DEPARTAMENTO DE TEORÍA DE LA SEÑAL Y COMUNICACIONES

INGENIERÍA INDUSTRIAL

PROYECTO DE FIN DE CARRERA

LOCALIZACIÓN DE OBJETOS UTILIZANDO TECNOLOGÍA ZIGBEE (PARTE II)

Daniel R. Cañoto López
Tutora: Ana García Armada

Mayo 2014

"*Dadme un punto de apoyo y moveré el mundo*"
- Arquímedes

A Miguel Castán por tener la idea e incluirme en ella, y a Ana García Armada por dirigirla y hacerla posible

A Ramiro, Amor y Aida, por estar a mi lado, incondicionalmente, siempre

A Raquel, por todos estos años (y los venideros)

A Alfonso, por enamorarme de la ingeniería

Y, por último, a los que están desde el principio, y a los que llegaron tarde, por hacer que de esta carrera, más bien maratón, me lleve mucho más que un título

Tablas de contenidos

Índice de capítulos

Índice de ilustraciones

Índice de ecuaciones

Índice de tablas

Resumen

La electrónica es, sin duda, una de las ramas de la tecnología que más ha avanzado en los últimos años.

Circuitos integrados cada vez más económicos y sencillos de fabricar junto con nuevas líneas de investigación hacen que el desarrollo de proyectos en esta rama de la ingeniería sea algo fácil y lleno de recursos y posibilidades.

Este proyecto se centra en el estudio de las WSN (*Wireless Sensor Networks* [1], Redes Inalámbricas de Sensores), y cómo utilizarlas para resolver un problema concreto.

Típicamente éstas son redes centralizadas en las que una estación base o nodo central recoge información de su entorno a través de nodos sensores.

A pesar de que el objetivo de este proyecto no es monitorizar el entorno, tras un estudio y comparación con otras tecnologías de comunicación inalámbrica (redes Bluetooth o Wifi…) encontramos que las WSN son las que más se adecúan a nuestras necesidades. Éste tema se trata con más profundidad en la primera parte del proyecto [2]

Este proyecto se divide en dos partes, la primera parte, realizada por Miguel Castán, presenta todo el estudio teórico relacionado con las WSN comparando las motas y los protocolos de comunicación existentes, para poder elegir los más convenientes, y terminando con un enfoque práctico diseñando un protocolo que se adecúe a las necesidades específicas del proyecto.

Esta segunda parte trata todo lo relativo a la programación y la electrónica de las motas utilizadas; estudio y comparación de los diferentes entornos de desarrollo disponibles para este tipo de aplicaciones, y, finalmente, el planteamiento del problema y su resolución mediante 3 aplicaciones programadas en distintos lenguajes que permiten la interacción usuario-nodo central-nodo sensor.

Abstract

Nowadays electronic technology is one of the most developing technological fields.

Integrated circuits getting cheaper and easier to build make development of projects in this field something easy and full of resources and possibilities.

This project is focused in the study of WSN (*Wireless Sensor Networks* [1]) and how they can be used to solve a given problem.

These are typically centralized networks in which a base station or central node collects information from its environment through sensor nodes

Although the aim of this project is not monitoring the environment, after a deep study and comparison among other similar wireless communication technologies (Bluetooth networks, Wifi) we found that WSN are the ones that best fit to our needs. This topic is deeply discussed in the first part of the project.

The project has been divided in two parts; the first one, headed by Miguel Castán presents the theoretical study related with WSN, comparing different motes and existing communication protocols, to choose the most convenient for us, ending with a practical approach designing a suitable protocol for our needs.

The second part presented here involves all the activities related to programming and electronics for the used motes; study and comparison among the different development environments available for this kind of applications, and, finally, the approach to the problem and its resolution through 3 applications written in different languages that allow the interaction between user, central node, and central sensor.

Introducción

Como ya se ha mencionado, las WSN nacen de un importante desarrollo en la tecnología electrónica, así como un abaratamiento de los circuitos integrados.

En 2003 el *Massachusetts Institute of Technology* publicó un artículo (*10 Emerging Technologies That Will Change The World* [1]*)* en el que se habla de las WSN como una de las tecnologías con más futuro a medio y largo plazo.

El bajo tamaño y consumo de los dispositivos, su alta capacidad de respuesta y su fácil programación hace sencillo y económico realizar proyectos basados en esta tecnología.

Motivación

Un problema tan cotidiano y antiguo como la pérdida de objetos sigue sin tener una solución rápida, económica y fiable. Sistemas basados únicamente en emisión de sonidos o luz tienen poco alcance y autonomía; sistemas basados en Bluetooth, Wifi o GPS dan poca precisión o precisan de una infraestructura para su funcionamiento.

En este proyecto se pretende desarrollar para este problema una solución sencilla, económica y con gran autonomía y fiabilidad.

Objetivo

El objetivo del proyecto es crear una aplicación que sirva de enlace entre el usuario y la WSN compuesta por sus objetos.

Para ello se han utilizado las motas MicaZ del fabricante *Crossbow*, con las que cuenta el Grupo de Comunicaciones de la Universidad Carlos III de Madrid.

Partiendo de esa idea y con los recursos disponibles, se desarrollan 3 aplicaciones: Una de ellas, idéntica para todos los nodos sensores, que se encarga de enviar, recibir y gestionar la comunicación de los nodos sensores; la segunda, una aplicación en el nodo base (o *Base Station*) que es un puente entre la comunicación inalámbrica con los sensores y la comunicación serie con el ordenador. Estas dos aplicaciones se han desarrollado en nesC [3] a través del entorno de desarrollo *TinyOS*

[4]. La tercera, por el contrario, se desarrolla en C y es con la que interacciona el usuario. La idea e implementación de las aplicaciones se detallan más adelante.

Ilustración 1- Esquema interacción Usuario - Estación Base - WSN

Estructura

El documento que se presenta como memoria del proyecto realizado se divide en, además del resumen, introducción y anexos, 4 capítulos.

En el primer capítulo se tratan todas las tecnologías, lenguajes y entornos utilizados: Se presenta la tecnología Zigbee, las familias de motas del fabricante *Crossbow*, haciendo especial hincapié en la familia utilizada en este proyecto (*MicaZ*). Se habla del entorno de desarrollo utilizado (*TinyOS*), del lenguaje de programación propio para estas motas (*nesC*) y de la estructura que debe cumplir una aplicación desarrollara en *nesC* utilizando el entorno *TinyOS* aplicaciones en motas MicaZ.

El segundo capítulo presenta el desarrollo que se ha hecho en el proyecto: Se presentan todas las aplicaciones y funciones desarrolladas íntegramente para el proyecto así como las incluidas en el entorno de desarrollo y se han adaptado para el alcance de este proyecto.

El capítulo 3 detalla las pruebas que se realizan, que tienen como objetivo determinar dos parámetros: distancia entre nodos y tráfico máximo admisible, analizando y modelizando el comportamiento de las motas con las aplicaciones desarrolladas.

Para determinar el primero se hacen estudios y mediciones a distintas distancias y se concluye con un modelo matemático que adapta el nivel de potencia de los mensajes enviados para optimizar energía y mejorar la estimación de la distancia entre nodos.

El segundo nos dará una frecuencia máxima de comunicación admisible para uno o varios nodos en los que el sistema puede trabajar sin problemas de pérdida de comunicación.

Por último, el capítulo 4 presenta las conclusiones, presupuesto del proyecto y futuras mejoras del mismo.

Definiciones y acrónimos

Bluetooth	Protocolo de comunicación inalámbrica de corto alcance. En su última versión, la 4.0, permite hasta 24 Mbit/s de tasa de transmisión.
dBm	Unidad de medida de potencia expresada en decibelios (dB) por milivatio (mW). Habitualmente utilizado en el entorno de las comunicaciones radio y microondas.
Domótica	Conjunto de sistemas capaces de automatizar una vivienda ofreciendo servicios de bienestar, comunicación, seguridad y gestión energética, normalmente integrados con un sistema de comunicaciones que permite su control a distancia.
Estación base	Nodo central de una WSN.
GPS	Sistema de Posicionamiento Global o *Global Positioning System*, es un sistema de posicionamiento mundial vía satélite desarrollado, implantado y actualmente operado por el Departamento de Defensa de los Estados Unidos que permite obtener la posición de cualquier objeto con precisión de metros.
IEEE	Institute of Electrical and Electronic Engineers: asociación técnico-profesional mundial, sin ánimo de lucro, que se dedica a estandarizar, promover la creatividad, desarrollar, compartir y aplicar los avances en las tecnologías de la información electrónica y ciencias en general.
IEEE 802.15.4	Estándar que define el nivel físico y el control de acceso al medio de redes inalámbricas de área personal con bajas tasas de transmisión de datos.
LQI	Indicador de la Calidad del Enlace o *Link Quality Indicator*, es un indicador (de acuerdo con la IEEE 802.15.4) que indica cómo de fuerte es el enlace de comunicación. Se obtiene ponderando el valor RSSI con el máximo y mínimo nivel de potencia definidos para la radio.
Makefile	Es un fichero que contiene las instrucciones necesarias para construir un programa utilizando el comando *make*. Es utilizado habitualmente en desarrollo de software, especialmente cuando el número de archivos a compilar es grande.
Mota	Es como se denomina a un dispositivo que puede formar parte de una WSN. Generalmente un microprocesador con

	una unidad de radio y algún sensor.
nesC	Lenguaje de programación derivado de C en el que se programan las motas que utilizan el entorno TinyOS.
Nodo	Cada uno de los elementos que forman una WSN.
Nodo Central	Elemento central de una WSN.
Powerset	Cada dispositivo que utiliza el estándar Zigbee de comunicación tiene establecidos unos valores máximos y mínimos de potencia RF para transmisión. El valor *powerset* divide ese rango de valores en niveles (en este caso, 31) de potencia que es con lo que trabajan las motas.
RF	Radio Frecuencia hace referencia al rango de frecuencias entre 3 kHz y 300 GHz que corresponde a la frecuencia de las ondas de radio. El término también se utiliza como sinónimo de radio, para referirse a la transmisión de señales eléctricas sin cables.
RSSI	Indicador, en dBm, de la intensidad de señal recibida (*Received Signal Strenght Indicator*).
TinyOS	Entorno de desarrollo utilizado para sistemas de comunicación inalámbricas de baja tasa de transmisión de datos.
Wifi	Tecnología de comunicación inalámbrica de ámbito local orientada principalmente al acceso a Internet a través de un punto de acceso fijo o *hotspot*. Además, Wi-Fi es una marca registrada de la compañía Wi-Fi Alliance que se encarga de probar y certificar que los equipos que cuentan con tecnología Wi-Fi cumplen el estándar IEEE 802.11.
WSN	Red inalámbrica de sensores (*Wireless Sensor Network*)

Tabla 1: Acrónimos y definiciones

Capítulo 1

Tecnologías y entornos utilizados

1.1 Tecnología Zigbee

Zigbee es el nombre que reciben una serie de protocolos basados en el estándar IEEE 802.15.4 pensados para crear redes de comunicación inalámbricas con baja tasa de transferencia de datos, trabajando a distintas frecuencias comprendidas entre 868 MHz y 2,4 GHz [5][6].

Las redes basadas en Zigbee están en auge en la actualidad, especialmente para aplicaciones de domótica, debido a que:

- Presentan un consumo muy bajo ya que pasan la mayor parte del tiempo "durmiendo" (ver tabla 1). Esto las hace una alternativa más que atractiva frente a las redes Bluetooth.
- Son redes en malla, donde cada nodo se comunica con los otros nodos directamente, y además con la estación base, a diferencia de la tecnología Wifi que necesita de un nodo central que gestione las comunicaciones.
- Los nodos se pueden fabricar con muy poca electrónica, y de forma sencilla y económica, lo que hace que puedan ser muy pequeños y funcionales.

	Zigbee	**Bluetooth**
Número máximo de nodos	65535, en subredes de 255 nodos	8
Consumo transmitiendo	30 mA	40 mA
Consumo en reposo	3 µA	200 µA

Tabla 2- Zigbee vs Bluetooth [7]

Dada la cantidad de aplicaciones disponibles que utilizan la tecnología Zigbee, disponen de varios estándares para unificar los productos y hacerlos compatibles entre sí según su mercado objetivo, entre éstos están electrónica de consumo, aplicaciones para domótica y hogar, telecomunicaciones… [8]

La frecuencia de comunicación más utilizada (y la única a nivel mundial) es 2,4 GHz, que admite transferencias hasta a 40 Kbps en 16 canales distintos. También se trabaja a 915 MHz (en América, 20 Kbps en 10 canales) o 868 MHz (en Europa, 20Kbps en un solo canal).

Zigbee es una tecnología líder en su mercado objetivo, ya que es la única tecnología de comunicación inalámbrica enfocada a la monitorización del entorno y muestreo de datos mediante sensores, y que optimiza tanto el consumo y el coste [9].

Aparte de las aplicaciones ya mencionadas y las típicas aplicaciones de tecnología de comunicación sin cables, algunos ejemplos interesantes de aplicaciones y estándares definidos para hardware que emplee la tecnología Zigbee son: sensores de presión sanguínea, micrófonos y altavoces, y timbres o botones de alerta ante emergencias [10].

1.2 Familias de motas Crossbow

Una mota es un dispositivo que puede formar parte de una WSN. Generalmente un microprocesador con una unidad de radio y algún sensor.

De entre los muchos fabricantes del mercado de motas y dispositivos orientados a las WSN que utilicen tecnología Zigbee, centraremos el estudio en el fabricante Crossbow (http://www.xbow.com/) , concretamente en la familia MicaZ, que es la proporcionado por la universidad para el desarrollo de este proyecto.

No obstante, a continuación se hará breve análisis entre las principales familias ofertadas por el fabricante y sus principales características.

1.2.1 Familia Mica2

Esta familia tiene características de tamaño similares a la utilizada en el proyecto, pero presenta un menor consumo, mayor alcance y una menor velocidad de transmisión. Algunas de sus características son:

Placa de comunicación	CC 1000
Rango de frecuencias de trabajo	315 / 433 / 915 [MHz]

Ratio de transmisión máximo	0,6 – 76,8 [Kbps]
Consumo en recepción (*RX Power*)	9,6 [mA]
Consumo en transmisión (*TX Power*)	25 [mA] (5 dBm de salida)
Consumo en *stand-by*	0,2 [μA]
Potencia RF	-20 dBm a 5/10 dBm (0,01 mW a 3,16/10mW)
Alcance (en exteriores)	Hasta 305 m

Tabla 3: Características de la mota Mica2

1.2.2 Familia Mica2dot

Esta mota presenta un tamaño muy reducido.

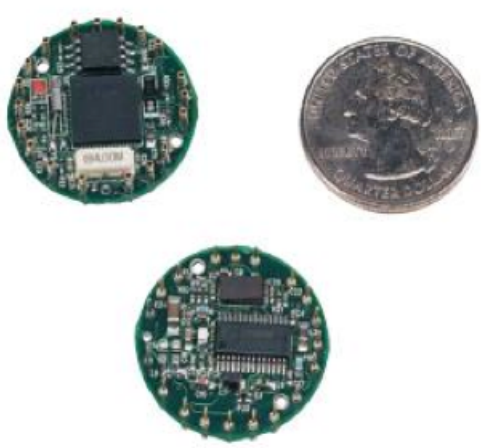

Ilustración 2: Mota Mica2dot

El modelo estándar utiliza una placa de comunicación MPR500CA con un procesador AtMega 128L del fabricante Atmel.

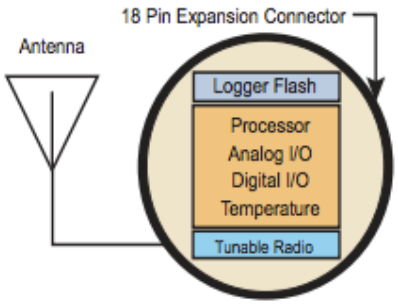

Ilustración 3: Diagrama de bloques de la mota mica2dot

A continuación algunas de sus características principales [16]:

Placa de comunicación	MRP500CA
Rango de frecuencias de trabajo	868-916 [MHz]
Ratio de transmisión máximo	Hasta 393 [Kbps]
Consumo en recepción (*RX Power*)	10 [mA]
Consumo en transmisión (*TX Power*)	27 [mA]
Consumo en *stand-by*	<1 [μA]
Potencia RF	-24 dBm a 0 dBm (0,004 mW a 1mW)
Alcance (en exteriores)	Hasta 152 m
Alcance (en exteriores)	Hasta 152 m

Tabla 4: Características de la familia mica2dot

Esta familia de motas presenta unas características óptimas para el desarrollo de este proyecto tanto en tamaño, ratio de transmisión y alcance. Sería la óptima para la aplicación final de este proyecto de haber estado disponibles.

1.2.3 Familia TelosB

Esta familia la desarrollan conjuntamente Crossbow y la universidad de Berkeley. Esta placa está muy orientada a la interacción con un conjunto grande de sensores, captura y procesamiento de datos (gracias a su unidad Flash externa) y fácil comunicación serie con otros dispositivos.

Ilustración 4: Mota de la familia TelosB

Como se puede observar en la Ilustración 4, carece tanto de antena (lleva una antena embebida en el circuito integrado) como del tradicional conector de 8 pines, que en otras familias hace las veces de conector de expansión para placas sensoras y de interfaz para el programador USB.

En su lugar, lleva un conector USB eliminando la necesidad de utilizar un programador, e incorpora dos conexiones de 6 y 10 pines, como se puede observar en la Ilustración 5.

Esto facilita las conexiones de *suites* de sensores y tanto la programación como la comunicación serie con otros dispositivos, cosa que en otros modelos es más complicada, y requiere desmontar la placa sensora y montar la mota en el programador USB tanto para programar la mota como para la comunicación por puerto serie.

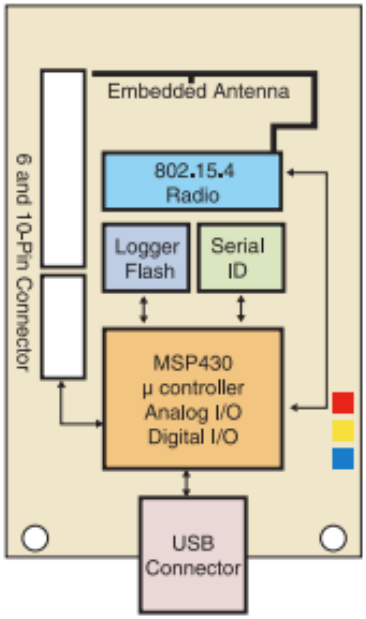

Ilustración 5: Diagrama de bloques de una mota de la familia TelosB

Además, esta familia posee una unidad de memoria extendida, y un procesador MSP430 [17], un procesador de Texas Instruments orientado a aplicaciones de bajo consumo.

En la tabla siguiente se pueden observar las principales características de la familia TelosB [18]:

Placa de comunicación	TPR2400CA
Rango de frecuencias de trabajo	2,4 – 2,4835 GHz
Ratio de transmisión máximo	250 kbps
Consumo en recepción (*RX Power*)	21 [µA]
Consumo en transmisión (*TX Power*)	23 [mA]
Consumo en *stand-by*	1 [µA]
Potencia RF	-24 dBm a 0 dBm (0,004 mW a 1mW)
Alcance	75 – 100 m (exteriores) /

	20-30 m (interiores)
Memoria	10 kB RAM + 1 MB de memoria externa Flash para recolección de datos.

Tabla 5: Características de la familia TelosB

Como se puede observar, también es la placa que menor consumo presenta para transmisión y recepción, de lo que se intuye que está pensada para aplicaciones con una comunicación muy activa.

1.3 Motas MicaZ

Además de todas las analizadas anteriormente, la última de las familias fabricadas por Crossbow es la utilizada en este proyecto, ya que es la proporcionada por la universidad. Es por ello que se le dedica un punto aparte y se hará un análisis más extenso.

Ilustración 6: Mota de la familia MicaZ

Esta familia de motas utiliza una placa de comunicación MPR2400, presenta una antena externa con lo cual se puede cambiar por otra de tamaño mayor para lograr mayor alcance.

Recupera la interfaz de comunicación del conector de 51 pines de expansión al que se pueden conectar tanto placas sensoras como la placa programadora USB para programación o comunicación serie.

