available: http://blog.330ohms.com/2017/05/11/que-es-sigfox-y-como-funciona/. [Accessed: 26-May-2018].

[25] "Home page | LoRa Alliance™." [Online]. Available: https://lora-alliance.org/. [Accessed: 26-May-2018].

[26] LoRa Alliance, "About LoRaWAN." [Online]. Available: https://lora-alliance.org/about-lorawan. [Accessed: 26-May-2018].

[27] Libelium, "LoRa vs LoRaWAN tutorial." [Online]. Available: http://www.libelium.com/development/waspmote/documentation/lora-vs-lorawan/. [Accessed: 26-May-2018].

[28] "LoRaWAN y LoRa." [Online]. Available: http://lorawan.es/. [Accessed: 26-May-2018].

[29] RS Components, "LoRaWAN." [Online]. Available: https://es.rs-online.com/web/generalDisplay.html?id=i%2Flora. [Accessed: 26-May-2018].

[30] Digikey Electronics, "Descripción general de la plataforma LoRa," Jun-2017.

[31] L. Aliance, "LoRaWAN ™ 101, A technical introduction," p. 39, 2017.

[32] J. Lampe and Z. Ianelli, "Introduction to Chirp Spread Spectrum (CSS) Technology," no. November, pp. 1–28, 2003.

[33] Y. A. Hussin, "Chirp-Spread-Spectrum Modulation," pp. 1–38, 2015.

[34] Semtech, “SX1276/77/78/79 Datasheet,” 2016.

[35] A. Augustin, J. Yi, T. Clausen, and W. Townsley, “A Study of LoRa: Long Range & Low Power Networks for the Internet of Things,” *Sensors*, vol. 16, no. 9, p. 1466, Sep. 2016.

[36] S. N, L. M, E. T, K. T, and H. O, “LoRaWAN ™ Specification,” 2015.

[37] I. O. Monfort, “Estudio de la arquitectura y el nivel de desarrollo de la red LoRaWAN y de los dispositivos LoRa.,” 2017.

[38] SparkFun Electronics, “ZX Distance and Gesture Sensor - SEN-13162.” [Online]. Available: https://www.sparkfun.com/products/13162. [Accessed: 26-May-2018].

[39] SparkFun Electronics, “ZX Distance and Gesture Sensor SMD Hookup Guide - learn.sparkfun.com.” [Online]. Available: https://learn.sparkfun.com/tutorials/zx-distance-and-gesture-sensor-smd-hookup-guide?_ga=2.172984547.176227172.1526414079-505081865.1526414079. [Accessed: 26-May-2018].

[40] X. Y. Z. I. Technologies *et al.*, “ZX Sensor,” vol. 20, pp. 1–7.

[41] B. Digitized *et al.*, “3-Axis Magnetic Sensor.”

[42] Seeed Wiki, “Seeeduino LoRaWAN.” [Online]. Available: http://wiki.seeedstudio.com/Seeeduino_LoRAWAN/. [Accessed: 31-May-2018].

[43] Pycom, “Pycom - Next Generation Internet of Things Platform.” [Online]. Available: https://pycom.io/. [Accessed: 26-May-2018].

[44] Pycom, “7.1.3 LoPy · Pycom Documentation.” [Online]. Available: https://docs.pycom.io/chapter/datasheets/development/lopy.html. [Accessed: 26-May-2018].

[45] Sigfox, “Sigfox - The Global Communications Service Provider for the Internet of Things (IoT).” [Online]. Available: https://www.sigfox.com/en. [Accessed: 26-May-2018].

[46] RS-Online, “LoPy | LoPy IoT LoRa WiFi BLE Development Board | Pycom.” [Online].

Available: https://es.rs-online.com/web/p/kits-de-desarrollo-de-radio-frecuencia/1259532/. [Accessed: 26-May-2018].

[47] Pycom, “Expansion Boards & Shields.” [Online]. Available: https://pycom.io/solutions/expansion-boards-shields/. [Accessed: 26-May-2018].

[48] Batteries4pro, “Batería de 3.7V 180mAh Li - Po para auriculares Plantronics CS60 y C65.” [Online]. Available: https://www.batteries4pro.com/es/comunicacion/Telefonía/3,3992-batería-de-37v-180mah-li-po-para-auriculares-plantronics-cs60-y-c65-4894128110941.html. [Accessed: 26-May-2018].

[49] GitHub, “Atom.” .

[50] GitHub, “Getting Started : Why Atom.” [Online]. Available: https://atom.io/docs/latest/getting-started-why-atom#the-native-web. [Accessed: 26-May-2018].