Las principales características [19] de esta placa en comparación con las comparadas anteriormente se pueden ver en la tabla

Característica	MicaZ	TelosB	Mica2	Mica2dot
Placa de comunicación	MPR2400	TPR2400CA	CC 1000	MRP500CA
Rango de frecuencias de trabajo	2,4 – 2,4835 GHz	2,4 – 2,4835 GHz	315 / 433 / 915 [MHz]	868-916 [MHz]
Ratio de transmisión máximo	250 kbps	250 kbps	0,6 – 76,8 [Kbps]	Hasta 393 [Kbps]
Consumo en recepción (*RX Power*)	8 [mA]	21 [µA]	9,6 [mA]	10 [mA]
Consumo en transmisión (*TX Power*)	8 [mA]	23 [mA]	25 [mA]	27 [mA]
Consumo en *stand-by*	< 15 [µA]	1 [µA]	0,2 [µA]	<1 [µA]
Potencia RF	-24 dBm a 0 dBm (0,004 a 1mW)	-24 dBm a 0 dBm (0,004 a 1mW)	-20 dBm a 5/10 dBm (0,01 mW a 3,16/10mW)	-24 dBm a 0 dBm (0,004 a 1mW)
Alcance	75 – 100 m (ext.) / 20-30 m (int.)	75 – 100 m (ext.) / 20-30 m (int.)	Hasta 305 m	Hasta 152 m

Tabla 6: Comparativa de características MicaZ, Mica2dot, Mica2, TelosB

Como se aprecia en la tabla, la familia MicaZ presenta características casi idénticas a la mota TelosB, y con algo menos de alcance que la Mica2 y la Mica2dot.

La familia MicaZ cumplen con los estándares de Zigbee, y se compone de un microprocesador, un chip de radio, y periféricos, que suelen ser sensores; todo esto puede estar integrado en una sola placa o bien formado por distintos módulos.

Como se ve en la Ilustración 6, las motas las forma una placa con un microprocesador, un chip de radio, y una interfaz de comunicación; un módulo de baterías y un módulo sensor, que se analizarán más adelante con detalle. Además de eso, se dispone de una placa programadora, que permite tanto la programación de las propias motas como la comunicación serie con un ordenador, a través de USB y dos puertos serie virtuales (uno para programación y otro para comunicación serie).

Ilustración 7: Material disponible

1.3.1 Placa de microprocesador, chip de radio y comunicación

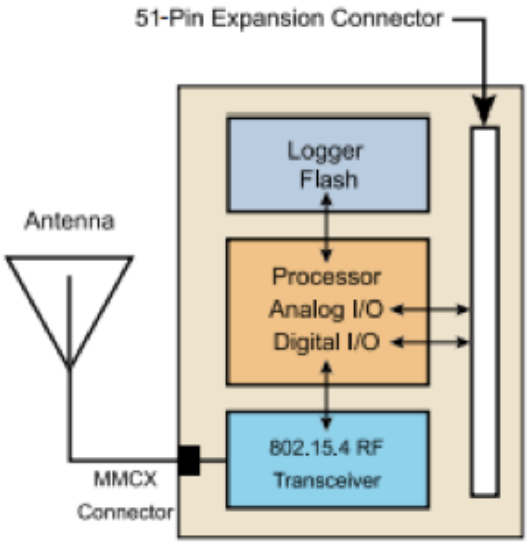

Ilustración 8: Diagrama de bloques de una mota MicaZ

Ilustración 9: Mota MicaZ

Esta placa está formada por:

- Conjunto RBG de diodos led: Se utilizan como leds de estado (lectura/escritura a través del puerto serie) y como salidas digitales en las aplicaciones programadas.

- Antena para la comunicación radio: Se conecta directamente con el chip de radio, y utiliza un conector de tipo *Micro Miniature Connector*, muy utilizado para este tipo de proyectos pues permite la rotación completa de la antena.
- Interfaz de comunicación: conector de 51 pines, tiene varias funciones:
    - Comunicación serie y programación: A través de la placa programadora, se puede tanto descargar código desde un ordenador como establecer una comunicación serie para envío y recepción de comandos.
    - Conexión de periféricos: Las placas compatibles, como la placa sensora descrita más adelante, se pueden conectar a través de este puerto.
    - Entradas y salidas digitales: Los pines se pueden utilizar como entradas o salidas digitales, tanto individualmente como bits, como en grupos de 8 como bytes.
    - Alimentación: A través de este puerto también se puede alimentar la placa, como alternativa a la entrada de alimentación.

- Interruptor de encendido/apagado.
- Conector de alimentación.
- Microprocesador Atmel Atmega 128L con 7 MHz de frecuencia de reloj. Tiene 2 memorias de tipo EEPROM (*Electrically Erasable Programmable Read-Only Memory*) con 4 KB para almacenamiento de datos y 128 KB para guardar programas. Además, la UART conecta este microprocesador con una memoria externa flash de 512 KB.
- Chip CC2420: Desarrollado por Texas Instruments, es un chip utilizado en comunicaciones inalámbricas de bajo consumo.

1.3.2 Módulo de baterías

Con capacidad para dos baterías de tipo AA, incluye las conexiones necesarias con la placa, además de elementos para garantizar su sujeción.

Ilustración 10: Módulo de baterías

1.3.3 Placa sensora

La placa sensora MTS400 incorpora sensores de temperatura, humedad, presión, movimientos sísmicos, e incorpora un altavoz.

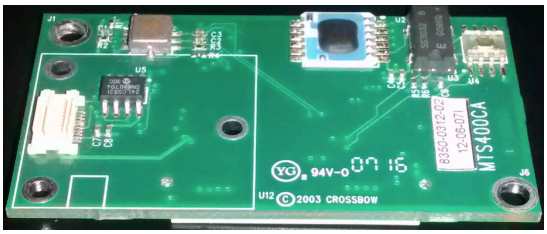

Ilustración 11: Placa sensora

1.3.4 Placa programadora

La placa MIB510 [11] sirve de enlace entre motas de la familia MICAZ y un ordenador. Da soporte tanto a la programación como a la comunicación serie. Como se puede observar, lleva el conector de 51 pines característico de la familia MicaZ

Ilustración 12: Placa programadora MIB520

Cualquier mota puede hacer la función de estación base conectada a un ordenador a través de una placa MIB520. Además de la comunicación serie, permite la

programación de las motas, ya que la MIB520 crea dos puertos serie virtuales: Uno para la programación de las motas y otro para la comunicación serie.

Adicionalmente, al utilizar una conexión USB, las motas se pueden alimentar directamente del ordenador eliminando así la necesidad de alimentación externa.

1.4 TinyOS

TinyOS [12] es un entorno de desarrollo gratuito y *Open Source* (código abierto) creado como un proyecto pensado para sistemas de comunicación inalámbrica de bajo consumo.

El proyecto se crea en la Universidad de Berkeley En colaboración con la agencia DARPA (*Defense Advanced Research Projects Agency*). DARPA es una agencia del departamento de Defensa de los Estados Unidos responsable del desarrollo de la tecnología para uso militar y prevenir desarrollos militares importantes en otros países.

La primera versión de TinyOS se lanza en 1999 y pasa por 19 versiones hasta la última, TinyOS 2.1.2, en Agosto de 2012.

Establece un entorno de desarrollo en el sistema en el que se instala, dotándolo de librerías, comandos, drivers y otras utilidades necesarias para el desarrollo de aplicaciones en este área.

Ilustración 13: Logotipo de TinyOS

TinyOS se puede instalar en Windows (a través de Cygwin [13]), sistemas Linux y OSX, si bien es totalmente recomendable hacerlo en Linux por la cantidad de problemas encontrados al intentar instalar el entorno en Windows o OSX.

Aunque la instalación no es un proceso demasiado complicado, en este proyecto ha supuesto una dificultad importante, por la gran cantidad de problemas encontrados y la falta de un canal de soporte oficial.

1.5 nesC

nesC (*Network Embebbed System C* [3][14]) es una extensión del lenguaje C optimizado para el entorno de desarrollo TinyOS.

Está pensado para trabajar con nodos de redes de sensores, que suelen tener recursos limitados en cuanto a memoria (ver detalles para el nodo utilizado en el apartado 1.2.1.1), y en los que los programas funcionan, principalmente, por eventos.

En nesC los programas no se ejecutan de forma secuencial, sino que se basan en interrupciones y eventos o "manejadores de interrupción".

A continuación se profundizará un poco más en la estructura de los programas escritos en nesC mediante ejemplos sencillos.

1.5.1 Estructura de un programa en nesC

Antes de entender la estructura de un programa escrito en nesC, es necesario definir algunos conceptos:

- Componentes: Son los bloques que configuran el programa. Pueden proveer o utilizar interfaces, y hay dos tipos: configuraciones y módulos.

- Interfaces: Representan la interacción entre dos componentes. Están formadas por comandos, tareas y eventos que pueden ser utilizados por el programa.

- Configuración: Una configuración es un componente que **cablea** otros componentes, conectando las interfaces de un componente con las de otro, permitiendo así que un componente **utilice** las interfaces que otro componente **provee**.

- Módulo: Un módulo es un componente que implementa una o más interfaces, bien por **usarlas** o por **proveerlas**.

Un programa en nesC se compone de dos archivos: Uno que contiene las configuraciones (el cableado entre interfaces de distintos componentes) y otro que contiene el módulo en el que se implementan las interfaces (y, típicamente, el cuerpo de la aplicación en sí, ya que es aquí donde se implementan los eventos que forman la aplicación). La nomenclatura típica es *nombreAppC.nc* para las configuraciones y *nombreC.nc* para las interfaces. Adicionalmente se puede incluir un *header* o cabecera, que se suele nombrar como *nombre.h*.

1.5.2 Ejemplo de aplicación en nesC

A continuación veremos un ejemplo detallado y comentado de la estructura de una aplicación en nesC:

1.5.2.1 *Header* o cabecera

En la cabecera se definen parámetros, estructuras… De forma parecida a como se hace en C. En el ejemplo se ve el código de *BlinkToRadio.h*, parte de una de las aplicaciones que se desarrollan en el proyecto.

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H


enum {
  AM_BLINKTORADIOMSG = 6,
  TIMER_PERIOD_MILLI = 250

};

// Estructura del mensaje que se enviará por radio
typedef nx_struct BlinkToRadioMsg {
 nx_uint8_t nodeid;
 nx_uint8_t counter;
 nx_uint8_t rssi;
 nx_uint8_t lqi;
 nx_uint8_t powerset;
} BlinkToRadioMsg;

#endif
```

Ilustración 14 Ejemplo de cabecera de aplicación

Se puede observar que se define un parámetro, necesario para una clase de Java, varios parámetros más y una estructura. Se profundizará más adelante en qué es cada uno de estos parámetros y para qué sirven.

1.5.2.2 Archivo de configuración

Se detalla a continuación el archivo de configuración *BlinkToRadioAppC.nc*

```
#include <Timer.h>
#include "BlinkToRadio.h"
configuration BlinkToRadioAppC {
}
implementation {
   components MainC;
   components LedsC;
   components BlinkToRadioC as App;
   components new TimerMilliC() as Timer0;
   //componentes necesarios para el envío de mensaje
   components ActiveMessageC;
   components new AMSenderC(AM_BLINKTORADIOMSG);
   components new AMReceiverC(AM_BLINKTORADIOMSG);
   //componente para vamos a ver si podemos leer la signal strenght
   components CC2420ActiveMessageC as Paquete;


   App.Boot -> MainC;
   App.Leds -> LedsC;
   App.Timer0 -> Timer0;
   App.Packet -> AMSenderC;
   App.AMPacket -> AMSenderC;
   App.AMSend -> AMSenderC;
   App.AMControl -> ActiveMessageC;
   App.Receive -> AMReceiverC;
   App.CC2420Packet -> Paquete;

}
```

Ilustración 15 Ejemplo de archivo de configuración

Primero se incluyen las librerías necesarias: la librería *Timer.h* que contiene los componentes asociados a los temporizadores y la librería *BlinkToRadio.h* que se ha visto anteriormente. La sintaxis es similar a C, llamándose a una librería en el directorio local con “”, y a una librería en el directorio por defecto (en este caso para el entorno de TinyOS) con <>.

A continuación se llama a los componentes que se van a utilizar de dos formas:

- Componentes a los que sólo se les va a dar un uso en la aplicación (como son *MainC, LedsC*), ya que no tiene sentido o no existe ningún recurso que necesite más usos del componente, se llaman con *components* seguido del nombre del componente. Por ejemplo, el componente *LedsC* controla los 3 leds de la mota, con lo cual sólo se le da un uso; se utiliza en varias ocasiones pero siempre con el mismo fin.

- Componentes que pueden tener varios usos (*TimerMilliC, AMSenderC*), se llaman con *components new* seguido del nombre del componente. Esto son

componentes a los que se les da varios usos diferentes, y con los que se interacciona de manera diferente. En este ejemplo:

- *TimerMilliC*: Se utiliza el mismo componente pero se puede llamar varias veces, ya que la mota dispone de más de un temporizador.
- *AMSenderC y AMReceiverC*: Son componentes utilizados para comunicación, y se utilizan en este proyecto tanto para la comunicación radio como para la comunicación serie con el ordenador.

Además se pueden especificar alias para los componentes. Por ejemplo el componente de la aplicación *BlinkToRadioC* se utiliza con el alias *App*, o el componente *TimerMilliC* con el alias *Timer0*. En el caso de componentes que se utilicen varias veces, de ser utilizados más de una vez, es necesario especificar alias distintos para cada uno.

Las líneas siguientes cablean las interfaces del componente *App* (la aplicación desarrollada) con cada uno de los componentes a los que pertenecen dichas interfaces. Por ejemplo, en la primera línea:

App.Boot -> MainC

Lo que se especifica es que la interfaz *Boot* que será usada por el componente *App* es provista por el componente MainC. Así, el módulo *App* **utiliza** interfaces y el módulo *MainC* **provee** interfaces.

1.5.2.3 Módulo de aplicación

Se detalla a continuación el módulo de aplicación *BlinkToRadioC.nc*