[51] GitHub, “Pymakr Atom Package.” .

[52] Pycom, “10.3 REPL vs Scripts · Pycom Documentation.” [Online]. Available: https://docs.pycom.io/chapter/docnotes/replscript.html. [Accessed: 26-May-2018].

[53] The Things Network, “The future of single-channel gateways - Gateways / Single Channel Gateway,” 2017. [Online]. Available: https://www.thethingsnetwork.org/forum/t/the-future-of-single-channel-gateways/6590/2. [Accessed: 26-May-2018].

[54] Pycom, “5.3.6 LoRaWAN Nano-Gateway · Pycom Documentation.” [Online]. Available: https://docs.pycom.io/chapter/tutorials/lora/lorawan-nano-gateway.html. [Accessed: 26-May-2018].

[55] GitHub, “Pycom-libraries,” 2018. [Online]. Available: https://github.com/pycom/pycom-libraries/tree/master/examples/lorawan-nano-gateway. [Accessed: 26-May-2018].

[56] TTN, “The Things Network.” [Online]. Available:

https://www.thethingsnetwork.org/. [Accessed: 26-May-2018].

[57] J. Lindsay, “RequestBin — Collect, inspect and debug HTTP requests and webhooks,” 2013. [Online]. Available: http://requestbin.fullcontact.com/. [Accessed: 26-May-2018].

[58] ConfusedByCode, “cURL for Windows: a Windows Installer for the Web Transfer Tool.” [Online]. Available: http://www.confusedbycode.com/curl/#downloads. [Accessed: 26-May-2018].

[59] TTN, “Quick Start.” [Online]. Available: https://www.thethingsnetwork.org/docs/applications/java/quick-start.html. [Accessed: 26-May-2018].

[60] “java.com: Java y Tú.” [Online]. Available: https://www.java.com/es/. [Accessed: 26-May-2018].

[61] Apache, “Maven – Welcome to Apache Maven.” [Online]. Available: http://maven.apache.org/. [Accessed: 26-May-2018].

[62] Mkyong, “How to install Maven on Windows.” [Online]. Available: https://www.mkyong.com/maven/how-to-install-maven-in-windows/. [Accessed: 26-May-2018].

[63] Apache, “Maven – Download Apache Maven.” [Online]. Available: http://maven.apache.org/download.cgi. [Accessed: 26-May-2018].

[64] GitHub, “Java-app-sdk,” 2017. [Online]. Available: https://github.com/TheThingsNetwork/java-app-sdk/tree/master/samples/samples-mqtt. [Accessed: 26-May-2018].

[65] Oracle, “Welcome to NetBeans.” [Online]. Available: https://netbeans.org/. [Accessed: 26-May-2018].

Pliego de condiciones

El Pliego de condiciones expone las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. A continuación, se muestra el conjunto de herramientas *hardware*, *software* y *firmware* empleadas durante su realización.

PC.1 Condiciones Hardware

En la Tabla PC.P-1 se recogen los equipos y dispositivos *hardware* utilizados con sus principales características.

Equipo/Dispositivo	Modelo	Fabricante/Comerciante
Ordenador	-CPU *Intel* Core i3 a 2.2 GHz -4 GB RAM -320 GB HDD -2 puertos USB	Asus
LoPy	-LoPy v1.0	Pycom
Seeeduino LoRaWAN	Seeeduino LoRaWAN v1.0	Seeedstudio
Sensor magnetómetro	QMC5883L	*QST Corporation*
Sensor de proximidad	*ZX Distance and Gesture*	*XYZ Interactive Technologies/SparkFun Electronics Inc*
Batería LiPo	-BAT573 -3.7V -180mAh -0.7Wh	Electronic NIMO
Expansion Board	V2.0	Pycom

Tabla PC. 1: Condiciones Hardware.

PC.2 Condiciones Software

En la Tabla PC.P-2 se recogen las aplicaciones *software* utilizadas, con su versión correspondiente.

Aplicación	Versión	Desarrollador/Comerciante
Sistema operativo	Microsoft Windows 7 Home Premium 64-bit Edition	Microsoft
Microsoft Office	2016	Microsoft
Microsoft Visio	2016	Microsoft
Microsoft Project	2016	Microsoft
Atom	v1.27.1	GitHub Inc.
Plataforma TTN	-	The Things Network
RequestBin	-	Runscope Inc.
Netbeans IDE	v8.1	Oracle
Java	v8.0.40.25	Oracle
Maven	v3.5.3	Apache
cURL	v7.59	cURL Project
Arduino IDE	v1.8.5	Arduino LLC
PuTTY	v0.70	PuTTY project
Mozilla Firefox	v60.0.1	Moz://a
Adobe Reader	v11.0.21.18	Adobe Systems Software Ireland Ltd.