```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
    uses interface Boot;
    uses interface Leds;
    uses interface Timer<TMilli> as Timer0;
// interfaces necesarias para la comunicación radio
    uses interface Packet;
    uses interface AMPacket;
    uses interface AMSend;
    uses interface SplitControl as AMControl;
    uses interface Receive;
    uses interface CC2420Packet;
}
implementation {...}
```

Ilustración 16 - Ejemplo de módulo de aplicación

Este es el módulo que en el módulo de configuración se nombra como App y se configura para que pueda utilizar las interfaces de los otros módulos.

En él encontramos:

- De nuevo, inclusión de las librerías utilizadas.

- En la definición del módulo (*module {…}*) se implementan todas las interfaces que el módulo **utiliza**. Observar que se utiliza el comando *uses*. Los módulos que **proveen** interfaces utilizan el comando *provides*.

- A continuación vendría la implementación (*implementation {…}*) de la aplicación, que se tratará con detalle en capítulos posteriores.

Capítulo 2

Aplicaciones desarrolladas

En este capítulo se tratará todo lo referente al desarrollo y funcionamiento de las aplicaciones de las que se compone este proyecto.

2.1 Planteamiento del problema

El problema a resolver es la pérdida de objetos, algo tan cotidiano como molesto. Se pretende, entonces, desarrollar un dispositivo pequeño, ligero, de bajo consumo, y con un alcance que debe ser alto pero tampoco necesita ser excesivo (alrededor de 15-20 metros).

El motivo de esto es que se desea conocer cuanto antes el olvido del objeto, por lo que irnos a dispositivos con alcance mayor sólo incrementa el coste, tamaño y consumo sin aportar valor añadido al proyecto.

Inicialmente se piensa en utilizar geolocalización, pero se descarta por la baja precisión para pequeñas distancias, y la falta de cobertura de GPS en interiores, algo clave para este proyecto.

Finalmente, se decide que utilizar dispositivos con una tecnología independiente (así, por ejemplo, en caso de aglomeraciones de gente donde la red móvil y GPS se debilita, nuestros dispositivos podrían funcionar sin problemas), capaces de establecer redes *ad hoc* sin puntos de acceso (lo que proporciona movilidad, también muy importante), y encontramos que Zigbee se ajustaba perfectamente a nuestras necesidades.

Cuando se empieza a estudiar el problema se plantea el hacer dos modos de funcionamiento: uno en el que el usuario solicite que el dispositivo perdido emita algún tipo de señal luminosa o sonora que permitiera su localización y otro en el cual se pueda conocer la distancia y dirección en la cual se encontrara el objeto con alta precisión, para encontrarlo.

El primero de los modos de funcionamiento se desarrolla finalmente, no presenta mayor problema y es particularmente útil para localizar el objeto perdido rápidamente.

A la hora de plantear el segundo modo de funcionamiento se consigue estimar la distancia sin mayores problemas (esto se detallará más adelante), en base al valor *RSSI* (*Received Signal Strength Indicator,* Indicador de la Intensidad de Señal Recibida). Este es el valor de la intensidad de la señal en dBm (decibelios por mili vatio).

No obstante, aparece la complicación de establecer la dirección relativa en la que se encuentra el objeto perdido sin tomar ninguna referencia externa (puntos de acceso fijos, etc.). Además, para la aplicación desarrollada, no parece demasiado relevante el conocer la posición exacta del objeto con precisión de centímetros, ya que el tiempo empleado en localizar el objeto de esta forma es mucho mayor que si simplemente se solicita que emita algún tipo de señal.

Con todo esto, se encuentra una aplicación más útil: aviso de pérdida. Así, cuando el usuario se aleja más de una distancia determinada del objeto, éste recibe un aviso en su dispositivo móvil (en las pruebas, un ordenador) que le notifica que ha perdido el objeto.

Es de esta forma como se llega a la idea que se desarrolla en este proyecto: Una red de dispositivos que ofrezca tres funciones principales:

- Localización individual selectiva o colectiva de los nodos adheridos a la red mediante una señal física.

- Estimación en todo momento de la distancia a del usuario a todos los nodos de la red.

- Aviso de pérdida de los nodos al alejarse el usuario más de una distancia de consigna de los objetos.

En este punto y con un claro concepto de lo que se pretende conseguir, se desarrollan 3 aplicaciones:

- Interfaz de usuario: Programada en C, sirve de interfaz para el usuario y de comunicación con la estación base.

- Estación base: Aplicación que hace de puente entre la comunicación serie con el ordenador y el usuario y la comunicación por radio con los nodos de la red.

- Aplicación de las motas: Es la aplicación que llevan todos los nodos de la red. Recibe y envía paquetes e interpreta los comandos que el usuario envía.

A continuación se analizan en detalle cada una de las aplicaciones.

Aunque no es motivo de este proyecto el análisis del protocolo de comunicación, en el Anexo I que detalla la estructura de los mensajes enviados, lo cual permitirá comprender mejor cómo funcionan las aplicaciones y qué envían y reciben.

2.2 Aplicación de interfaz de usuario

El objetivo de esta aplicación es servir tanto de interfaz de cara al usuario como dar soporte y proporcionar las funciones necesarias para realizar el envío y recepción de información por puerto serie de la estación base.

Esta aplicación está programada en C, está dividida en varios archivos debido a que se ejecuta en múltiples procesos en diferentes ventanas de terminal para dar una mejor experiencia de usuario.

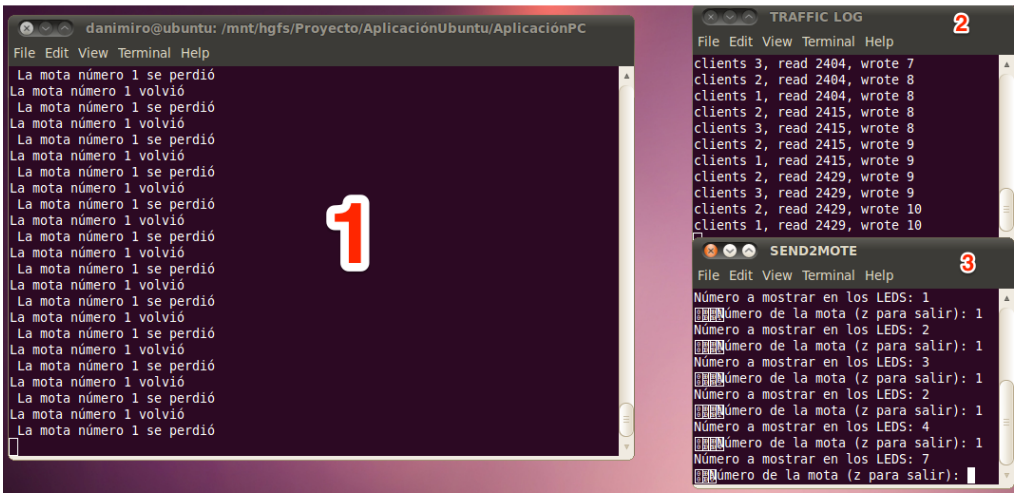

Ilustración 17: Aplicación de interfaz de usuario funcionando

En la imagen se pueden ver 3 ventanas:

- Ventana 1: Registro de mensajes recibidos por las motas hacia la estación base.
- Ventana 2: Log del tráfico: Número de clientes (motas) en la red, mensajes leídos y recibidos.
- Ventana 3: Envío de mensajes a las motas.

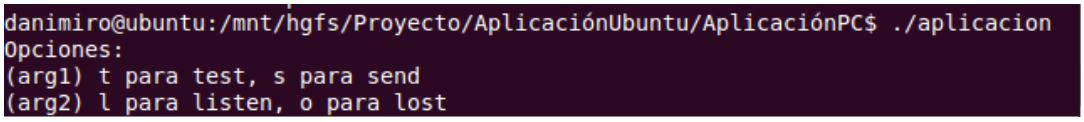

Ilustración 18: Mensaje de inicio de la aplicación

La aplicación tiene varios modos de funcionamiento, que se detallarán a continuación. Como se puede ver en la Ilustración 18, al ejecutar la aplicación hay que especificar en qué modo se quiere hacer.

Un punto clave en el funcionamiento de la aplicación es la creación de varios procesos hijo por varios motivos:

- Para ofrecer una estructura multa ventana como se puede observar en la Ilustración 17, que da una mejora experiencia de usuario
- Al reutilizar funciones ya incluidas en el sistema TinyOS, algunas de ellas requieren que, a su finalización, el proceso que se ejecuta muera.

2.2.1 Funciones propias de TinyOS utilizadas en la aplicación

A continuación se detallan funciones/programas que no han sido creadas sino reutilizadas para el proyecto, ya que son propias del sistema TinyOS.

2.2.1.1 Función serialforwarder()

Esta función crea un proceso hijo que llama al programa *sf*, incluido en el sistema TinyOS. Este programa necesita de un proceso propio para ejecutarse y es por ello que se ejecuta en un proceso hijo.

El programa *sf* actúa como un servidor para gestionar las conexiones en la red de sensores.

El programa requiere de una serie de argumentos que se pasan por línea de comandos:
>>./sf <port> <SerialCOM>

- *<port>*: El puerto de comunicación. Por defecto se usa 9002.

- *<SerialCOM>*: El puerto de comunicación serie al que está conectada la placa MIB520 al ordenador. La nomenclatura es */dev/ttyUSBn*, donde n es el número de puerto. La placa se conecta por dos puertos virtuales al ordenador, siendo estos normalmente consecutivos, utilizándose el más alto para la comunicación serie y el de número más bajo para la programación.

2.2.1.2 Función open_sf_source()

Abre la comunicación con un servidor creado con la función *serialforwarder()*

Se llama de la forma:
*int open_sf_source(const char *host, int port)*

Esta función funciona de manera idéntica a casi cualquier tipo de comunicación en C (tuberías, colas de mensajes, ficheros…).

- *Const char *host*: Un puntero a un array de tipo *char*, o string, en el que se especifica el host de la comunicación. Puede ser una dirección IP o *localhost* si es el equipo local, que es como se usa en este proyecto.

- *int port*: Número de puerto al que conectarse. Se utiliza 9002 por defecto.

- Devuelve una variable de tipo *int* mayor que 0 que será el descriptor de fichero para la comunicación si no ha habido ningún error, o un *int* menor que 0 en caso de error.

2.2.1.3 Función close(fd)

Esta función cierra la comunicación con el servidor, a la que se refiere el descriptor de fichero que se pasa como argumento. Es una función de cierre de descriptor de fichero genérica en C.

Se llama de la forma:
int close (int filedescriptor)

Devuelve:

- 0 si el cierre se realiza correctamente.
- -1 y salida fin con error en caso de algún error en el cierre.

2.1.1.4 Función read_sf_packet

Esta función lee cualquier paquete comunicación que la placa MIB520 transmite a través del puerto serie. Se ejecuta de la siguiente manera:

*void *read_sf_packet(int fd, int *len)*

- *int fd*: Descriptor de fichero asociado a la comunicación con el servidor, que se obtiene de la función *open_sf_source.*
- *Int *len*: Puntero que devuelve la longitud del mensaje recibido.
- Devuelve: un array de tipo *void* (se lee como array de tipo *unsigned *char*) con el mensaje recibido. *NULL* en caso de error.

También reserva memoria para la variable en la que se lee el mensaje del tamaño del mensaje.

La estructura de los mensajes recibidos se detalla en el Anexo I.

2.2.2 Módulo Aplicación

Este módulo está formado por el programa que contiene el flujo principal, *aplicación*, y la cabecera donde se encuentra el código de las funciones, *aplicacion.h*.

Para comprender mejor el diagrama de flujo, se recuerda que en C la función *fork()* crea un proceso hijo del proceso actual y devuelve:

- -1 en caso de error en el proceso de *fork.*

- 0 cuando el proceso ejecutándose es el proceso hijo (ya que 0 es el id del padre).
- El id del hijo cuando el proceso ejecutándose es el proceso padre.

Así, en los diagramas de flujo se ha considerado como un nodo multiopción.

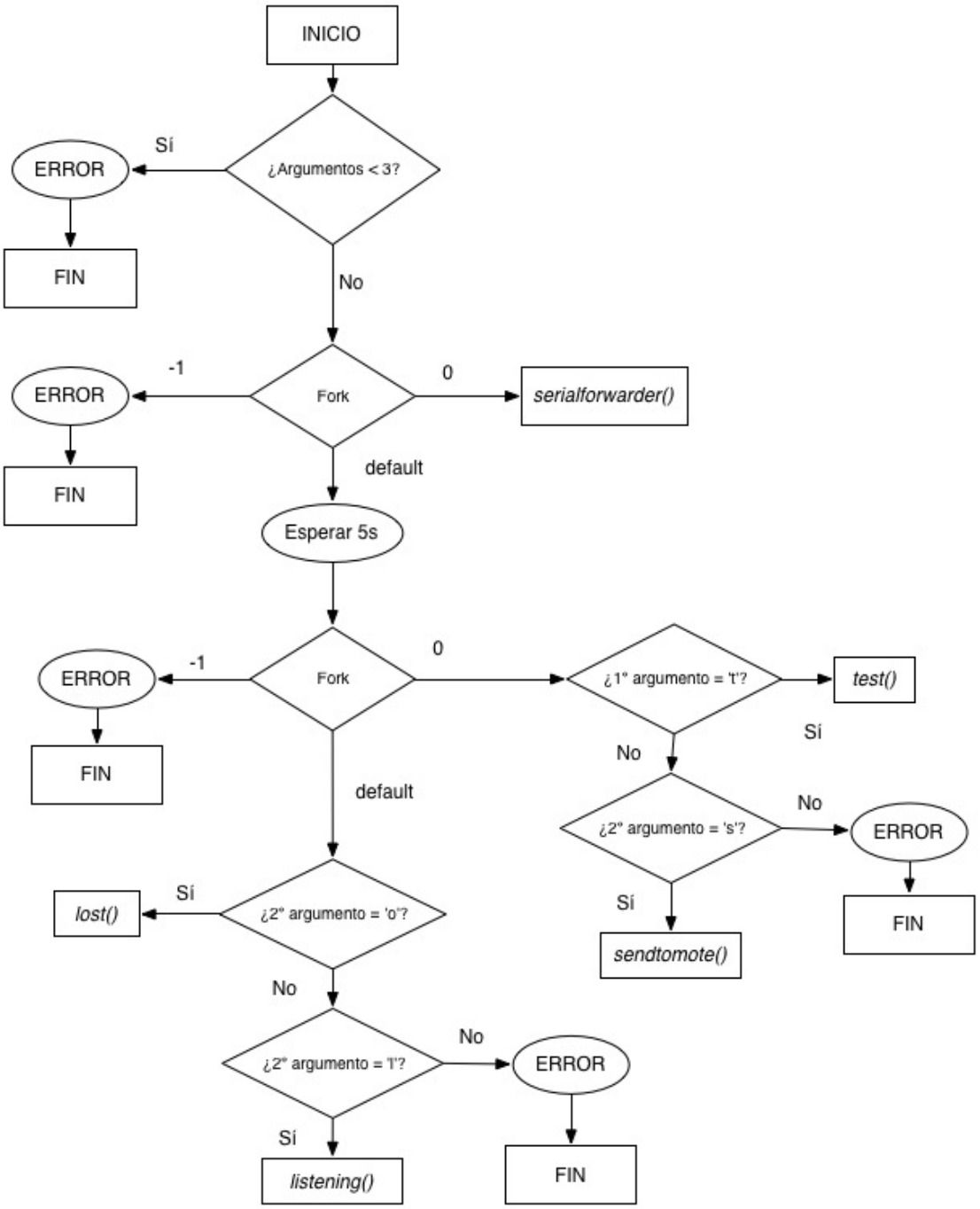

Ilustración 19: Diagrama de flujo del módulo *aplicación*

El funcionamiento de este módulo es sencillo, evalúa los argumentos de entrada y según ellos ejecutando el modo correspondiente. Así, la aplicación se ejecuta con:

./aplicación arg1 arg2

Donde *arg1* puede ser:

- "t" para ejecutar un modo de test en el que se envían constantemente mensajes a todas las motas de la red.
- "s" para ejecutar el modo de envío en el que se envía un valor a cualquiera de las motas o a todas ellas y estas lo muestran con el array de leds.

Y *arg2* puede ser:

- "l" para el modo de escucha en el cual se muestran por pantalla los mensajes procedentes de las motas.
- "o" para el modo en el cual se muestran mensajes cuando las motas se "pierden" o alejan demasiado.

Además, la aplicación termina con error si:

- El número de argumentos es menor que 3 (el propio nombre de la aplicación se cuenta como *arg0*, y los argumentos más allá del segundo se ignoran).
- Hay algún error en cualquiera de los *fork*.
- El modo introducido no es válido.

2.2.2.1 Función listening()

Esta función escucha todo lo que se recibe de la red de motas y lo muestra por pantalla. La función se llama de esta forma:

void listening()

Y tiene la siguiente estructura:

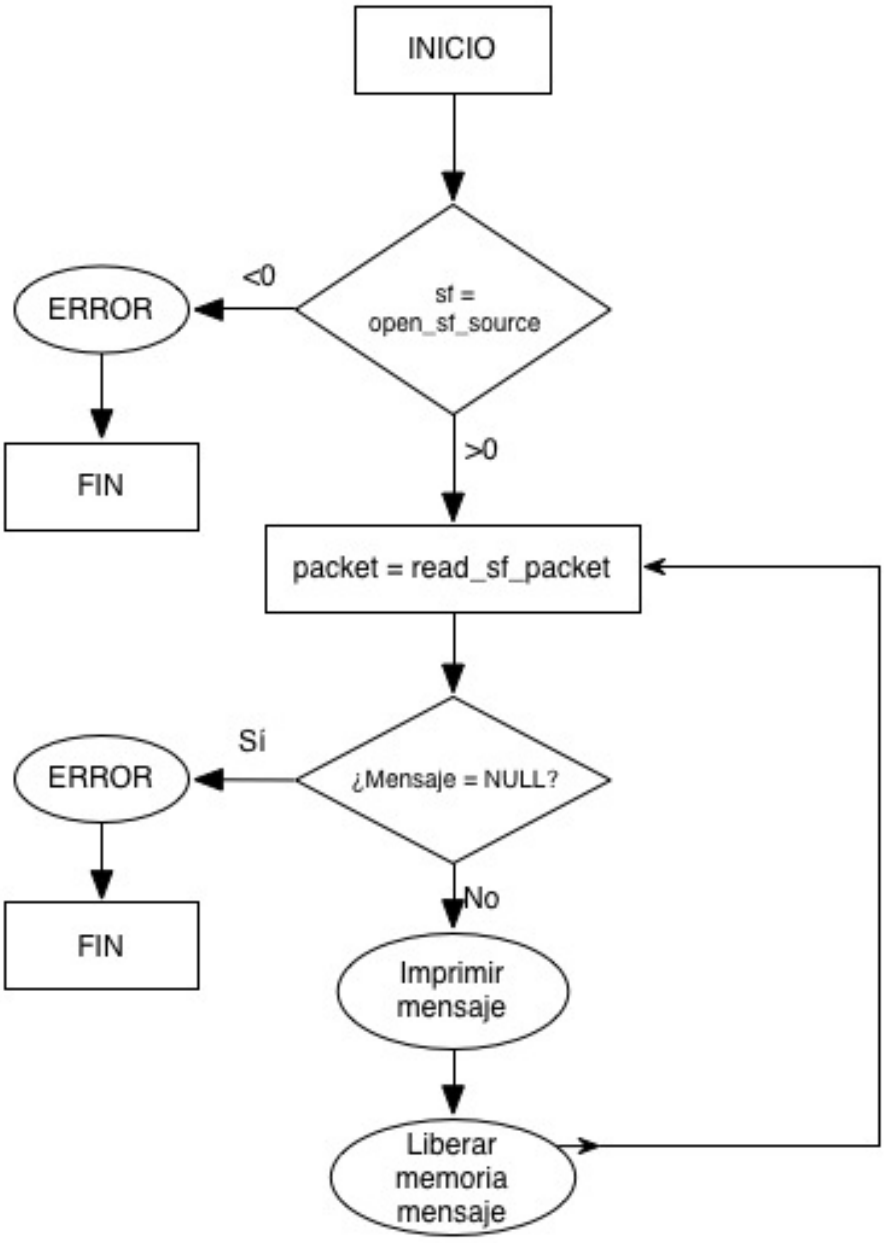

Ilustración 20: Diagrama de flujo de la función *listening*

La función se ejecuta en bucle infinito ya que será un proceso hijo que controla los envíos de mensajes (y con el que interactúa el usuario) el que finalizará este proceso y todos sus procesos hijo con una señal de tipo *SIGKILL*.

La aplicación cargada en las motas, entre otras cosas, hace que estas emitan mensajes espaciados un período de tiempo consignado. Esto se verá más adelante cuando se detalle dicha aplicación.

2.2.2.2 Función lost()

La función *lost* es muy parecida a la función *listening*, vista en el apartado anterior.

Mientras que *listening* saca por pantalla todo mensaje enviado o recibido por las motas, *lost* sólo saca un mensaje cuando alguna mota de la red ser "pierde". Para la pérdida se establece un nivel de RSSI o bien una distancia límite a partir de la cual se considera el objeto como perdido y el sistema alerta de ello. Cada vez que una mota se pierde, el nodo base (en este caso, el ordenador al que está conectado) emite un sonido.

Además, cada mota tiene asignado un *flag* de pérdida. Cuando la mota se pierde el flag se pone a 1 (así el sistema no alerta varias veces de la pérdida de la misma mota), y cuando se "encuentra", el *flag* se vuelve a poner a 0.

Cabe destacar que la idea es que el nivel de pérdida se produzca cuando todavía hay comunicación, de forma que se puede hacer que el objeto perdido suene o brille para facilitar su búsqueda.

La función se llama así:
void lost()

Y su estructura es la siguiente:

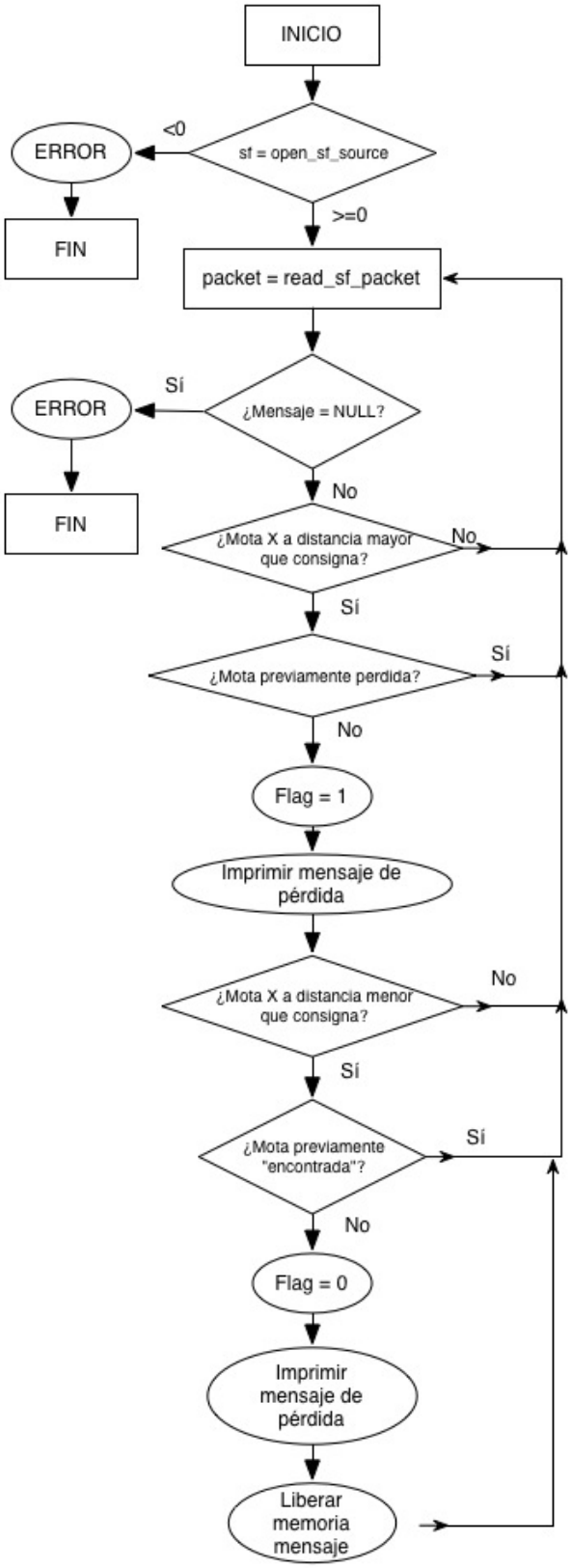

Ilustración 21: Estructura de la función *lost()*

Al igual que la función *listening,* esta función se ejecuta en bucle infinito hasta que se finaliza el proceso hijo.

2.2.3 Módulo send()

Este módulo contiene todo lo necesario para el envío de mensajes a las motas y la interacción del usuario con la aplicación. Se divide en dos procesos, el primero de los cuales envía información a la mota a petición del usuario, y el segundo envía mensajes de localización, los cuales el nodo replica y permite conocer dónde se encuentra.

Ya que corre en una ventana independiente, esto es, en un proceso hijo del módulo aplicación, se desarrolla en un programa aparte.

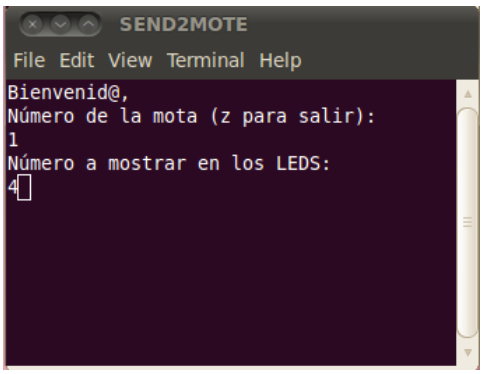

Ilustración 22: Módulo *send*

Como se ve en la Ilustración 22, al inicio la aplicación muestra las opciones disponibles para el usuario. Estas son:

- Número del nodo al que se quiere enviar el mensaje.
- 0 para enviar un mensaje a todos los nodos de la red.
- "Z" para salir y provocar el cierre de este proceso hijo, de su proceso padre, y de todos los procesos hijo del padre.

Imagen del número a enviar, y presumiblemente imagen de una mota con el número de nodo bien claro y el número mostrado.

Con el fin de reutilizar al máximo las funcionas ya incluidas en TinyOS, se adapta la información recibida a la estructura del mensaje que se envía. La estructura del mensaje (que incluye la genérica de tinyOS y la estructura que se desarrolla dentro del área *payload* para este proyecto en concreto).

La forma de leer y transformar la información puede parecer algo compleja, pero experimentalmente se comprueba que es la única forma de conseguir que la información introducida por teclado se envíe y lea correctamente.

Esta forma es la siguiente:

- Las variables se leen o definen como *char* en un array de tipo *char** (ya que las variables *char* en C son de 1 byte).
- A cada uno de los elementos de dicho array se les añade a continuación el carácter 'NULL' o '\0', que indica el final de una cadena de caracteres.
- Cada uno de estos elementos se transforma individualmente al tipo *long* y se leen en un array de tipo *unsigned char*.

Esto puede parecer extraño pero, como ya se ha mencionado, es la única forma con la que se prueba, experimentalmente, que la información se recibe correctamente. Además, es la forma que se utiliza en el programa *sfsend.c*, función incluida en el sistema TinyOS y cuyo código se encuentra en el Anexo II.

La estructura más detallada del programa *send* es la siguiente:

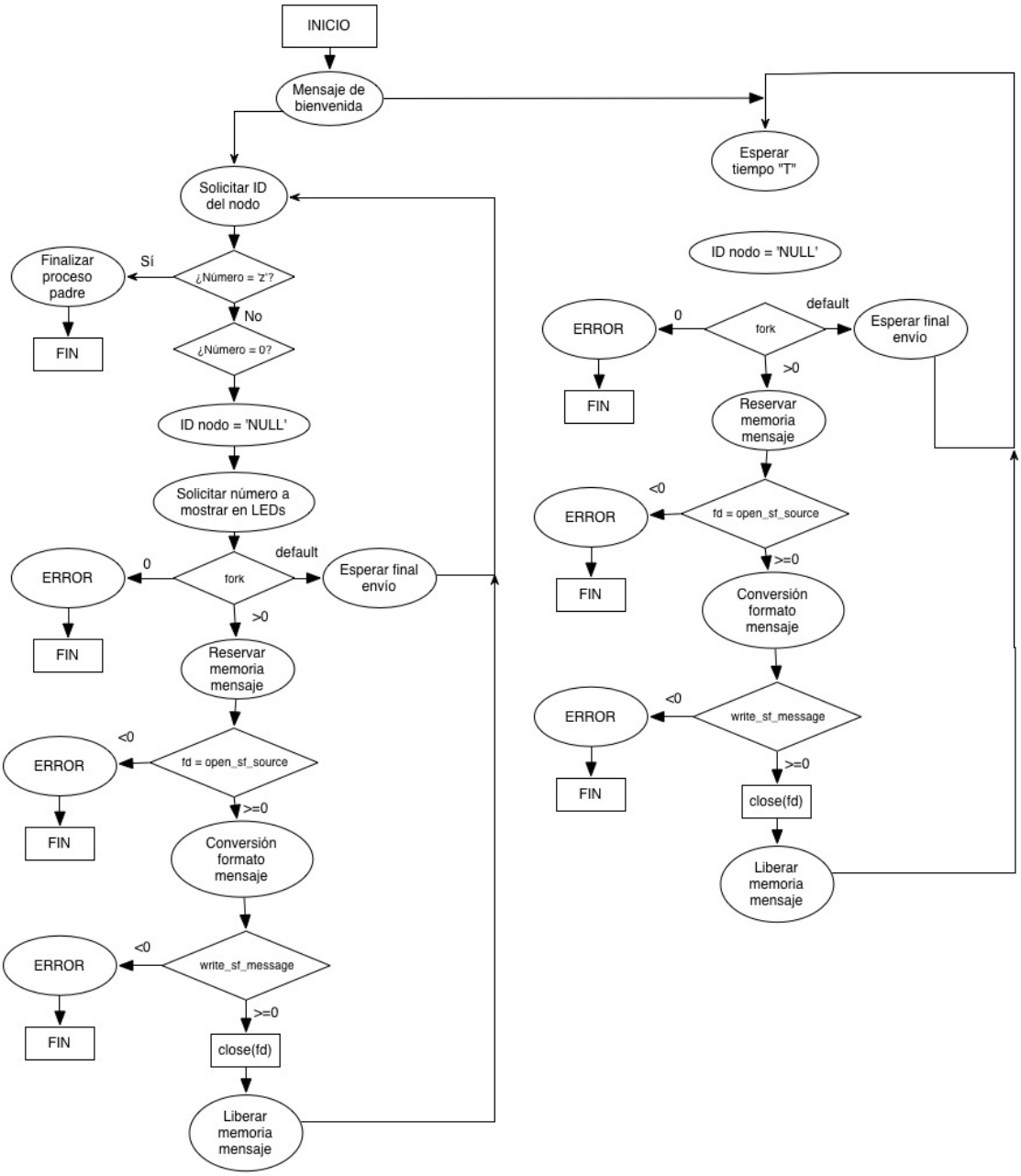

Ilustración 23: Estructura del programa *send*

El tiempo T es el tiempo definido como período de envío entre mensaje y mensaje.

Notar que la función se ejecuta en bucle hasta que se introduce el carácter 'z', que finaliza tanto este programa como su proceso padre y todos los otros hijos que este tuviera, resolviendo así el problema planteado anteriormente de que otros programas se ejecutaran en bucle infinito.

El mensaje enviado contiene muchos más campos que se detallan en el Anexo I.

2.2.4 Módulo test()

Este módulo, como su nombre lo indica, es para realizar pruebas.

Funciona de la misma manera que *send*, pero envía constantemente un contador que se incrementa hasta 7 y luego vuelve a 0, y envía y muestra esos números en los leds de todas las motas de la red (con un período determinado).

La estructura del programa es la siguiente:

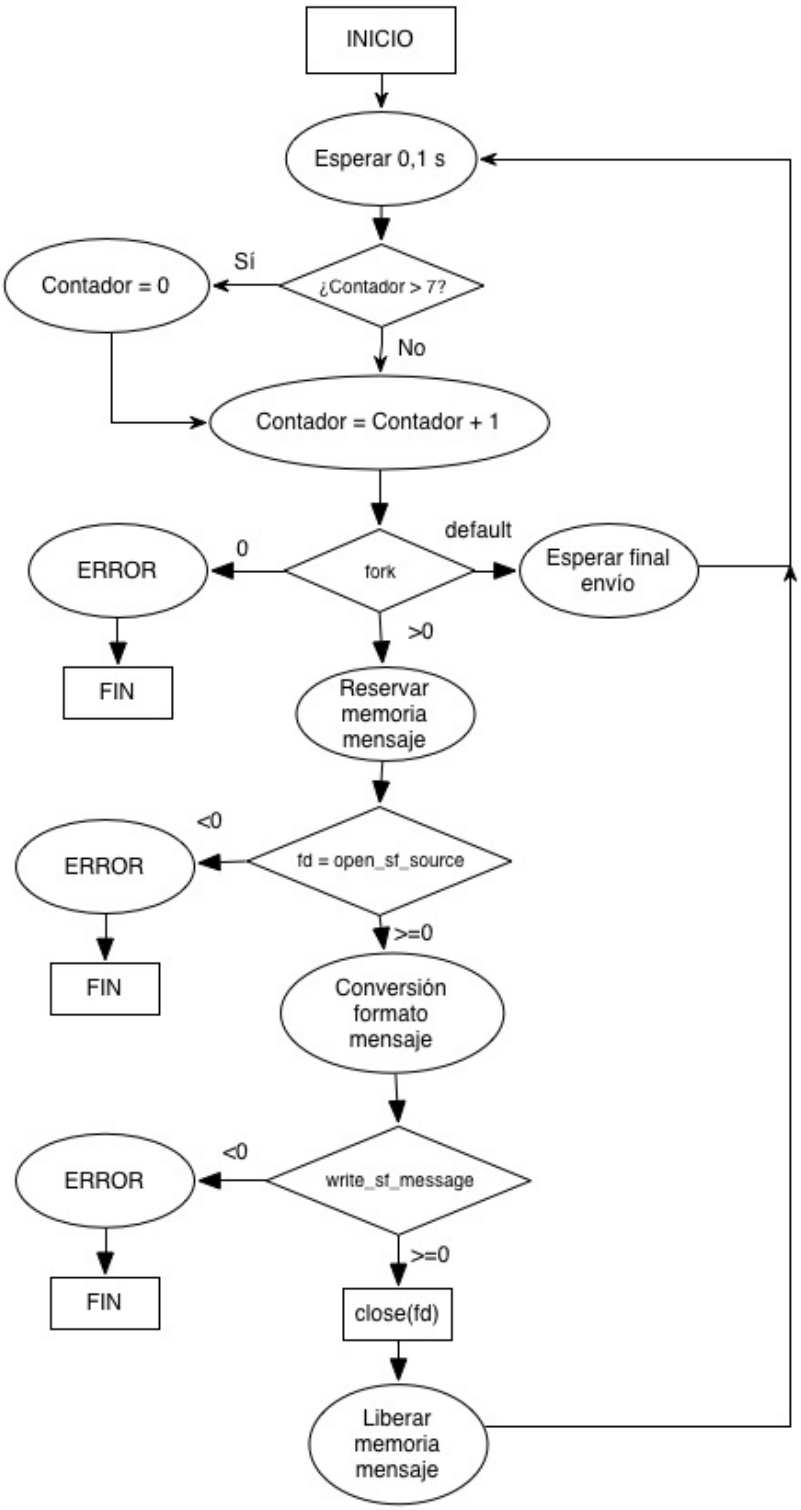

Ilustración 24: Estructura del módulo *test*

2.2.5 *Makefile*

Para facilitar la portabilidad y compilación de la aplicación, y dado que son muchos los archivos fuente a compilar, se desarrolla e incluye un *makefile*.

Las ventajas de usar un *makefile*, además de ahorrar tiempo y líneas de comandos, es que compila de forma inteligente, compilando sólo los archivos fuente que hayan cambiado desde la última compilación.

Tiene dos modos de funcionamiento:

- *make*: Compila los archivos fuente que hayan sufrido cambios, primero creando archivos objeto, para que puedan ser portados sin poder ver el código, y posteriormente genera los ejecutables.

- *make clean*: En caso de que la compilación no se realice de manera satisfactoria, con este comando se eliminan los archivos objeto y se vuelven a compilar y a generar los ejecutables a partir de los archivos fuente.

Alternativamente a esto, los archivos fuente pueden ser compilados de forma normal (en Linux, con el comando *gcc*).

2.3 Aplicación findmemote

Esta es la aplicación que se carga en los nodos móviles o motas. Su principal función es la de interpretar los mensajes procedentes del nodo base, y enviar la información requerida para su localización.

La aplicación está escrita en NesC, lo que hace que esté compuesta de varios archivos, necesita de varios componentes y utiliza interfaces de esos componentes como se verá a continuación.

En el archivo *findmemoteAppC* se declaran los componentes utilizados (la propia aplicación es uno de ellos), y se cablean las interfaces entre los componentes que las **proveen** y las que las **usan**. En este caso todos los componentes son proveedores de interfaces excepto el componente principal de aplicación, con lo cual se cablean todas las interfaces del componente de aplicación con los otros componentes declarados.

En el archivo *findmemoteC* se declaran las interfaces que la aplicación **usa**, y la implementación del componente (lo que para nosotros será el flujo de la aplicación *findmemote*).

En el archivo *findmemote.h* se definen algunos parámetros (como el tipo de mensaje o el tiempo para el temporizador) y la estructura del mensaje utilizadas.

El archivo *makefile* contiene una referencia al objeto que se debe compilar y a las *MAKERULES* específicas de TinyOS.

2.3.1 Componentes e interfaces utilizados

Componentes utilizados:

- MainC: Controla el inicio de la aplicación y de sus componentes básicos. Además ciertas interfaces necesitan se inicializadas aquí (como, por ejemplo, las que utilizan el envío de mensajes por puerto serie o radio).

- LedsC: Componente que controla el comportamiento de los leds. Permite varios modos de funcionamiento, desde controlar cada led individualmente hasta utilizar los 3 en conjunto para representar números en binario.

- FindmemoteC: Es el componente principal de la aplicación y el que se ejecutará en las motas.

- TimerMilliC(): Se crea un nuevo componente para utilizar un temporizador. Lleva delante *new* porque puede ser utilizado varias veces como temporizadores independientes.

- ActiveMessageC: Dado que TinyOS soporta múltiples plataformas que pueden tener distintas configuraciones y drivers para las radios, ActiveMessageC se encarga de proveer las implementaciones específicas necesarias para que cada plataforma funcione correctamente. En este caso, proveerá las implementaciones y configuraciones óptimas para la mota MicaZ.

- AMSenderC(msgtype): Provee todas las interfaces para el envío de mensajes por radio o puerto serie.

- AMReceiverC(msgtype): Provee todas las interfaces para la recepción de mensajes por radio o puerto serie.

Para ambos AMSenderC y AMReceiverC se crea un nuevo componente ya que se puede tener varios componentes para varios tipos de mensajes o de envíos. El tipo de mensaje ayuda a multiplexar y así ampliar el acceso a la radio o al puerto serie según proceda, actuando de forma similar al tipo de mensaje en el protocolo de Ethernet.

- CC2420ActiveMessageC: Este componente provee interfaces específicas de la plataforma MicaZ para leer parámetros que se incluyen en los mensajes tales como la intensidad de la señal, la calidad del *enlace* de comunicación, etc.

Interfaces utilizadas:

- Boot: Controla la inicialización de los componentes y de la aplicación.

- Leds: Controla los 3 leds de la placa.

- Timer<TMillli>: Temporizador con sus funciones de control (start, stop, startoneshot…).

- Packet: Interfaz para interpretar los mensajes enviados y recibidos para una plataforma específica. Permite leer los distintos campos del mensaje, y obtener el tamaño de la *payload* (o área de carga, la parte del mensaje que contiene la información que se envía intencionadamente), y crear un puntero a esa *payload* para facilitar su lectura.

- AMPacket: Es una interfaz similar a Packet, pero que además permite leer parámetros que se incluyen en el mensaje como consecuencia de la comunicación radio, como por ejemplo la AM ID de la mota en cuestión.

- AMSend: Es similar a la interfaz Send, pero adaptada a la comunicación por radio. Básicamente, provee la interfaz básica de envío del componente ActiveMessageC, adaptado a la plataforma en uso, incluyendo además el campo de destinatario para enviar a un nodo específico en la red.

- SplitControl: Es una interfaz que se utiliza para controlar el inicio y fin de los componentes.

- Receive: Controla la recepción de mensajes y su correcta lectura. Provee tanto un evento para recibir mensajes como funciones para obtener el tamaño y el contenido de la *payload* del mensaje.

- C2420Packet: Es una interfaz del componente C2420ActiveMessageC que se utiliza para leer otros datos que contiene el mensaje además de la *payload* como pueden ser la intensidad de señal, la calidad del enlace, etc. Además, permite establecer la potencia del mensaje enviado (o *powerset*).

Notar que en algunas ocasiones se utiliza la palabra reservada *as*. Esta palabra permite darle a las interfaces o componentes un nombre que resulte más identificable para el usuario que el nombre genérico. Así, al componente *TimerMilliC()* se le renombra como *Timer0*, al componente *C2420ActiveMessageC* se le renombra como *Paquete*, y a la interfaz *SplitControl* se la renombra como *AMControl*.

2.3.2 Estructura de la aplicación

Como ya se ha comentado, a pesar de que la aplicación utiliza varios componentes, el que realmente contiene la implementación de la aplicación será el componente *findmemoteC*.

Esta aplicación, como en general todas en nesC, se basa en eventos (lo que se conoce como interrupciones) los cuales proveen las interfaces, y son los que van condicionando la ejecución de la aplicación. Así, no se trata de una aplicación secuencial, sino que la mota se queda a la espera de que se ejecuten distintos eventos y va ejecutando código en función de esto, exceptuando el evento *boot.booted()* que marca el inicio de todos los componentes y de la aplicación.

Interfaces sencillas como un temporizador no requieren de una especial atención, ya que su inicialización es trivial y es complicado que se produzca algún error.

Interfaces más complejas como la comunicación AM, el envío de mensajes o la inicialización de la propia aplicación pueden dar lugar a algún error y es por ello que se deben inicializar y luego evaluar las siguientes acciones en un evento que se genera si el inicio no ha dado lugar a error.

Por ejemplo, la interfaz *AMControl* se inicializa una vez que se comprueba que la aplicación se ha inicializado correctamente (*evento boot.booted()*):

```
event void Boot.booted() {
        call AMControl.start();
}
```

Pero las acciones asociadas a la comunicación por radio (en este caso, un temporizador periódico) ocurren cuando se recibe el evento de que la interfaz *AMControl* se ha realizado correctamente:

```
event void AMControl.startDone(error_t err) {/
        if (err == SUCCESS) {
                call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
        }
        else {
                call AMControl.start();
        }
}
```

Así se comprueba que no surja ningún error que comprometa la correcta ejecución.

Findmemote tiene dos funciones principales:

- Recepción e interpretación de mensajes: Se interpreta la información recibida por radio tomando acciones (por ejemplo, mostrar un valor numérico en el array de leds).

- Envío periódico de mensajes al nodo central: Además de los datos "indirectos" (calidad del enlace, potencia de la señal, etc.) se puede incluir otra información (en este caso se envía el valor de un contador). La aplicación está diseñada para enviar un mensaje cada vez que recibe un mensaje, de forma que si se pierde la conexión con la base, no continúa mandando mensajes y consumiendo energía innecesariamente.

La estructura de la aplicación es la siguiente:

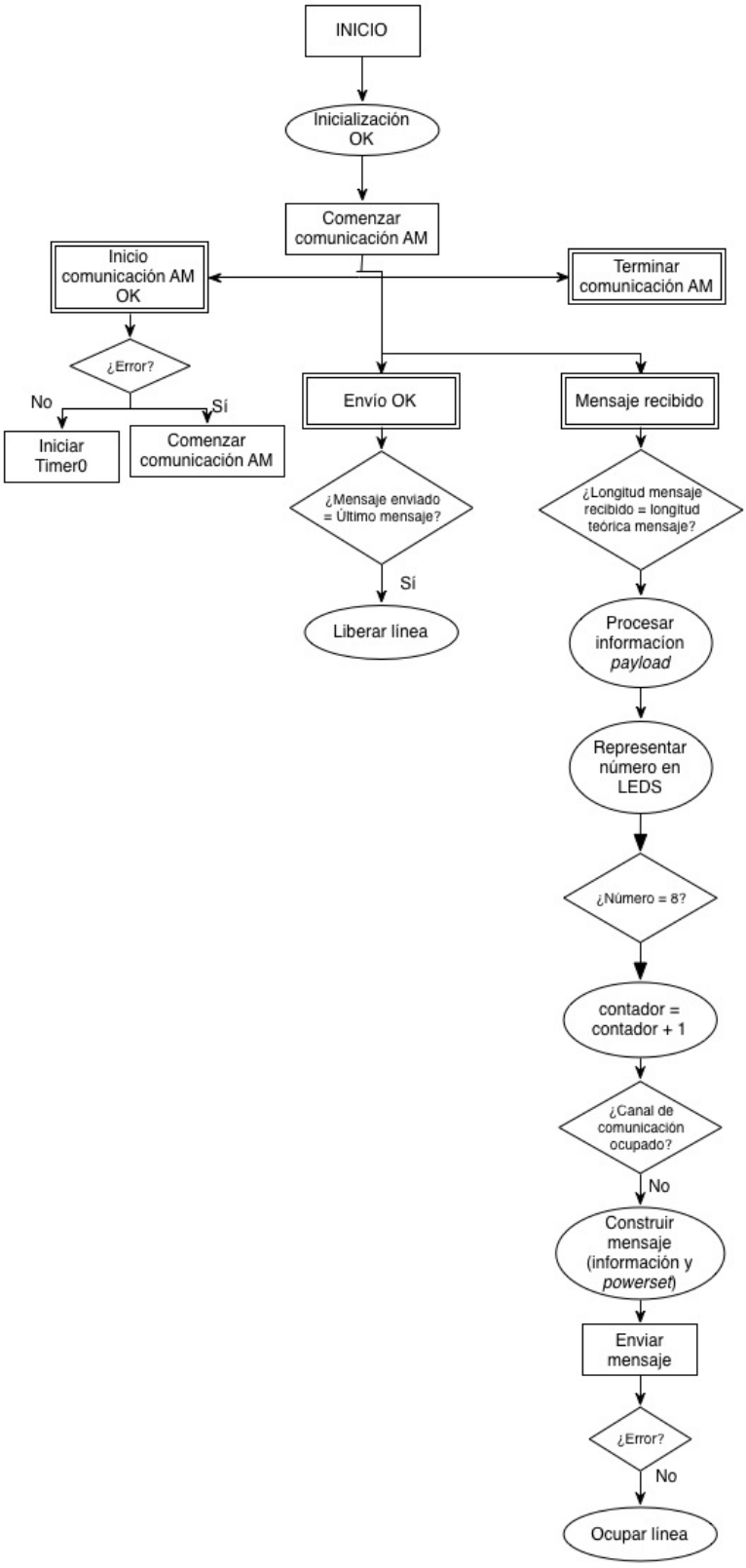
INICIO
Inicialización OK
Comenzar comunicación AM
Inicio comunicación AM OK
Terminar comunicación AM
¿Error?
No
Sí
Iniciar Timer0
Comenzar comunicación AM
Envío OK
Mensaje recibido
¿Mensaje enviado = Último mensaje?
Sí
Liberar línea
¿Longitud mensaje recibido = longitud teórica mensaje?
Procesar información payload
Representar número en LEDS
¿Número = 8?
contador = contador + 1
¿Canal de comunicación ocupado?
No
Construir mensaje (información y powerset)
Enviar mensaje
¿Error?
No
Ocupar línea

Ilustración 25: Estructura de la aplicación *findmemote*

Notas:

- Dado que la aplicación se basa en eventos, estos se han marcado con un cuadrado de borde doble. Se debe entender que, una vez terminado un evento, el programa no finaliza, sino que se queda a la espera de que otro evento ocurra.

- A pesar de que no se ejecuta ningún código, el evento de comprobación de finalización de la comunicación AM se debe incluir para que el programa compile y funcione.

- El contador que se transmite no se inicializa ya que vuelve a 8 al desbordar, al llegar a 255.

- La comprobación de que el mensaje enviado es el último mensaje atiende a las siguientes razones:
    - Puede ocurrir que se dispare el evento por el envío de un mensaje anterior al último que se ha puesto en "cola" de envíos, con lo cual el último mensaje no se enviaría y se perdería.
    - Podría ocurrir algún error a la hora de escribir el mensaje.
    - La comprobación se realiza comprobando que la dirección de memoria en la que se aloja la *payload* del último mensaje enviado se corresponde con la dirección de memoria en la que se aloja el último mensaje puesto en cola.

- Se comprueba que la longitud del mensaje recibido se corresponde con la longitud teórica que debería tener para detectar posibles errores en la transmisión o lectura.

2.4 Aplicación BaseStationMod

Esta es la aplicación sirve de puente entre el ordenador y la red de motas. Corre en el nodo base y su principal función es replicar los mensajes que se reciben por radio por el puerto serie al ordenador y viceversa. Además, se representa con el parpadeo de los leds si se envía (Led0, rojo), recibe (Led1, verde) o si se pierde un mensaje (Led2, amarillo)

Esta aplicación consta de dos archivos: *BaseStationC*, donde se declaran todos los componentes y se cablean las interfaces utilizadas, y *BaseStationP*, que es donde se definen las interfaces que la aplicación **utiliza** y que contiene el código de la aplicación en sí.

2.4.1 Componentes e interfaces utilizados

Los componentes utilizados son: MainC, BaseStationP, LedsC, ActiveMessageC, SerialActiveMessageC, CC2420ActiveMessageC.

El único nuevo con respecto a la aplicación *findmemote* es el componente SerialActiveMessageC, que es el homólogo a ActiveMessageC pero para comunicación serie.

Las interfaces utilizas son: Boot, SplitControl, AmSend, Packet, Receive y CC2420Packet, todas las cuales han sido explicadas anteriormente en el apartado 2.3.1.

Esta aplicación está escrita utilizando lo que se conoce como "Safe TinyOS" [18], que es una variante de TinyOS que principalmente tiene la ventaja de mejorar la seguridad de la memoria.

Se puede observar que al principio del módulo se escribe *@safe*:

```
module BaseStationP @safe() {
```

Aparte de eso, se utilizan arrays de tipo *ONE* y *ONE_NOK* como punteros seguros o "safe pointers". Esto funciona de la siguiente manera:

- Cuando se crea un puntero de tipo *ONE*, este nunca puede apuntar a algo que no sea una variable del tipo especificado. Por ejemplo, el puntero

uint8_t pointer ONE puntero*

Sólo puede apuntar a direcciones de memoria en las que se almacenen datos de tipo uint8_t (enteros de 8 bits).

- Un puntero de tipo *ONE_NOK* es igual que un puntero de tipo *ONE*, sólo que además también puede apuntar a *NULL*.

Otros comandos poco habituales que se utilizan en el código de esta aplicación son:

- *post task*: TinyOS incluye un planificador de tareas. Cuando una tarea se llama con *post*, implica que la tarea se ejecutará cuando todos los recursos que necesita estén disponibles.

- *atomic*: Las instrucciones comprendidas en *atomic{...}* se ejecutarán de forma secuencial sin que se ejecute ninguna otra instrucción procedente de otro evento o función entre medias.

2.4.2 Funcionamiento de la aplicación

Esta aplicación está incluida en la parte de soporte y comunicaciones de TinyOS. Para su uso para este proyecto se hacen algunas modificaciones para poder leer determinados campos del mensaje recibido (RSSI, LQI), establecer el *powerset* o potencia de señal enviada, e implementar la estructura de mensaje que se utiliza en toras las aplicaciones.

Esta aplicación se basa en las interfaces de comunicación específicas de la placa CC2420, con los requerimientos que estas presentan, como por ejemplo tener un buffer de mensajes para almacenar los mensajes en la pila, y un buffer de direcciones que apunta en todo momento al elemento que se debe transmitir o donde se debe almacenar el siguiente elemento recibido, para conservar siempre un sistema de tipo *FIFO.*

De esta manera, el sistema tiene una cola de mensajes predefinida (por defecto, 12), recibe los mensajes por radio, los reenvía por el puerto serie, y viceversa.

Además, el programa gestiona las colas de mensajes y avisa en el caso de pérdida de mensajes.

En el Capítulo 3 se harán pruebas de recepción de mensajes y se verá cuál es el límite en mensajes simultáneos y en tiempo mínimo entre mensaje y mensaje necesarios para que no se pierdan mensajes.

Capítulo 3

Pruebas, análisis y modelo de comportamiento

En este capítulo se desarrollan pruebas con dos objetivos: estimación de distancia de un nodo a la base, y frecuencia máxima de comunicación admisible para uno o varios nodos.

3.1 Variación del valor RSSI con la distancia

El RSSI (*Received Signal Strength Indicator*, indicador de intensidad de señal recibida*)* es un indicador de la intensidad de la señal recibida en dBm (decibelios por mili vatio). Este varía con la distancia y con el *powerset* (valor que limita la potencia de la señal enviada). A continuación se muestran varias mediciones que se han hecho, y cómo se puede relacionar con bastante exactitud el valor del RSSI con la distancia a la que se encuentran los nodos.

Para un mensaje enviado se puede establecer un *powerset*, o potencia de señal enviada, en función de la cual el valor RSSI varía de forma diferente. Este valor varía entre 1 y 31.

A continuación se presentan las medidas hechas para niveles de potencia de 2, 3, 5, 10, 20 y 30 con el fin de determinar un modelo matemático que permita dar un valor analítico de la distancia que separa el nodo base de cualquier otro nodo con la mayor exactitud posible. Se eligen estos niveles de potencia concretos porque con un solo nivel no es posible estimar la distancia en todos los rangos de manera exacta. Se necesitan más niveles a distancias más cortas (como se verá más adelante) y menos cuanto mayor es la distancia porque las medidas divergen más a menor distancia, y hay mayor variación entre uno u otro modo.

El valor de RSSI es muy sensible y varía en función de muchas cosas, por eso, para obtener los mejores resultados y más fiables, se realizan varias medidas en cada distancia, dando como resultado el promedio de todas ellas con la idea de que los datos atípicos no influyan en el resultado del análisis.

3.1.1 *Powerset 2*

A continuación se muestran los datos obtenidos para un nivel de potencia de 2:

Distancia [m]	RSSI
0,05	189
0,10	183
0,20	173
0,30	172
0,40	172

Tabla 7: Distancia vs RSSI para *powerset* 2

Llevando estos datos a una gráfica, obtenemos:

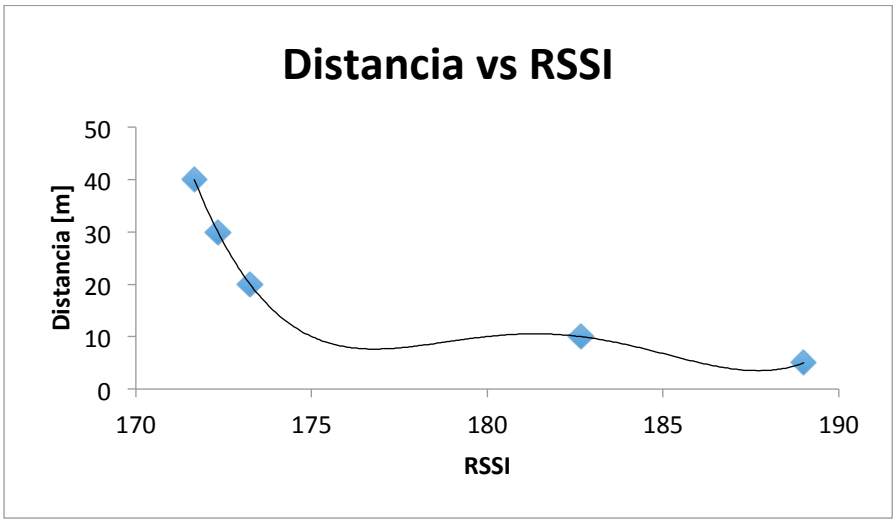

Ilustración 26: Distancia vs RSSI para *powerset* 2

En donde se observa una clara tendencia de los datos, que viene dada por la ecuación (con y la distancia en metros, x el RSSI):

$$y = 0{,}000053142689442609\,x^4 - 0{,}038677983722572\,x^3 + 10{,}5531928011355x^2 - 1.279{,}35492439x + 58.143{,}4712968649$$

Ecuación 1: Distancia [m] en función del RSSI para *powerset* 2

Analizando la ecuación podemos observar los datos experimentales frente a los analíticos, y el error:

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
5	189	5,00	0,00	0,00
10	183	9,23	1,38	0,14
20	173	21,50	7,02	0,35
30	172	30,57	5,68	0,19
40	172	42,59	14,36	0,36

Tabla 8: Distancia analítica, error y error porcentual para *powerset* 5

Para este rango de distancias se obtiene un error promedio de 21 % (desviación estándar de 0,15), lo que permite afirmar que el modelo planteado es una estimación razonablemente buena de la distancia real.

3.1.2 *Powerset* 3

A continuación se muestran los datos obtenidos para un nivel de potencia de 3:

Distancia [m]	RSSI
0,315	184
0,63	179,3333333
0,945	176,6666667
1,26	174,6666667
1,575	174,3333333
1,89	171,6666667
2,205	171
2,52	172
2,835	169,3333333
3,15	168

Tabla 9: Distancia vs RSSI para *powerset* 3

Las medidas terminan a una distancia de 3,15 m porque para distancias mayores la comunicación se vuelve irregular, las medidas divergen y se pierde gran cantidad de los mensajes.

Tras graficar los datos obtenidos:

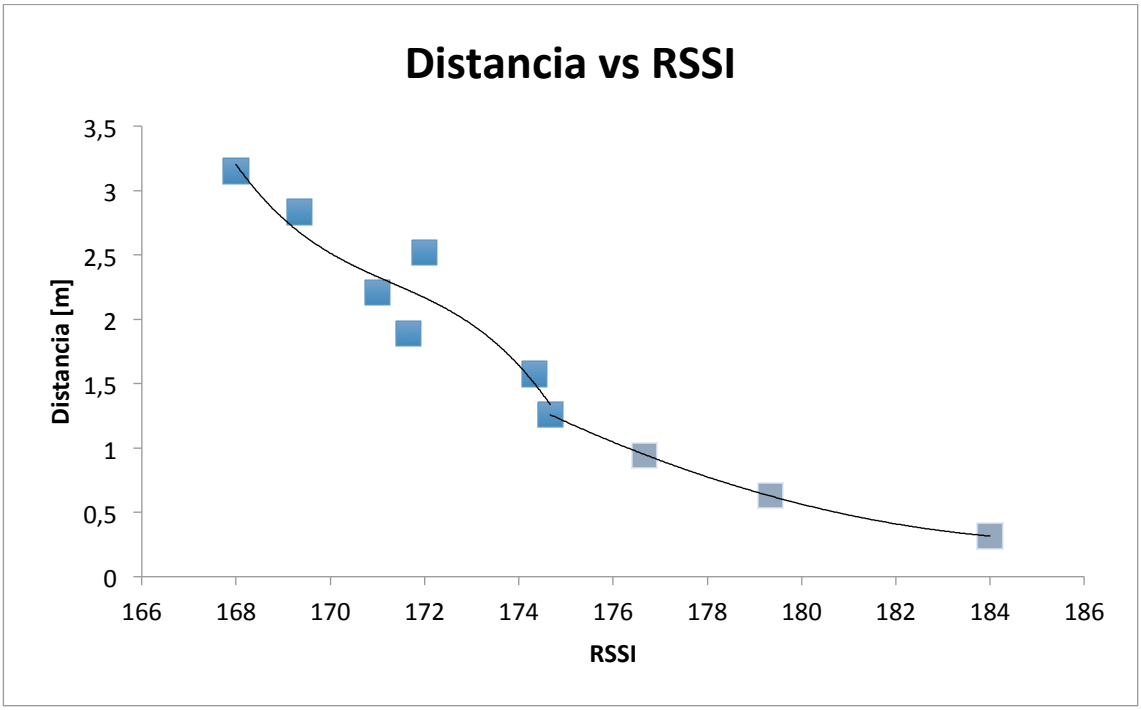

Ilustración 27: Distancia vs RSSI para *powerset* 3

Se puede apreciar que los puntos siguen una tendencia más o menos clara. En el gráfico se representan dos series, esto es fruto de varias iteraciones probando diferentes tipos de línea de tendencia para minimizar el error de la función analítica frente a los valores reales, llegando a la conclusión de que dos series con dos ecuaciones distintas es la opción que minimiza los errores.

Así, la expresión que define la curva es (con y la distancia en metros, x el RSSI):

$$y = 0{,}00732231920199501\,x^2 - 2{,}727286159601\,x + 254{,}231685536;\ x \leq 1{,}6\ [m]$$
$$y = -0{,}0108932308035641\,x^3 + 5{,}59840418767656\,x^2 - 959{,}229369000047x + 54.796{,}0675445303;\ x > 1{,}6\ [m]$$

Ecuación 2: Distancia [m] en función del valor de RSSI para *powerset* 3

Con eso, se obtienen los resultados siguientes:

Distancia real [m]	RSSI	Distancia Analítica [m]	Error	Error (%)
0,315	185	0,305	0,011	3,45%
0,630	179	0,649	0,163	25,94%
0,945	177	0,960	0,157	16,56%
1,260	175	1,260	0,076	6,06%
1,575	174	1,366	0,334	21,18%
1,890	172	2,267	0,543	28,74%
2,205	171	2,337	0,157	7,11%
2,520	172	2,127	0,567	22,50%
2,835	169	2,742	0,337	11,88%
3,150	168	3,202	0,052	1,64%

Tabla 10: Distancia analítica, error y error porcentual para *powerset* 3

Con esta ecuación y para las medidas realizadas, se obtiene un error promedio de 0,25m (15,43%). El error que se obtiene es bastante homogéneo para todas las medidas (desviación estándar de 0,098) con lo cual podemos asumir que las medidas concuerdan con la ecuación analítica propuesta.

3.1.3 *Powerset* 5

A continuación se muestran los datos obtenidos para un nivel de potencia de 5:

Distancia [m]	RSSI
0,315	189
0,63	188,6666667
0,945	184,3333333
1,26	183
1,575	178,6666667
1,89	178,3333333
2,205	177
2,52	175
2,835	172,3333333
3,15	173,3333333
3,465	171,3333333

Distancia [m]	RSSI
3,78	171,6666667
4,095	172
4,41	171,3333333
4,725	172,3333333
5,04	170,3333333
5,355	170

Tabla 11: Distancia vs RSSI para *powerset* 5

Las medidas terminan a una distancia de 5,355 m porque para distancias mayores la comunicación se vuelve irregular, las medidas divergen y se pierde gran cantidad de los mensajes.

Tras graficar los datos obtenidos:

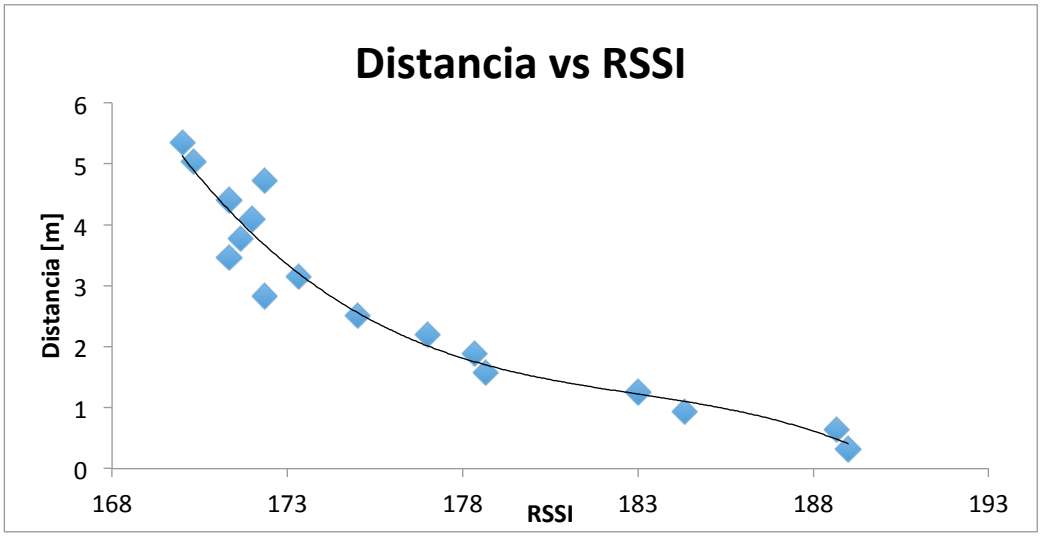

Ilustración 28: Distancia vs RSSI para *powerset* 5

Los puntos siguen una tendencia más o menos clara, que se representa con la línea negra. A continuación se presenta la ecuación analítica fruto de dicha línea (con y la distancia en metros, x el RSSI):

$$y = -0{,}00128684353137221x^3 + 0{,}706126703133589x^2 - 129{,}244544708688x + 7.891{,}90155040081$$

Ecuación 3: Distancia [m] en función del RSSI para *powerset* 5

En base a esto, se obtienen los siguientes resultados:

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
0,315	189	0,39	0,19	58,90%
0,63	189	0,45	0,29	45,25%
0,945	184	1,09	0,16	17,26%
1,26	183	1,22	0,21	16,70%
1,575	179	1,70	0,13	8,20%
1,89	178	1,76	0,13	6,96%
2,205	177	2,07	0,37	16,65%
2,52	175	2,69	0,65	25,63%
2,835	172	3,74	0,91	31,95%
3,15	173	3,26	0,50	15,83%
3,465	171	4,37	0,98	28,39%
3,78	172	4,16	0,96	25,35%
4,095	172	4,18	1,12	27,47%
4,41	171	4,31	0,60	13,72%
4,725	172	3,74	0,98	20,83%
5,04	170	4,96	0,71	14,17%
5,355	170	5,13	0,23	4,23%

Tabla 12: Distancia analítica, error y error porcentual para *powerset* 5

Con esta ecuación y para las medidas realizadas, se obtiene un error promedio de 0,56m (23,03%). Se obtiene un error bastante homogéneo, no obstante se aprecia que es más alto para distancias menores a 3 metros, y más bajo para distancias entre 3 y 5,355 metros, lo que indica que este valor de *powerset* es más adecuado para distancias en ese rango.

3.1.4 *Powerset* 10

A continuación se muestran los datos obtenidos para un nivel de potencia de 10:

Distancia [m]	RSSI
0,315	197
0,63	193
0,945	192

Distancia [m]	RSSI
1,26	192
1,575	187
1,89	183
2,205	180
2,52	180
2,835	181
3,15	179
3,465	179
3,78	175
4,095	175
4,41	173
4,725	173
5,04	174
5,355	173
5,67	173
5,985	173
6,3	173
6,615	172
6,93	172
7,245	172
7,56	172
7,875	170
8,19	170
8,505	170
8,82	168
9,135	169
9,45	167

Tabla 13: Distancia vs RSSI para *powerset* 10

Con este nivel de potencia el rango de distancias es mayor, y se llega hasta 9,45 m de alcance con medidas fiables y comunicación estable.

Mostrando los datos en una gráfica:

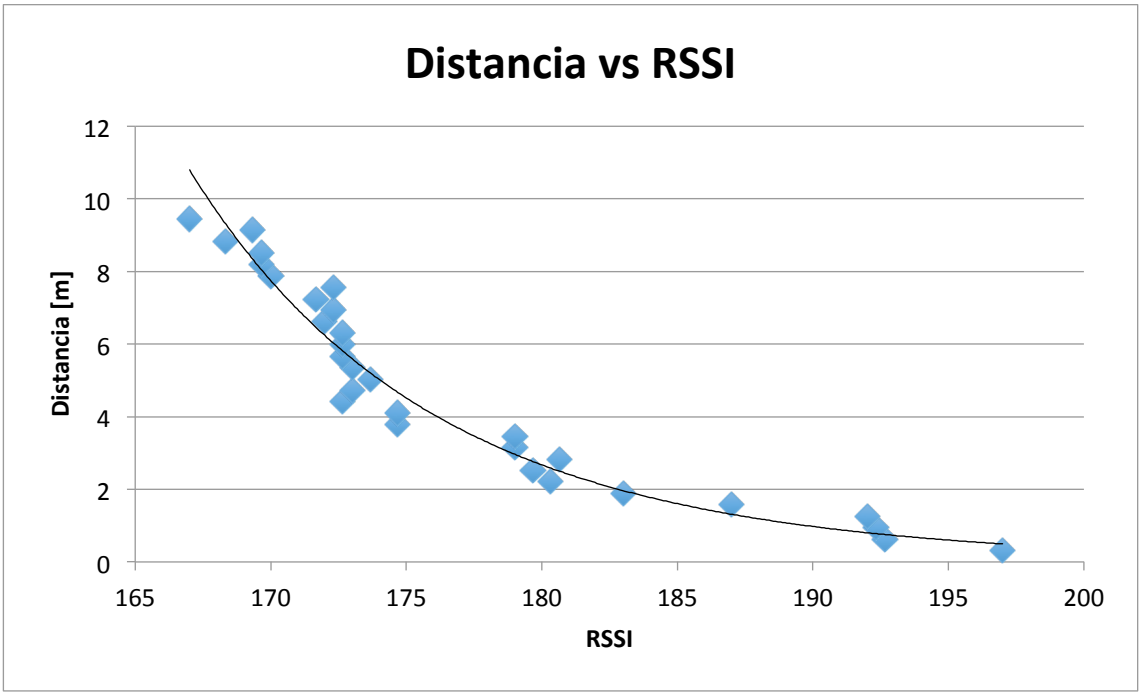

Ilustración 29: Distancia vs RSSI para *powerset* 10

Se obtiene que la ecuación que mejor ajusta a la nube de puntos es (con y la distancia en metros, x el RSSI):

$$y = 2{,}65512924705x\ 10^{54}\ x\text{^} - 18{,}6216804106358$$

Ecuación 4: Distancia [m] en función del RSSI para *powerset* 10

Con estos datos se obtienen datos analíticos y el error que la estimación de la ecuación supone:

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
0,315	197	0,50	0,18	58,11%
0,63	193	0,78	0,17	26,90%
0,945	192	0,79	0,15	16,25%
1,26	192	0,81	0,45	35,99%
1,575	187	1,34	0,25	15,90%
1,89	183	2,15	0,73	38,41%
2,205	180	2,68	0,63	28,69%
2,52	180	2,90	0,75	29,78%
2,835	181	2,50	0,34	11,85%
3,15	179	2,98	0,27	8,49%

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
3,465	179	2,98	0,49	14,07%
3,78	175	4,76	1,06	28,10%
4,095	175	4,73	0,65	15,92%
4,41	173	5,81	1,40	31,72%
4,725	173	5,60	0,87	18,44%
5,04	174	5,26	0,58	11,44%
5,355	173	5,62	0,48	9,02%
5,67	173	5,83	0,59	10,42%
5,985	173	5,86	0,77	12,87%
6,3	173	5,96	1,31	20,83%
6,615	172	6,26	0,58	8,74%
6,93	172	6,02	0,91	13,11%
7,245	172	6,47	0,77	10,66%
7,56	172	6,02	1,54	20,35%
7,875	170	7,78	0,61	7,73%
8,19	170	8,12	1,05	12,83%
8,505	170	8,08	0,62	7,25%
8,82	168	9,33	0,62	7,00%
9,135	169	8,35	0,78	8,58%
9,45	167	10,80	1,35	14,27%

Tabla 14: Distancia analítica, error y error porcentual para *powerset* 10

De los datos se saca que el error medio es de 0,68 [m] (18,55%). Con una desviación típica de 0,117, no hay demasiada dispersión en los datos. Aún así, se aprecia que el error disminuye conforme aumenta la distancia, lo que parece confirmar la hipótesis de que los niveles de potencia mayores son mejores para mayor distancia y viceversa.

3.1.5 *Powerset* 20

A continuación se muestran los datos obtenidos para un nivel de potencia de 20:

Distancia [m]	RSSI
0,315	209
0,63	200
0,945	197
1,26	197

Distancia [m]	RSSI
1,575	193
1,89	195
2,205	195
2,52	185
2,835	183
3,15	189
3,465	186
3,78	183
4,095	185
4,41	181
4,725	184
5,04	179
5,355	183
5,67	180
5,985	179
6,3	180
6,615	179
6,93	178
7,245	173
7,56	181
7,875	182
8,19	178
8,505	178
8,82	175
9,135	177
9,45	175
9,765	173
10,08	174
10,395	171
10,71	170
11,025	172
11,34	171
11,655	169
11,97	173
12,285	173
12,6	174
12,915	171
13,23	172
13,545	174

Distancia [m]	RSSI
13,86	179
14,175	178
14,49	172
14,805	172
15,12	173
15,435	171
15,75	170

Tabla 15: Distancia vs RSSI para *powerset* 20

Llevando estos datos a una gráfica:

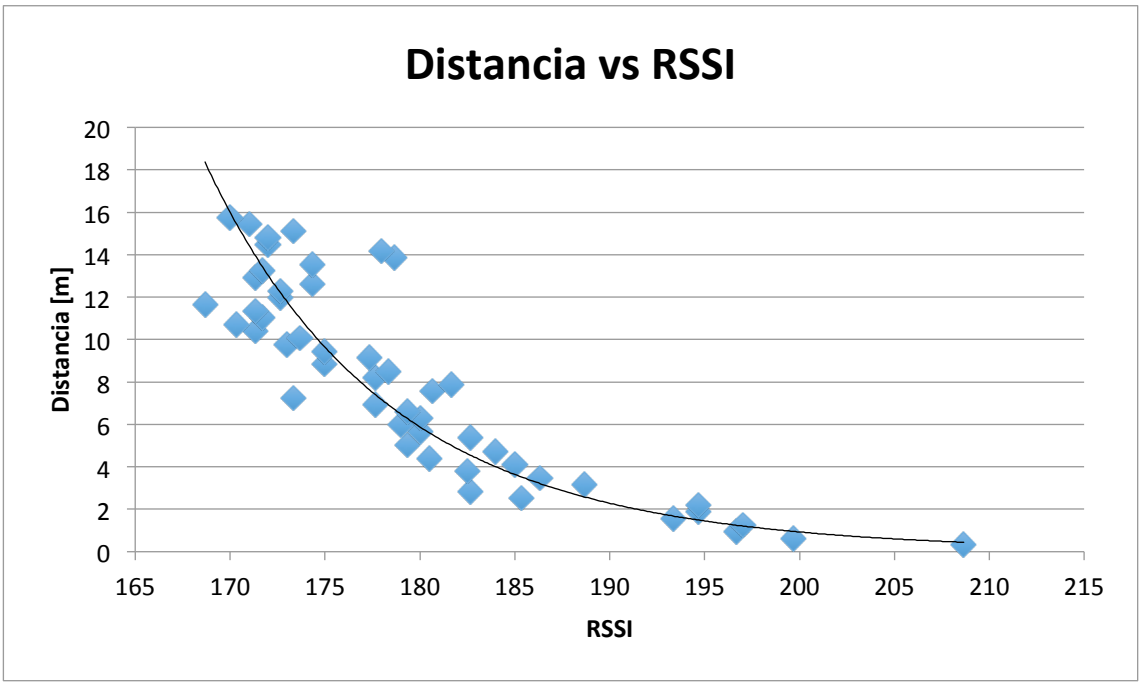

Ilustración 30: Distancia vs RSSI para *powerset* 20

Una vez representados los datos y su tendencia, se ve que siguen la ecuación (con y la distancia en metros, x el RSSI):

$$y = 1{,}80210321417014x10^{40}\ x^{-17{,}5084232112578}$$

Ecuación 5: Distancia [m] en función de la RSSI para *powerset* 20

Con esta ecuación y calculados los valores de distancia para los valores de RSSI obtenidos, se obtiene:

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
0,315	209	0,45	0,13	41,27%
0,63	200	0,96	0,33	52,49%
0,945	197	1,25	0,30	32,17%
1,26	197	1,21	0,09	6,99%
1,575	193	1,69	0,20	12,94%
1,89	195	1,50	0,39	20,50%
2,205	195	1,50	0,70	31,86%
2,52	185	3,54	1,02	40,59%
2,835	183	4,58	1,75	61,61%
3,15	189	2,65	0,61	19,21%
3,465	186	3,30	0,79	22,85%
3,78	183	4,67	0,89	23,56%
4,095	185	3,80	1,05	25,64%
4,41	181	5,67	1,26	28,48%
4,725	184	4,00	0,72	15,28%
5,04	179	6,48	1,57	31,17%
5,355	183	4,82	1,73	32,32%
5,67	180	6,00	1,15	20,33%
5,985	179	6,68	1,46	24,34%
6,3	180	6,28	1,75	27,84%
6,615	179	6,41	1,06	16,01%
6,93	178	7,40	0,47	6,78%
7,245	173	11,40	4,16	57,38%
7,56	181	5,63	1,93	25,51%
7,875	182	5,21	2,66	33,80%
8,19	178	7,50	1,04	12,74%
8,505	178	6,98	1,53	17,97%
8,82	175	10,06	2,79	31,68%
9,135	177	8,55	4,12	45,08%
9,45	175	11,08	4,70	49,71%
9,765	173	11,82	2,06	21,06%
10,08	174	11,18	2,01	19,91%
10,395	171	14,28	3,88	37,34%
10,71	170	15,72	5,01	46,78%
11,025	172	13,77	2,74	24,87%
11,34	171	14,41	3,07	27,11%

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
11,655	169	18,39	6,73	57,78%
11,97	173	12,37	1,96	16,36%
12,285	173	12,89	3,53	28,76%
12,6	174	10,53	2,36	18,77%
12,915	171	14,28	2,87	22,24%
13,23	172	14,04	3,50	26,42%
13,545	174	10,31	3,24	23,88%
13,86	179	6,71	7,15	51,61%
14,175	178	7,18	7,00	49,37%
14,49	172	13,47	3,19	22,00%
14,805	172	13,63	3,13	21,14%
15,12	173	11,75	3,95	26,14%
15,435	171	15,38	4,42	28,67%
15,75	170	16,09	1,65	10,48%

Tabla 16: Distancia analítica, error y error porcentual para *powerset* 20

Con estos valores, el error medio obtenido es de 2,37 [m] (29,32%). A pesar de ser un error bastante alto, la desviación estándar es bastante. baja (0,14), lo cual indica que los datos reales se ajustan bastante a la curva de tendencia.

3.1.6 *Powerset* 30

A continuación se presentan los datos obtenidos para un nivel de potencia de 30:

Distancia [m]	RSSI
0,315	205
0,63	198
0,945	199
1,26	197
1,575	194
1,89	195
2,205	190
2,835	194
3,15	190
3,465	187

Distancia [m]	RSSI
3,78	189
4,095	186
4,41	186
4,725	186
5,04	186
5,355	187
5,67	187
5,985	183
6,3	185
6,615	185
6,93	184
7,245	185
7,56	185
7,875	182
8,19	179
8,505	182
8,82	178
9,135	177
9,45	178
9,765	176
10,08	174
10,395	177
10,71	175
11,025	173
11,34	177
11,655	180
11,97	176
12,285	176
12,6	175
12,915	175
13,23	174
13,86	173
14,175	176
14,49	173
14,805	173
15,12	171
15,435	172
15,75	172
16,065	172

Distancia [m]	RSSI
16,38	172
16,695	170
17,01	171
17,325	171
17,64	170
17,955	170

Tabla 17: Distancia vs RSSI para *powerset* 30

Representando los datos en una gráfica:

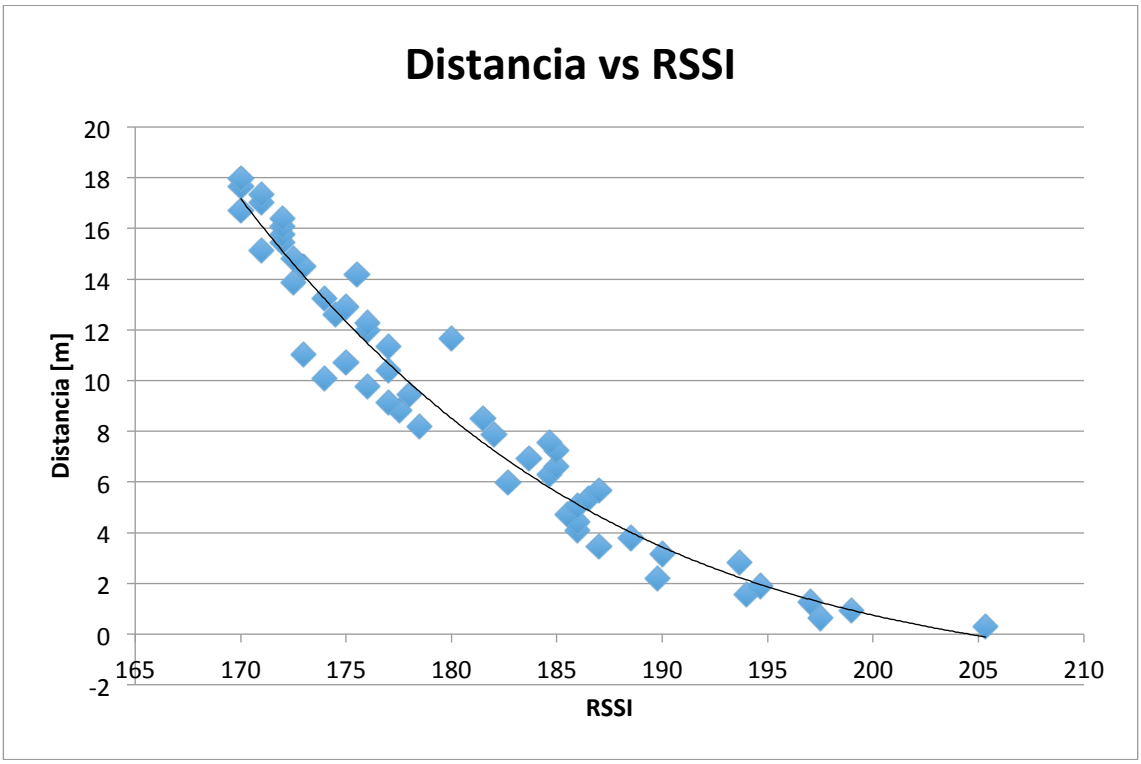

Ilustración 31: Distancia vs RSSI para *powerset* 30

Observamos que los datos se ajustan a la curva de tendencia definida por la ecuación (con y la distancia en metros, x el RSSI):

$$y = -3{,}021289498734E-4\,x^3 + 1{,}84416661355944x10^{-1}\,x^2 - 3{,}76960857787379 * 10^1\,x + 2{,}5804235066942x10^3$$

Ecuación 6: Distancia [m] en función del RSSI para *powerset* 30

Al comparar las medidas obtenidas a partir de esta ecuación con las medidas reales se obtiene:

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
0,315	205	-0,10	0,42	132,33%
0,63	198	1,34	0,71	112,55%
0,945	199	1,03	0,08	8,95%
1,26	197	1,44	0,21	16,84%
1,575	194	2,16	0,59	37,16%
1,89	195	1,96	0,08	4,50%
2,205	190	3,39	1,19	53,92%
2,835	194	2,23	0,61	21,38%
3,15	190	3,32	0,34	10,75%
3,465	187	4,44	0,97	28,11%
3,78	189	3,87	0,57	14,99%
4,095	186	5,02	1,36	33,20%
4,41	186	5,01	1,33	30,24%
4,725	186	5,14	0,70	14,83%
5,04	186	4,91	0,54	10,63%
5,355	187	4,69	0,67	12,47%
5,67	187	4,44	1,23	21,71%
5,985	183	6,68	1,43	23,96%
6,3	185	5,52	0,78	12,40%
6,615	185	5,34	1,27	19,26%
6,93	184	6,04	0,90	12,97%
7,245	185	5,34	1,90	26,28%
7,56	185	5,51	2,05	27,15%
7,875	182	6,97	0,91	11,52%
8,19	179	9,32	1,13	13,81%
8,505	182	7,29	1,21	14,24%
8,82	178	10,18	1,97	22,30%
9,135	177	10,65	2,43	26,65%
9,45	178	10,01	3,07	32,45%
9,765	176	11,26	1,50	15,35%
10,08	174	13,11	3,03	30,04%
10,395	177	10,65	2,43	23,42%
10,71	175	12,17	1,46	13,65%
11,025	173	14,07	3,04	27,61%
11,34	177	10,54	1,34	11,79%
11,655	180	8,21	3,44	29,54%

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)
11,97	176	11,49	2,58	21,56%
12,285	176	11,29	1,00	8,11%
12,6	175	12,67	1,40	11,13%
12,915	175	12,15	0,77	5,95%
13,23	174	13,08	0,15	1,13%
13,86	173	14,65	1,57	11,30%
14,175	176	11,70	2,47	17,43%
14,49	173	14,10	1,02	7,01%
14,805	173	14,76	2,61	17,64%
15,12	171	16,21	1,09	7,22%
15,435	172	15,13	0,82	5,33%
15,75	172	15,23	2,15	13,62%
16,065	172	15,13	1,03	6,42%
16,38	172	15,11	1,27	7,74%
16,695	170	17,37	0,68	4,05%
17,01	171	16,23	1,02	5,99%
17,325	171	16,21	1,11	6,42%
17,64	170	17,40	1,19	6,74%
17,955	170	17,37	0,58	3,25%

Tabla 18: Distancia analítica, error y error porcentual para *powerset* 30

Los datos analíticos presentan un error medio de 1,24 m (24,11%) con una desviación estándar de 0,23. A pesar de tener un error absoluto relativamente grande, los datos ajustan bastante bien a la línea de tendencia propuesta.