Tabla PC. 2: Condiciones Software.

PC.3 Condiciones Firmware

En la Tabla PC.P-3 se muestra el *firmware* utilizado y su versión.

Firmware	Versión
LoPy	v1.17.3.b1
Seeeduino	v2.0.10

Tabla PC. 3: Condiciones Firmware.

Presupuesto

Este capítulo presenta el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. Dicho presupuesto se divide en las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividida a su vez en:
  - Amortización del material *hardware*.
  - Amortización del material *software*.
- Redacción de la documentación.
- Derechos de visado del COITT (Colegio Oficial de Ingenieros Técnicos de Telecomunicación).
- Gastos de tramitación y envío.
- Material fungible.

Finalmente, cuando se tengan analizados todos los puntos que forman el presupuesto, se aplicarán los impuestos vigentes y se procederá a la obtención del coste total del TFG.

P.1 Trabajo tarifado por tiempo empleado

Este concepto contabiliza los gastos que corresponden a la mano de obra, según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Para el cálculo de este coste se utiliza la fórmula de la Ecuación P.1:

$$H = C_t \cdot 74{,}88 \cdot H_n + C_t \cdot 96{,}72 \cdot H_e \quad \text{(P.1)}$$

Donde:

- H representa los honorarios totales por el tiempo dedicado.
- H_n detalla el número de horas normales trabajadas dentro de la jornada laboral.
- C_t es un factor de corrección en función del número de horas trabajadas.
- H_e detalla el número de horas especiales trabajadas.

Para la realización del presente TFG, se han invertido un total de 300 horas. Todas ellas se han realizado dentro del horario normal, por lo que el número de horas especiales es cero. Además, de acuerdo con lo establecido por el COITT, el factor de corrección C_t que se aplica para 300 horas trabajadas es de 0,60, tal y como se puede comprobar en la Tabla P.1.

Horas	Factor de corrección
Hasta 36	1,00
Exceso de 36 hasta 72	0,90
Exceso de 72 hasta 108	0,80
Exceso de 108 hasta 144	0,70
Exceso de 144 hasta 180	0,65
Exceso de 180 hasta 360	0,60

Tabla P. 1: Coeficientes reductores para trabajo tarifado según el COITT.

Por lo tanto, en vista de la situación del TFG desarrollado, el coste total de honorarios se calcula como se muestra en la Ecuación P.2:

$$H = 0{,}6 \cdot 74{,}88 \cdot\ 300 +\ 0{,}6 \cdot 96{,}72 \cdot\ 0 = 13.478{,}40\ € \qquad \text{(P.2)}$$

El trabajo tarifado por tiempo empleado asciende a la cantidad de *trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos*.

P.2 Amortización del inmovilizado material

Se entiende como inmovilizado material aquellos recursos *hardware* y *software* empleados para la realización de este TFG.

Para el cálculo del coste de amortización en un periodo de 3 años se utiliza un sistema de amortización lineal, en el que se supone que el inmovilizado material se desprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula como se muestra en la Ecuación P.3.

$$Cuota\ anual = \frac{Valor\ de\ adquisición - Valor\ residual}{Número\ de\ años\ de\ vida\ útil} \qquad (P.3)$$

El sustraendo *Valor residual* se corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil.

P.2.1 Amortización del material *hardware*

La duración de este Trabajo Fin de Grado es de 4 meses, es decir, un periodo muy inferior al de 3 años que se estipula para el coste de amortización. Por esta razón se calculan los costes sobre la base de los derivados de los primeros 4 meses.

En la Tabla P.2 se recogen los elementos *hardware* amortizables necesarios para la realización del trabajo, indicando su valor de adquisición y su amortización, teniendo en cuenta un tiempo de uso de 4 meses.

Equipo/Dispositivo	Valor de adquisición	Amortización
Ordenador Asus X54H	439,00 €	48,79 €
LoPy (x2)	69,90 €	69,90 €
Seeeduino LoRaWAN	66,66 €	66,66 €
Sensor magnetómetro	2,09 €	2,09 €

Sensor de proximidad	21,32 €	21,32 €
Batería LiPo	8,10 €	8,10 €
Expansion Board	16,00 €	16,00 €
Total	623,07 €	**232,86 €**

Tabla P. 2: Amortización del material hardware.