Se observa que el error relativo disminuye a medida que aumenta la distancia, haciéndose particularmente pequeño a partir de 15 metros, lo que induce a pensar que un nivel de potencia tan alto es adecuado para distancias largas, mientras que la precisión disminuye y las medidas divergen para distancias cortas, como era de esperar.

3.3.7 Conclusiones

La conclusión más importante es que el comportamiento del RSSI sigue una tendencia clara y modelable. Además, se observa que, como era de esperar, las medidas para niveles de potencia más bajos son más exactas a distancias más cortas y para niveles más altos, a distancias más largas.

Así y viendo los datos obtenidos, se puede sacar una relación de valores idóneos de *powerset* según la distancia a la que se encuentre el objeto.

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)	*Powerset*
0,05	189	0,05	0,00	0,00%	2
0,1	183	0,09	0,01	13,82%	2
0,2	173	0,21	0,07	35,08%	2
0,3	172	0,31	0,06	18,92%	2
0,315	185	0,30	0,01	3,45%	3
0,630	179	0,65	0,16	25,94%	3
0,945	177	0,96	0,16	16,56%	3
1,260	175	1,26	0,08	6,06%	3
1,575	174	1,37	0,33	21,18%	3
1,890	172	2,27	0,54	28,74%	3
2,205	171	2,34	0,16	7,11%	3
2,520	172	2,13	0,57	22,50%	3
2,835	169	2,74	0,34	11,88%	3
3,150	168	3,20	0,05	1,64%	3
3,15	173	3,26	0,50	15,83%	5
3,465	171	4,37	0,98	28,39%	5
3,78	172	4,16	0,96	25,35%	5
4,095	172	4,18	1,12	27,47%	5
4,41	171	4,31	0,60	13,72%	5
4,725	172	3,74	0,98	20,83%	5
5,04	170	4,96	0,71	14,17%	5
5,355	170	5,13	0,23	4,23%	5
5,67	173	5,83	0,59	10,42%	10
5,985	173	5,86	0,77	12,87%	10
6,3	173	5,96	1,31	20,83%	10
6,615	172	6,26	0,58	8,74%	10
6,93	172	6,02	0,91	13,11%	10
7,245	172	6,47	0,77	10,66%	10
7,56	172	6,02	1,54	20,35%	10
7,875	170	7,78	0,61	7,73%	10
8,19	170	8,12	1,05	12,83%	10
8,505	170	8,08	0,62	7,25%	10
8,82	168	9,33	0,62	7,00%	10
9,135	169	8,35	0,78	8,58%	10

Distancia real [m]	RSSI	Distancia analítica [m]	Error	Error (%)	*Powerset*
9,45	167	10,80	1,35	14,27%	10
9,765	173	11,82	2,06	21,06%	20
10,08	174	11,18	2,01	19,91%	20
10,395	177	10,65	2,43	23,42%	30
10,71	175	12,17	1,46	13,65%	30
11,025	173	14,07	3,04	27,61%	30
11,34	177	10,54	1,34	11,79%	30
11,655	180	8,21	3,44	29,54%	30
11,97	176	11,49	2,58	21,56%	30
12,285	176	11,29	1,00	8,11%	30
12,6	175	12,67	1,40	11,13%	30
12,915	175	12,15	0,77	5,95%	30
13,23	174	13,08	0,15	1,13%	30
13,86	173	14,65	1,57	11,30%	30
14,175	176	11,70	2,47	17,43%	30
14,49	173	14,10	1,02	7,01%	30
14,805	173	14,76	2,61	17,64%	30
15,12	171	16,21	1,09	7,22%	30
15,435	172	15,13	0,82	5,33%	30
15,75	172	15,23	2,15	13,62%	30
16,065	172	15,13	1,03	6,42%	30
16,38	172	15,11	1,27	7,74%	30
16,695	170	17,37	0,68	4,05%	30
17,01	171	16,23	1,02	5,99%	30
17,325	171	16,21	1,11	6,42%	30
17,64	170	17,40	1,19	6,74%	30
17,955	170	17,37	0,58	3,25%	30

Tabla 19: Distancia analítica, RSSI, error y error porcentual

De esta forma conseguimos un error promedio de 0,99 m (14%) con una desviación estándar de 0,082, lo cual parece ser la mejor forma de estimar la distancia en función del RSSI. El nivel de potencia será dinámico y deberá variar en función de la distancia a la que se encuentren las motas.

Así, se utilizará:

- Nivel 2 hasta 0,315 m.
- Nivel 3 hasta 1,6 m.
- Nivel 5 entre 1,6 y 3,15 m.

- Nivel 10 entre 5,355 y 9,45 m.
- Nivel 20 entre 9,45 y 10,08 m.
- Nivel 30 a partir de 10,08 m.

De esta forma, la distancia en función del RSSI queda representada como (con y la distancia en metros, x el RSSI):

$y = 0{,}000053142689442609\,x^4 - 0{,}038677983722572\,x^3 + 10{,}5531928011355\,x^2 - 1.279{,}35492439\,x + 58.143{,}4712968649$	y ≤ 0,315 m (ps 2)
$y = 0{,}000053142689442609\,x^4 - 0{,}038677983722572\,x^3 + 10{,}5531928011355\,x^2 - 1.279{,}35492439\,x + 58.143{,}4712968649$	0,315 < y ≤ 1,6 m (*ps* 3)
$y = 0{,}00732231920199501\,x^2 - 2{,}727286159601\,x + 254{,}231685785536$	1,6 < y ≤ 3,15 m (*ps* 3)
$y = -0{,}00128684353137221\,x^3 + 0{,}706126703133589\,x^2 - 129{,}244544708688\,x + 7.891{,}90155040081$	3,15 < y ≤ 5,355 m (*ps* 5)
$y = 2{,}655129124705x10^{54}\ x^{-18{,}6216804106358}$	5,355 < y ≤ 9,45 m (*ps* 10)
$y = 1{,}80210321417014x10^{40}x^{-17{,}508423112578}$	9,45 < y ≤ 10,08 m (*ps* 20)
$y = -3{,}021289498734E-4\,x^3 + 1{,}84416661355944x10^{-1}\,x^2 - 3{,}76960857787379 * 10^1\,x + 2{,}5804235066942x10^3$	y > 10,08 m (*ps* 30)

Ecuación 7: Distancia [m] en función del RSSI para un nivel de potencia variable

3.2 Recepción masiva de mensajes

Estas pruebas tienen como objetivo determinar la frecuencia máxima de envío de mensajes que permite la aplicación, tanto con un nodo como con varios, y asimismo cuantificar la pérdida de mensajes por encima de esa frecuencia, ya que si se observa una pérdida sistemática de mensajes se podría arreglar enviando el mismo mensaje varias veces en cada iteración.

Seguidamente se hará lo mismo imprimiendo información por pantalla, y para varios nodos (hasta 4).

Para comprobar si se reciben todos los mensajes, se leerá una variable que el nodo envía y que se incrementa en cada envío de mensaje.

Número de nodos	*printf*	Frecuencia límite	Comentarios
1	No	38 Hz	Para frecuencias más altas, la aplicación colapsa
1	Sí	34 Hz	
4	No	31 Hz	
4	Sí	30 Hz	

Después de estas pruebas, se observa que, tanto para uno como para varios nodos, la frecuencia es bastante superior a la necesaria para la aplicación planteada (en el peor de los casos, se podrían enviar 30 mensajes por segundo sin impactar en el funcionamiento de la aplicación) ya que ni se va a mover tanto volumen de datos como para necesitar frecuencias tan altas, ni es óptimo para el consumo energético de los nodos. Las pruebas y el desarrollo normal de la aplicación se han hecho pensando en frecuencias entorno a los 5-10Hz.

También se observa que, a pesar de ser una función relativamente lenta, el hecho de imprimir o no información por pantalla no afecta demasiado a la frecuencia límite de envío, aunque afecta lo suficiente como para tenerlo en cuenta para otro tipo de aplicaciones que utilicen esta tecnología.

Capítulo 4

Presupuesto, conclusiones y líneas futuras

En este capítulo se hará una estimación del presupuesto del proyecto ayudándonos de la plantilla que aparece en la web de la universidad [20], conclusiones, experiencias y dificultades a la hora de desarrollar el proyecto e ideas sobre líneas futuras.

4.1 Presupuesto

De acuerdo con la plantilla proporcionada por la Universidad [20] se debe desglosar el coste en: costes de personal, equipos, subcontratación de tareas y otros costes. En el caso de este proyecto sólo aplican los costes de personal, de equipo y otros costes directos.

4.1.1 Coste de personal

De acuerdo con la tarifa propuesta en la plantilla anteriormente mencionada, el coste de personal es:

Apellidos y nombre	**Categoría**	**Dedicación (hombres mes)** [a)]	**Coste hombre mes**	**Coste (Euro)**
Castán Artal, Miguel	Ingeniero	4	2694,39	10.777,56 €
Cañoto López, Daniel	Ingeniero	4	2694,39	10.777,56 €
García Armada, Ana	Ingeniero Senior	1	4289,54	4.289,54 €
	Hombres mes	**9**	**Total**	25.844,66 €

Tabla 20: Presupuesto de personal del proyecto

Con una dedicación equivalente a 4 meses a jornada completa (estimación de 131,25 h por mes) de los estudiantes involucrados en ambas partes del proyecto y de un mes de la tutora del proyecto, sale un total de 25.844,66 € de gastos de personal. Evidentemente este coste es de desarrollo y se debería repartir entre las unidades de producto producidas.

4.1.2 Coste de equipos

Concepto	Coste unitario	Período de dedicación	% de dedicación durante el período	Período de depreciación	Coste (Euro)
Ordenador Packard Bell	800,00 €	4	50%	60	26,67 €
Ordenador Apple	1.000,00 €	4	50%	60	33,33 €
Kit de motas MicaZ	250,00 €	4	100%	60	16,67 €
				TOTAL	**76,67 €**

Tabla 21: Presupuesto de equipos

4.1.3 Otros costes directos

Concepto	Coste (Euro)
Material de oficina	100 €
Electricidad	150 €
Baterías	10€
TOTAL	**260 €**

Tabla 22: Costes indirectos

4.1.4 Costes totales

Concepto	Coste (Euro)
Personal	25.844,66 €
Equipos	76,67 €
Otros costes directos	260 €
TOTAL	**26.181,33 €**

Tabla 23: Costes totales

Estos costes reflejan todo el material y personal necesario para el desarrollo del prototipo. Si este proyecto siguiera adelante y se comercializara, se debería establecer un número mínimo de unidades a producir entre las cuales se repartiría este coste de desarrollo para que el producto resultara rentable pero sin elevar demasiado el precio del mismo.

4.2 Conclusiones

Inicialmente, el objetivo de este proyecto era simple y se podría resumir en una frase: "Ayudar a encontrar". Algo tan cotidiano y molesto como es perder objetos personales y sobre lo que se ha desarrollado mucho, pero todavía no se ha encontrado una solución definitiva, cómoda, asequible y abierta.

Fue con esta idea con la que se empieza a investigar los actuales sistemas de comunicación orientados a este fin, tratando de buscar la tecnología que más se ajustara a las necesidades del proyecto, tratando de que fuera algo diferente a las distintas alternativas del mercado.

Así, tras descartar otras tecnologías por diversos motivos (necesidad de puntos fijos, mala precisión en interiores) nos decantaremos por Zigbee. Una tecnología en auge, abierta, que hace uso de frecuencias abiertas y sobre todo, que da mucha libertad para poder desarrollar y trabajar sobre ella.

Tecnologías como el GPS ya cubren la localización en largas distancias y seguimiento mediante una red de satélites. El objetivo de este proyecto no es suplir esta función ya existente sino complementarla con dos puntos clave los cuales hoy en día todavía no están cubiertos por la geolocalización: Precisión en distancias cortas y localización en interiores.

Además de todo lo anterior, la forma de trabajar con Zigbee en este proyecto permite establecer redes *ad hoc*, es decir, no necesita de una infraestructura fija para establecer una red.

Esto conlleva algunas dificultades, como la imposibilidad de conocer la posición relativa de los nodos de la red (solamente pudiendo estimar la distancia) por la ausencia de nodos fijos de referencia. Además de que la precisión de dicha distancia disminuye sin una infraestructura de nodos fijos.

En este punto se vuelve necesario redefinir el concepto del proyecto. Pasar de "Ayudar a encontrar" a "Evitar perder". Ayudar a que el usuario se anticipe a la pérdida. Y esta es la idea que perdura y define el proyecto.

Para el usuario el prototipo desarrollado ofrece dos funciones: Anticipación a pérdida y localización.

La primera no implica que se avise al usuario cuando el nodo queda fuera del alcance de la red, eso dificultaría su localización una vez perdido, sino en avisar al usuario antes de que la conexión con la red se pierda, facilitando así su localización.

La localización se basa en señales luminosas que permiten la localización del objeto. En principio la localización se basó únicamente en medir la distancia a la que se encuentra el nodo perdido, con idea de que el usuario se acerque a él y lo recupere, pero en la práctica, al manejar distancias relativamente cortas, se comprueba que un buen complemento a esto es la emisión de señales luminosas o acústicas que permiten al usuario localizar el objeto cómoda y rápidamente.

A la hora de desarrollar la aplicación el principal problema que aparece es la falta de soporte tanto para el sistema TinyOS como para el lenguaje nesC. Si bien el proceso de instalación del entorno de desarrollo se puede encontrar en su página web [21] y parece relativamente sencillo, la operativa se complica ya que muchos de los enlaces a paquetes necesarios no se encuentran en el repositorio oficial de TinyOS y son difíciles de encontrar.

A la hora del desarrollo en nesC, a pesar de que el lenguaje es altamente parecido a C, surgen varios problemas debido a la peculiar estructura que presenta y a la falta de material de soporte.

En resumen y evaluando los objetivos planteados inicialmente, se puede considerar que se cumplen: el prototipo desarrollado es funcional, permite la comunicación con varios nodos y la estimación de la distancia a la que se encuentran con el modelo matemático desarrollado.

4.3 Líneas futuras

Este prototipo utiliza la familia MicaZ, proporcionada por la Universidad, que si bien han servido perfectamente para este desarrollo, no son prácticas por su tamaño para el producto final, para el cual se debería utilizar otro tipo de motas, bien una

familia existente, o bien algún desarrollo hecho a medida, que contenga y potencie sólo las características usuales de las motas utilizadas en este proyecto.

El caso más interesante sería el de desarrollar una mota propia, ya que Zigbee es una tecnología abierta, y en el caso de producción en masa, el coste unitario disminuye mucho, con lo que el desarrollo se amortizaría rápidamente, y se podría mejorar en prototipo actual disminuyendo el tamaño y el consumo.

Por otro lado, en este caso el nodo central, o nodo base, es un ordenador portátil, en futuros desarrollos esto se debería portar a una aplicación móvil complementada con accesorios Zigbee para el dispositivo.

En cuanto a eficiencia energética y precisión, se debería dinamizar el nivel de potencia de envío de mensajes y la frecuencia de envío. Aumentar la frecuencia y disminuir el nivel de potencia para distancias cortas mejoraría la precisión de la estimación de distancia y disminuiría el consumo al enviar menor número de mensajes a menor nivel de potencia.

Además de todo esto, el servidor de comunicación es accesible remotamente, aunque en este proyecto se utilice a nivel local, el servidor es accesible por internet, lo cual da un amplio margen de posibilidades y nuevas aplicaciones, ya que el usuario no tiene que estar físicamente en el lugar en el que se encuentren los nodos que se desea controlar.

En conclusión y para terminar, a pesar de que son amplias las posibilidades que este proyecto ofrece, se ven limitadas por el tamaño y el consumo del prototipo, que es lo primero que se debería tratar de reducir en desarrollos futuros, tanto a nivel software como a nivel hardware.

ANEXO I

Estructura de los mensajes

Los mensajes que se envían en aplicaciones que utilizan el entorno TinyOS tienen una estructura genérica [19] con una serie de campos fijos y que no deben ser modificados, y una parte que es configurable por el usuario.

Además de esta parte, que es la "visible" por el programa, existe una meta estructura de mensaje en la que se incluye una cabecera, el cuerpo del mensaje (la parte "visible"), códigos CRC y de comprobación el final del mensaje. Esta parte está desarrollada de acuerdo con el protocolo de comunicación Zigbee y no puede ser modificada.

Analizando los mensajes recibidos por la aplicación *BaseStationMod*:

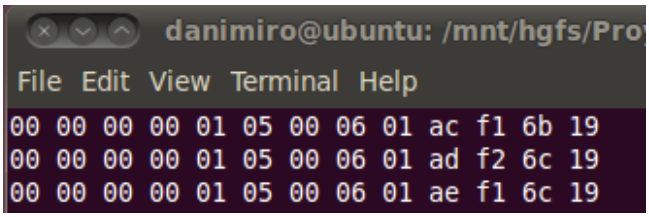

Ilustración 32: Mensajes recibidos por la aplicación *BaseStationMod*

Se observa que el mensaje se compone de varios campos en hexadecimal. Si se observa el último mensaje y se pasa a sistema decimal:

00 00 00 00 01 05 00 06 | 01 174 241 108 25

Se observa que, para nuestro proyecto, el mensaje consta de dos partes diferenciadas:

- Estructura genérica de mensaje (azul): Esta parte no es modificable (no sin al menos modificar varios parámetros del sistema y de la forma de transmisión). Se divide en:
  - **Inicio del mensaje** (1 byte): Este campo es siempre 0.
  - **ID de destino** (2 bytes): El ID del nodo al que va dirigido el mensaje. Este valor se asigna cuando se instala una aplicación, y el valor por omisión es 0. Si se envía un mensaje a la dirección *NULL*, se envía a todos los nodos del grupo que se especifique.
  - **ID remitente** (2 bytes): La ID del nodo que envía el mensaje.

- **Longitud de mensaje** (1 byte): Longitud de la *payload* ("área de carga") del mensaje enviado.
- **ID de grupo** (1 byte): Además de tener cada nodo su ID, pueden estar agrupados. Para este proyecto siempre se utiliza el mismo grupo.
- **Tipo de mensaje** (1 byte): Define el tipo de mensaje a enviar. No se utiliza en este proyecto, pero esto es utilizado normalmente en aplicaciones de comunicación para filtrar mensajes.

- *Payload* o "área de carga" (roja): Es el área de mensaje configurable por el usuario. Puede tener hasta 28 bytes. En este proyecto tiene los siguientes campos:
  - **ID remitente** (1 byte): Es el mismo campo que en la estructura general del mensaje.
  - **Contador** (1 byte): Se utiliza como un contador auxiliar de mensajes para detectar pérdidas de mensajes en la recepción de los nodos a la base, y para especificar el número a mostrar en los leds de los nodos en el envío desde la base.
  - ***RSSI (Received Signal Strenght Indicator*, Indicador de potencia de señal recibida*)*** (1 byte): Indicador de la potencia de la señal del mensaje recibido.
  - ***LQI (Link Quality Indicator, Indicador de Calidad de conexión)*** (1 byte): Indicador de la calidad de la conexión de la base con el nodo correspondiente.
  - ***Powerset* (*PS,* nivel de potencia de mensaje enviado)** (1 byte): Se utiliza para establecer el nivel de potencia de mensaje enviado por la base; dicho nivel se envía a los nodos para que establezcan su nivel de potencia acorde con la base.

Así, el mensaje de ejemplo se interpreta como:

Inicio	ID destino	ID Remitente	Long. *payload*	ID Grupo	Tipo msg	ID Remitente	Contador	*RSSI*	*LQI*	*Powerset*
00	00 00	00 01	05	00	06	01	174	241	108	25

ANEXO II

Código de la función sfsend

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "sfsource.h"

void send_packet(int fd, char **bytes, int count)
{
 int i;
 unsigned char *packet;

 packet = malloc(count);
 if (!packet)
  exit(2);

 for (i = 0; i < count; i++)
  packet[i] = strtol(bytes[i], NULL, 0);

 fprintf(stderr,"Sending ");
 for (i = 0; i < count; i++)
  fprintf(stderr, " %02x", packet[i]);
 fprintf(stderr, "\n");

 write_sf_packet(fd, packet, count);
}

int main(int argc, char **argv)
{
 int fd;

 if (argc < 4)
  {
   fprintf(stderr, "Usage: %s <host> <port> <bytes> - send a raw packet to a serial forwarder\n", argv[0]);
   exit(2);
  }
 fd = open_sf_source(argv[1], atoi(argv[2]));
 if (fd < 0)
  {
   fprintf(stderr, "Couldn't open serial forwarder at %s:%s\n",
           argv[1], argv[2]);
```

```
    exit(1);
  }

 send_packet(fd, argv + 3, argc - 3);
 close(fd);
}
```