Debido al bajo precio de todos los elementos, exceptuando el ordenador portátil, la amortización coincide con su valor de adquisición en todos ellos. La amortización del ordenador portátil se ha calculado usando el sistema de amortización lineal explicado anteriormente.

El coste total del material *hardware* asciende a *doscientos treinta y dos euros con ochenta y seis céntimos*.

P.2.2 Amortización del material *software*

Para realizar el cálculo de los costes de amortización del material *software* se consideran, al igual que con el material *hardware*, los costes derivados de los primeros 4 meses.

La Tabla P.3 muestra los elementos *software* necesarios para la realización del trabajo, así como su valor de adquisición y su amortización.

Aplicación	Valor de adquisición	Amortización
Sistema operativo Windows 7 Home Premium	0,00 € (*)	0,00 € (*)
Microsoft Office	0,00 € (*)	0,00 € (*)
Microsoft Visio	0,00 € (*)	0,00 € (*)
Microsoft Project	0,00 € (*)	0,00 € (*)
Atom	0,00 € (**)	0,00 € (**)
Plataforma TTN	0,00 € (**)	0,00 € (**)
RequestBin	0,00 € (**)	0,00 € (**)

Netbeans IDE	0,00 € (**)	0,00 € (**)
Java	0,00 € (**)	0,00 € (**)
Maven	0,00 € (**)	0,00 € (**)
cURL	0,00 € (**)	0,00 € (**)
Arduino IDE	0,00 € (**)	0,00 € (**)
PuTTY	0,00 € (**)	0,00 € (**)
Mozilla Firefox	0,00 € (**)	0,00 € (**)
Adobe Reader	0,00 € (**)	0,00 € (**)
Total	0,00 €	**0,00 €**

Tabla P. 3: Amortización del software.

(*) Licencia de uso proporcionada por la ULPGC.

(**) Software de acceso libre.

El coste total del material software es de *cero* euros.

Si se suman los costes del inmovilizado material *hardware* y *software* se obtiene el coste total de inmovilizado material. Este cálculo se muestra en la Tabla P.4.

Concepto	Coste
Material hardware	232,86 €
Material Software	0,00 €
Total	**232,86 €**

Tabla P. 4: Amortización del inmovilizado material.

Por lo que el coste total del inmovilizado material, tanto *hardware* como *software*, es de *doscientos treinta y dos euros con ochenta y seis céntimos*.

P.3 Redacción del trabajo

Utilizando la Ecuación P.4 se determina el coste asociado a la redacción de la memoria del presente Trabajo Fin de Grado.

$$R = 0{,}07 \cdot P \cdot C_n \qquad \text{(P.4)}$$

Donde:

- R son los honorarios por la redacción del trabajo.
- P es el presupuesto.
- C_n es el coeficiente de ponderación en función del presupuesto.

Para obtener el valor del presupuesto se suman los costes del trabajo tarifado por tiempo empleado y la amortización del inmovilizado material. En la Tabla P.5 se muestra este cálculo de presupuesto.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	232,86 €
Total	**13.711,26 €**

Tabla P. 5: Presupuesto, incluyendo trabajo tarifado y amortización del inmovilizado material.

Por otra parte, el coeficiente de ponderación C_n para presupuestos inferiores a 30.050,00€ tiene un valor de 1,00, según el COITT. Por lo que el coste derivado de la redacción del Trabajo Fin de Grado se calcula en la Ecuación P.5:

$$R = 0{,}07 \cdot 13.711{,}26 \cdot 1 = 959{,}79\ € \qquad \text{(P.5)}$$

El coste de la redacción del trabajo asciende a *novecientos cincuenta y nueve euros con setenta y nueve céntimos*.

P.4 Derechos de visado del COITT

El COITT establece que, para proyectos técnicos de carácter general, los derechos de visado para el año 2018 se calculan sobre la base de la Ecuación P.6.

$$V = 0{,}006 \cdot P_1 \cdot C_1 + 0{,}003 \cdot P_2 \cdot C_2 \qquad \text{(P.6)}$$

Donde:

- V es el coste de visado del trabajo.
- P_1 es el presupuesto del proyecto.
- C_1 es el coeficiente reductor en función del presupuesto.
- P_2 es el presupuesto de ejecución material correspondiente a la obra civil.
- C_2 es el coeficiente reductor en función del presupuesto de ejecución material correspondiente a la obra civil.

En la Tabla P.6 se muestra el presupuesto del proyecto, que se obtiene a partir de la suma de las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del trabajo. Por otra parte, el coeficiente C_1 para proyectos de presupuesto inferior a 30.050,00€ es de 1,00, y el valor de P_2 es de 0,00€, ya que no se realiza ninguna obra civil. Por esta misma razón no se aplica el coeficiente C_2.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	232,86 €
Redacción del trabajo	959,79 €
Total	**14.671,05 €**

Tabla P. 6: Presupuesto, incluyendo trabajo tarifado, amortización y redacción del trabajo.

De esta forma, el cálculo del coste por derechos de visado del presupuesto se realiza en la Ecuación P.7.

$$V = 0{,}006 \cdot \ 14.671{,}05 \cdot \ 1 + 0{,}003 \cdot 0 \cdot \ C_2 = 88{,}03 \ € \qquad \text{(P.7)}$$

Por lo tanto, el coste por derechos de visado del presupuesto asciende a *ochenta y ocho euros con tres céntimos*.

P.5 Gastos de tramitación y envío

Los gastos de tramitación y envío ascienden a *seis euros* (6,00€) por cada documento visado de forma telemática.

P.6 Material fungible

Durante el desarrollo de este Trabajo Fin de Grado se han empleado otros materiales a parte de los recursos *hardware* y *software* ya comentados. El material adicional se documenta como material fungible. En la Tabla P.7 se muestran los costes derivados de estos recursos.

Concepto	Coste
Folios	10,00 €
Tóner de la impresora	30,00 €
Encuadernado	5,00 €
Tres CD-ROM	6,00 €
Total	**51,00 €**

Tabla P. 7: Coste de material fungible.

El coste del material fungible asciende a *cincuenta y un euros*.

P.7 Aplicación de impuestos y coste total

La realización del presente TFG está gravada por el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7%). Teniendo en cuenta la aplicación de los impuestos se realiza el cálculo del presupuesto total del Trabajo Fin de Grado. Este cálculo se muestra en la Tabla P-8.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	232,86 €
Redacción del trabajo	959,79 €
Derechos de visado del COITT	88,03 €
Gastos de tramitación y envío	6,00 €
Costes de material fungible	51,00 €
Total (Sin IGIC)	**14.816,08 €**
IGIC (7%)	1.037,13 €
Total	**15.853,21 €**

Tabla P. 8: Presupuesto total del Trabajo Fin de Grado.

El presupuesto total del Trabajo Fin de Grado "*Desarrollo de una plataforma HW/SW de Smart Parking basada en tecnología LoRa/LoRaWAN*" asciende a *quince mil ochocientos cincuenta y tres euros con veintiún céntimos*.

Fdo.: D. Luis González Álvarez

En Las Palmas de Gran Canaria a 8 de junio de 2018

Anexo

Anexo A. Contenido del CD-ROM.

Adjunto a la memoria del presente Trabajo Fin de Grado se encuentra disponible un *Compact Disc Read-Only Memory* (CD-ROM). En este anexo se describe la estructura de su contenido a partir de la lista que se expone a continuación.

- En el directorio `Memoria del TFG` se encuentra la memoria del TFG "Desarrollo de una plataforma HW/SW de Smart Parking basada en tecnología LoRa/LoRaWAN" en lengua española y en formato PDF.

- En el directorio `Código` se encuentran cuatro subdirectorios:

  - En el subdirectorio `Nodo Final` se incluyen los archivos que contienen la codificación del elemento nodo final de la solución desarrollada. Estos son los archivos `pymakr.conf` y `main.py`.

  - En el subdirectorio `Nanogateway` se incluyen los archivos que contienen la codificación del elemento *nanogateway* de la solución desarrollada. Estos son los archivos `pymakr.conf`, `config.py`, `nanogateway.py` y `main.py`.

  - En el subdirectorio `TTN e Integración HTTP` se incluyen los archivos que contienen la codificación de los elementos *The Things Network* e Integración HTTP de la solución desarrollada. Estos son los archivos `Decoder.txt`, `Encoder.txt` y `HTTPaddressAndDownlinkMessage.txt`.

  - En el subdirectorio `Aplicación Java` se incluyen los archivos que contienen la codificación del elemento Aplicación *Java* de la solución desarrollada. Estos son los archivos `App.java`, `VentanaP.java`, `VentanaP.form`, `StatusPanel.java`,

`StatusPanel.form`, `ConfigurationPanel.java`, `ConfigurationPanel.form` y `pom.xml`.

- En el directorio `Vídeo` se encuentra el vídeo donde se demuestra el funcionamiento de la solución desarrollada.

- En el directorio `Abstract` se encuentra el resumen de la solución desarrollada en lengua inglesa y en formato PDF.