Se resalta en negrita la transformación necesaria de los datos para que puedan ser enviados.

ANEXO III

Código de las aplicaciones

A continuación se presenta el código de la aplicación de usuario, que se divide en 3 módulos: de aplicación (con los archivos *aplicacion.c* y *aplicación.h*), módulo *send.c* y módulo *test*

Código de *aplicación.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "aplicacion.h"
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

void main (int argc, char *argv[])
{

	int status,flag;
	pid_t pid,pidsf,pidlisten;

	if(argc<3)
	{
		printf("Opciones: \n(arg1) t para test, s para send\n(arg2) l para listen, o para lost\n");
		exit(1);
	}
	pidsf=fork();
	switch(pidsf)
	{
	case 0:
		serialforwarder();
	case -1:
		perror("fork pidsf");
		exit(EXIT_FAILURE);
	default:
		sleep(2);
		switch(fork())
		{
		case 0:
			if (argv[1][0]=='t')		test();
			if (argv[1][0]=='s')		sendtomote();
			exit(1);
		case -1:
			perror("Error fork sendtomote");
			exit(EXIT_FAILURE);
		default:
```

```
                        if (argv[2][0]=='o')        lost();
                        if (argv[2][0]=='l')        listening();
                        exit(1);
                }
        }

}
```

Código de *aplicación.h*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "sfsource.h"
#include "sfsource.c"
#include <math.h>


void serialforwarder()
{

        if(execlp("gnome-terminal","gnome-terminal","--title=TRAFFIC                    LOG","--
geometry=40x12+1000+0","--execute","sf/sf","9002", "/dev/ttyUSB1","micaz", (char *)0)==-1)
//      if(execle("gedit","gedit",NULL,NULL))
        {
                perror("Error opening serial forwarder");
                exit(EXIT_FAILURE);
        }

}

void sendtomote()
{
        if(execlp("gnome-terminal","gnome-terminal","--title=SEND2MOTE","--
geometry=40x12+1000+270","--execute","./send",(char *)0)==-1)
        {
                perror("Error opening send");
                exit(EXIT_FAILURE);
        }
}

void test()
{
        if(execlp("gnome-terminal","gnome-terminal","--title=SEND2MOTE","--
geometry=40x12+1000+270","--execute","./test",(char *)0)==-1)
        {
                perror("Error opening send");
                exit(EXIT_FAILURE);
```

```
        }
}


long double dist(int RSSI2, int powerset2)
{
long double distancia;


if(powerset2 == 2) distancia = 0.000053142689442609*pow(RSSI2,4) - 0.038677983722572*pow(RSSI2,3) + 10.5531928011355*pow(RSSI2,2) - 1279.35492439*RSSI2 + 58143.4712968649;
//y = 0,000053142689442609x4 - 0,038677983722572x3 + 10,5531928011355x2 - 1279,35492439x + 58143,4712968649
if((powerset2 == 3) && (RSSI2 >= 174)) distancia = (0.00732231920199501*powl(RSSI2,2) - 2.727286159601*RSSI2 + 254.231685785536);
// y = 0,00732231920199501x2 - 2,727286159601x + 254,231685785536
if ((powerset2 == 3) && (RSSI2 < 173)) distancia = ((powl(RSSI2,3)*-0.0108932308035641) + 5.59840418767656*(powl(RSSI2,2)) - 959.229369000047*RSSI2 + 54796.0675445303);
// y = -0,0108932308035641x3 + 5,59840418767656x2 - 959,229369000047x + 54.796,0675445303
if (powerset2 == 5) distancia = (-0.00128684353137221*powl(RSSI2,3) + 0.706126703133589*powl(RSSI2,2) - 129.244544708688*RSSI2 + 7891.90155040081);
// y = -0,00128684353137221x3 + 0,706126703133589x2 - 129,244544708688x + 7.891,90155040081
if (powerset2 == 10) distancia = (2.655129124705*powl(10,54)*powl(RSSI2,-18.6216804106358));
// y = 2,655129124705E+54 x-18,6216804106358
if (powerset2 == 20) distancia = (1.80210321417014*powl(10,40)*powl(RSSI2,-17.5084232112578));
// y = 1,80210321417014E+40 x-17,5084232112578
if (powerset2 == 30) distancia = -0.000302128949873 4*powl(RSSI2,3);// + 0.184416661355944*powl(RSSI2,2) – 37.6960857787379*RSSI2 + 2580.4235066942
// y = -3,021289498734E-4 x3 + 1,84416661355944E-1 x2 – 3,76960857787379E+1 x + 2,5804235066942E+3
else distancia == 99;

return distancia;
}

void imprimedistancia(int len, char* packet)
{
int i;

        for (i = 0; i < len; i++) printf("%02x ", packet[i]);
}

int estimarssi(int *a, int num_elements)
{// Calculamos la media de 5 medidas
        int i;
        int resul=0;
        for(i=0;i<num_elements;i++) resul=resul+a[i];
        resul/=num_elements;
        for (i = 0; i < num_elements; i++) printf("%d ", a[i]);

return resul;
```

```
}




void listening()
{
        int fd;
        fd = open_sf_source("localhost", 9002);
        float distan,dista[5];
        int rssimedia,rssiarray[5];
        unsigned char *packet;
        int len, i=0;

        if (fd < 0)
        {
                fprintf(stderr, "Couldn't open serial forwarder at %s:%s\n","localhost", "9002");
                exit(1);
        }

                while(1)
                {

                        packet = read_sf_packet(fd, &len);
                        if (!packet) exit(0);


                        rssiarray[i]=(int)packet[10]-45;
                        i++;

                        if(i==5)
                        {

                        rssimedia=(int)estimarssi(rssiarray,5);
                        distan=(float)dist(rssimedia,(int)packet[12]);
                        imprimedistancia(len,packet);
                        printf("PS %d RSSI %d Mota %d: %f
m\n",(int)packet[12],rssimedia,packet[8],distan);
                        i=0;
                        }
                        free((void *)packet);

                }
        close(fd);
}

void lost()
{
        int fd,flag=0,perdidas[12]={0,0,0,0,0,0,0,0,0,0,0,0};


        fd = open_sf_source("localhost", 9002);
```

```
	if (fd < 0)
	{
		fprintf(stderr, "Couldn't open serial forwarder at %s:%s\n","localhost", "9002");
		exit(1);
	}

		while(1)
		{
			int len, i;
			unsigned char *packet=read_sf_packet(fd, &len);

			if (!packet)
			exit(0);
			//los límites de "in" y "out" son distintos para evitar conflictos
			if((((int)packet[10]-45)<180) && !perdidas[packet[8]])
			{
				perdidas[packet[8]]=1;
				printf("\a \b \7");
				printf("La mota número %d se perdió\a\n",packet[8]);
			}

			if((((int)packet[10]-45)>185) && perdidas[packet[8]])
			{
				perdidas[packet[8]]=0;
				printf("La mota número %d volvió\a\n",packet[8]);
			}
			free((void *)packet);


		}
	close(fd);
}
```

Código de *send.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "sfsource.h"
#include "sfsource.c"



void enviar (int modo);
void controlador (int);

void main(void)
```

```
{
	printf("Bienvenid@,\n");

	switch(fork()) // fork 1 send
	{
	case 0:
		while(1)
		{
			enviar(0);
		}
	case -1:
		perror("Error sending fork 1 send");
		exit(EXIT_FAILURE);

	default:
		while(1)
		{
			enviar(1);
		}
	}


}



void enviar (int modo)
{
	int fd, i=1;
	int port=9002;
	int contador=0;
	int status;
	unsigned char *packetf;
	char moteid[2], count[2],longitud[2];
	int length;
	length=13;
	longitud[0]=(char)length-8+48;
	longitud[1]='\0';
	useconds_t periodo=100000;
	if(modo==0)
	{
		printf("Número de la mota (z para salir): \n");
		moteid[0]=getchar();
		moteid[1]='\0';
		getchar();
		if(moteid[0]=='z')
		{
			kill(getppid(),SIGUSR1);
			exit(1);
		/* Para salir y no dejar ningún proceso abierto, se lee el ID del padre y se
		envía señal de KILL, con lo cual este y todos sus hijos terminan */
		}
		if(moteid[0]=='0') moteid[0]=NULL;
```

```
		printf("Número a mostrar en los LEDS: \n");
		count[0]=getchar();
		count[1]='\0';
		printf("\e[1;1H\e[2J");
	}

	else
	{
		usleep(periodo);
		moteid[0]='0';
		moteid[1]='\0';
		count[0]='8';
		count[1]='\0';
	}


		switch(fork())
		{
		case 0:
			for(i=0;i<2;i++)
			{
				packetf=malloc(length);

				if(fd = open_sf_source("localhost", port)<0)
				{
					perror("Couldn't open serial forwarder at localhost:port");
					exit(EXIT_FAILURE);
				}

				packetf[0] = strtol("0", NULL, 0);
				packetf[1] = strtol("0", NULL, 0);
				packetf[2] = strtol(moteid, NULL, 0);//nodeid
				packetf[3] = strtol("0", NULL, 0);
				packetf[4] = strtol("0", NULL, 0);
				packetf[5] = strtol(longitud, NULL, 0);//número de bytes de la
payload
				packetf[6] = strtol("0", NULL, 0);
				packetf[7] = strtol("6", NULL, 0);
				packetf[8] = strtol("0", NULL, 0);//sending id
				packetf[9] = strtol(count, NULL, 0);//count
				packetf[10] = strtol("0", NULL, 0);//rssi
				packetf[11] = strtol("0", NULL, 0);//lqi
				packetf[12] = strtol("0", NULL, 0);//powerset

				if (write_sf_packet(fd, packetf, length)<0)
				{
					perror("Error sending message");
					exit(EXIT_FAILURE);
				}

				close(fd);
				free((void *)packetf);
				getchar();
```

```
                        }
                        kill(getpid(),SIGKILL);// cerrar el proceso de envío
                        /* Esto se hace así porque se comprueba que, si el proceso de envío no se
                        cierra, continúa enviando mensajes y el sistema colapsa*/

                case -1:
                        perror("Error fork send");
                        exit(EXIT_FAILURE);
                default:
                        wait(&status);
                        if(modo == 0) getchar();
                        signal (SIGUSR1, controlador);

                }
        }


        void controlador (int h)
{
        kill(getppid(),SIGUSR1);
}
```

Código de *test.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "sfsource.h"
#include "sfsource.c"

void main(void)
{
        int fd, i=1;
        int port=9002;
        int contador=48;
        int status;
        unsigned char *packetf;
        char moteid[2], count[2],longitud[2];
        int lenght;
        lenght=13;
        longitud[0]=(char)lenght-8+48;
        longitud[1]='\0';

        while(1)
        {
                sleep(1);
                if(contador>=55) contador=47;
                contador++;
                count[0]=(char)contador;
```

```
                count[1]='\0';
                switch(fork())
                {
                case 0:
                        for(i=0;i<2;i++)
                        {
                                packetf=malloc(lenght);
                                if(fd = open_sf_source("localhost", port)<0)
                                {
                                        perror("Couldn't open serial forwarder at localhost:port");
                                        exit(EXIT_FAILURE);
                                }
                                packetf[0] = strtol("0", NULL, 0);
                                packetf[1] = strtol("0", NULL, 0);
                                packetf[2] = strtol("NULL", NULL, 0);//nodeid
                                packetf[3] = strtol("0", NULL, 0);
                                packetf[4] = strtol("0", NULL, 0);
                                packetf[5] = strtol(longitud, NULL, 0);//número de bytes de la
payload
                                packetf[6] = strtol("0", NULL, 0);
                                packetf[7] = strtol("6", NULL, 0);
                                packetf[8] = strtol("0", NULL, 0);//id "enviante"
                                packetf[9] = strtol(count, NULL, 0);//count para los leds
                                packetf[10] = strtol("0", NULL, 0);//rssi
                                packetf[11] = strtol("0", NULL, 0);//lqi
                                packetf[12] = strtol("0", NULL, 0);//powerset
                                printf("longitud %d",lenght);
                                write_sf_packet(fd, packetf, lenght);
                                close(fd);
                                free((void *)packetf);

                        }
                        kill(getpid(),SIGKILL);

                case -1:
                        perror("Error fork send");
                        exit(EXIT_FAILURE);
                default:
                        wait(&status);


                }
        }
}
```

Código de *findmemote*:

Fichero de configuración:

```
#include <Timer.h>
```

```
#include "findmemote.h"
configuration findmemoteAppC {
}
implementation {
        components MainC;
        components LedsC;
        components findmemoteC as App;
        components new TimerMilliC() as Timer0;
        //componentes necesarios para el envío de mensaje
        components ActiveMessageC;
        components new AMSenderC(AM_findmemoteMSG);
        components new AMReceiverC(AM_findmemoteMSG);
        //componente para leer los parámetros de los mensajes enviados y recibidos
        components CC2420ActiveMessageC as Paquete;


        App.Boot -> MainC;
        App.Leds -> LedsC;
        App.Timer0 -> Timer0;
        App.Packet -> AMSenderC;
        App.AMPacket -> AMSenderC;
        App.AMSend -> AMSenderC;
        App.AMControl -> ActiveMessageC;
        App.Receive -> AMReceiverC;
        App.CC2420Packet -> Paquete;
}
```

Componente aplicación:

```
#include <Timer.h>
#include "findmemote.h"

module findmemoteC {
        uses interface Boot;
        uses interface Leds;
        uses interface Timer<TMilli> as Timer0;
// interfaces necesarias para la comunicación radio
        uses interface Packet;
        uses interface AMPacket;
        uses interface AMSend;
        uses interface SplitControl as AMControl;
        uses interface Receive;
        uses interface CC2420Packet;

}
implementation {
        uint8_t counter,power;
        bool busy = FALSE;
        message_t pkt;

        event void Boot.booted() {
                call AMControl.start();//Comienza la com AM
```

```
	}

	event void AMControl.startDone(error_t err) {
	//Se llaman automáticamente estas dos y se les pasa un parámetro de tipo error_t. Las llama AMControl.start()
		if (err == SUCCESS) {
			call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
			// no nos interesa empezar el timer antes de que esté la radio inicializada
		}
		else {
			call AMControl.start();
		}
	}

	event void AMControl.stopDone(error_t err) {
 }


event void Timer0.fired() {
//	counter++;
//	call Leds.set(counter);
	// Aquí es donde comprobamos si está ocupada la línea y transmitimos


	/*if (!busy) {
		findmemoteMsg* fmmpkt = (findmemoteMsg*)(call Packet.getPayload(&pkt, sizeof (findmemoteMsg)));
		fmmpkt->nodeid = TOS_NODE_ID; //ID propia
		fmmpkt->counter = counter;
		call CC2420Packet.setPower(&pkt,power);
		fmmpkt->powerset = call CC2420Packet.getPower(&pkt);
		if (call AMSend.send(0, &pkt, sizeof(findmemoteMsg)) == SUCCESS) {
			busy = TRUE;
		}
	}*/
}

event void AMSend.sendDone(message_t* msg, error_t error) {
		if (&pkt == msg) {
			busy = FALSE;
		}
}


event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
//	call Leds.set(sizeof(findmemoteMsg));
//	call Leds.set(len);
	if (len == sizeof(findmemoteMsg))
	{
		findmemoteMsg* fmmpkt = (findmemoteMsg*)payload;
		if(fmmpkt->counter<8) call Leds.set(fmmpkt->counter);
		power=(uint8_t)fmmpkt->powerset;
```

```
                if (!busy) {
                                findmemoteMsg*  fmmpkt  =  (findmemoteMsg*)(call
Packet.getPayload(&pkt, sizeof (findmemoteMsg)));
                                fmmpkt->nodeid = TOS_NODE_ID; //ID propia
                                fmmpkt->counter = ++counter;
                                call CC2420Packet.setPower(&pkt,power);
                                fmmpkt->powerset = call CC2420Packet.getPower(&pkt);
                                if (call AMSend.send(0, &pkt, sizeof(findmemoteMsg)) == SUCCESS) {
                                        busy = TRUE;
                                }
        }
        }
        return msg;
}
}
```

Código de *BaseStationMod*:

Fichero de configuración:

```
configuration BaseStationC {
}
implementation {
  components MainC, BaseStationP, LedsC;
  components ActiveMessageC as Radio, SerialActiveMessageC as Serial;
  components CC2420ActiveMessageC as Paquete;

  MainC.Boot <- BaseStationP;

  BaseStationP.RadioControl -> Radio;
  BaseStationP.SerialControl -> Serial;

  BaseStationP.UartSend -> Serial;
  BaseStationP.UartReceive -> Serial.Receive;
  BaseStationP.UartPacket -> Serial;
  BaseStationP.UartAMPacket -> Serial;

  BaseStationP.RadioSend -> Radio;
  BaseStationP.RadioReceive -> Radio.Receive;
  BaseStationP.RadioSnoop -> Radio.Snoop;
  BaseStationP.RadioPacket -> Radio;
  BaseStationP.RadioAMPacket -> Radio;

  BaseStationP.Leds -> LedsC;
  BaseStationP.CC2420Packet -> Paquete;
}
```

Componente de aplicación:

```
#include "AM.h"
#include "Serial.h"

module BaseStationP @safe() {
  uses {
    interface Boot;
    interface SplitControl as SerialControl;
    interface SplitControl as RadioControl;
/* SplitControl: Interfaz que sirve para iniciar o parar lo que sea, en este caso la radio o el Serial (se linka al componente
ActiveMessageC por ejemplo). Se llama con SplitControl.start, que devuelve una variable de tipo "error_t". Necesita
la llamada de SplitControl.startdone en un futuro cercano.
Devuelve Success, ebusy (si el dispositivo se está apagando), ealready (si el dispositivo se estaba ya iniciando) o fail.*/

    interface AMSend as UartSend[am_id_t id];
/*Control sending, get payload...*/
    interface Receive as UartReceive[am_id_t id];
    interface Packet as UartPacket;
    interface AMPacket as UartAMPacket;

    interface AMSend as RadioSend[am_id_t id];
    interface Receive as RadioReceive[am_id_t id];
    interface Receive as RadioSnoop[am_id_t id];
    interface Packet as RadioPacket;
    interface AMPacket as RadioAMPacket;
    interface CC2420Packet;

    interface Leds;
  }
}

implementation
{
  enum {
    UART_QUEUE_LEN = 12,
    RADIO_QUEUE_LEN = 12,
    SET_POWER = 3,/**********************POWER**********************/
// Define el tamaño de las colas de mensajes para la radio y el serial
  };
 typedef nx_struct findmemoteMsg {
  nx_uint8_t nodeid;
  nx_uint8_t counter;
  nx_uint8_t rssi;
  nx_uint8_t lqi;
  nx_uint8_t powerset;
} findmemoteMsg;

  message_t  uartQueueBufs[UART_QUEUE_LEN];
  message_t  * ONE_NOK uartQueue[UART_QUEUE_LEN];
```

```
  uint8_t    uartIn, uartOut;
  bool       uartBusy, uartFull;

  message_t  radioQueueBufs[RADIO_QUEUE_LEN];
  message_t  * ONE_NOK radioQueue[RADIO_QUEUE_LEN];
  uint8_t    radioIn, radioOut,k=0;
  bool       radioBusy, radioFull;
/*Crea variables */
  uint8_t setpower=5; //Me creo una variable para definir el power de las señales

  task void uartSendTask();
  task void radioSendTask();
// Simplemente declara funciones y variables

  void dropBlink() {
    call Leds.led2Toggle();
  }

  void failBlink() {
    call Leds.led2Toggle();
  }

  event void Boot.booted() {
    uint8_t i;

    for (i = 0; i < UART_QUEUE_LEN; i++)
      uartQueue[i] = &uartQueueBufs[i];
    uartIn = uartOut = 0;
    uartBusy = FALSE;
    uartFull = TRUE;

    for (i = 0; i < RADIO_QUEUE_LEN; i++)
      radioQueue[i] = &radioQueueBufs[i];
    radioIn = radioOut = 0;
    radioBusy = FALSE;
    radioFull = TRUE;

    call RadioControl.start();
    call SerialControl.start();
  }

  event void RadioControl.startDone(error_t error) {
    if (error == SUCCESS) {
      radioFull = FALSE;
    }
  }

  event void SerialControl.startDone(error_t error) {
    if (error == SUCCESS) {
      uartFull = FALSE;
    }
  }
```

```
  event void SerialControl.stopDone(error_t error) {}
  event void RadioControl.stopDone(error_t error) {}

  uint8_t count = 0;

  message_t* ONE receive(message_t* ONE msg, void* payload, uint8_t len);
// Ésto de alguna forma está declarando el evento que va a ocurrir abajo

  event message_t *RadioSnoop.receive[am_id_t id](message_t *msg,void *payload,uint8_t len) {
    return receive(msg, payload, len);
  }

  event message_t *RadioReceive.receive[am_id_t id](message_t *msg,void *payload,uint8_t len) {

    return receive(msg, payload, len);
  }

  message_t* receive(message_t *msg, void *payload, uint8_t len) {
    message_t *ret = msg;
    findmemoteMsg* btrpkt = (findmemoteMsg*)payload;
    btrpkt->rssi = call CC2420Packet.getRssi(msg);
    btrpkt->lqi = call CC2420Packet.getLqi(msg);
/*Aquí podemos modificar los campos de lo que recibimos por radio antes de enviarlo por serie*/

    atomic
    {
      if (!uartFull)
            {
              ret = uartQueue[uartIn];
              uartQueue[uartIn] = msg;
/*Ésto lo que hace es ir vacianco la pila de mensajes pendientes in*/

              uartIn = (uartIn + 1) % UART_QUEUE_LEN;

              if (uartIn == uartOut)
                uartFull = TRUE;

              if (!uartBusy)
                {
                  post uartSendTask();
                  uartBusy = TRUE;
                }
            }
      else
            dropBlink();
    }

    return ret;
  }

  uint8_t tmpLen;

  task void uartSendTask() {
```

```
  uint8_t len;
  am_id_t id;
  am_addr_t addr, src;
  message_t* msg;
  atomic
    if (uartIn == uartOut && !uartFull)
          {
            uartBusy = FALSE;
            return;
          }

  msg = uartQueue[uartOut];
  tmpLen = len = call RadioPacket.payloadLength(msg);
  id = call RadioAMPacket.type(msg);
  addr = call RadioAMPacket.destination(msg);
  src = call RadioAMPacket.source(msg);
  call UartPacket.clear(msg);
  call UartAMPacket.setSource(msg, src);

  if (call UartSend.send[id](addr, uartQueue[uartOut], len) == SUCCESS)
    call Leds.led1Toggle();
  else
    {
          failBlink();
          post uartSendTask();
    }
}

 event void UartSend.sendDone[am_id_t id](message_t* msg, error_t error) {
  if (error != SUCCESS)
    failBlink();
  else
    atomic
          if (msg == uartQueue[uartOut])
            {
              if (++uartOut >= UART_QUEUE_LEN)
                uartOut = 0;
              if (uartFull)
                uartFull = FALSE;
            }
  post uartSendTask();
 }

 event message_t *UartReceive.receive[am_id_t id](message_t *msg,
                                                  void *payload,
                                                  uint8_t len)
          {
/*Aquí podemos configurar los campos del mensaje in serie out radio*/
  message_t *ret = msg;
  findmemoteMsg* btrpkt = (findmemoteMsg*)payload;

  bool reflectToken = FALSE;
```

```
  btrpkt->powerset=(uint8_t)setpower;//Así conseguimos que el power sea el mismo para base y mota
  atomic
   if (!radioFull)
          {
           reflectToken = TRUE;
           ret = radioQueue[radioIn];
           radioQueue[radioIn] = msg;
           if (++radioIn >= RADIO_QUEUE_LEN)
            radioIn = 0;
           if (radioIn == radioOut)
            radioFull = TRUE;

           if (!radioBusy)
            {
              post radioSendTask();
              radioBusy = TRUE;
            }
          }
    else
          dropBlink();

  if (reflectToken) {

  }

  return ret;
 }

 task void radioSendTask() {
  uint8_t len;
  am_id_t id;
  am_addr_t addr,source;
  message_t* msg;

  atomic
   if (radioIn == radioOut && !radioFull)
          {
           radioBusy = FALSE;
           return;
          }

  msg = radioQueue[radioOut];
  len = call UartPacket.payloadLength(msg);
  addr = call UartAMPacket.destination(msg);
          if (addr==NULL) addr=0xffff;
/*Cuando entremos en el modo "test" de envío contínuo, mandaremos addr=NULL para que se envíe el
mensaje a todas las motas*/
  source = call UartAMPacket.source(msg);
  id = call UartAMPacket.type(msg);

  call RadioPacket.clear(msg);
  call RadioAMPacket.setSource(msg, source);
  call CC2420Packet.setPower(msg,setpower);/*Seteamos el power de la señal radio out*/
```

```
  if (call RadioSend.send[id](addr, msg, len) == SUCCESS)
    call Leds.led0Toggle();
  else
    {
          failBlink();
          post radioSendTask();
    }
 }
 event void RadioSend.sendDone[am_id_t id](message_t* msg, error_t error) {
  if (error != SUCCESS)
    failBlink();
  else
    atomic
          if (msg == radioQueue[radioOut])
            {
              if (++radioOut >= RADIO_QUEUE_LEN)
                radioOut = 0;
              if (radioFull)
                radioFull = FALSE;
            }
    post radioSendTask();
 }
}
```

ANEXO IV

Compilación, instalación y ejecución de las aplicaciones

Todos los módulos de la aplicación de interfaz de usuario se compilan con un *makefile*, el cual genera los ficheros objeto primeramente, y luego los compila a ejecutables.

También se pueden compilar de forma manual ejecutando:

>> gcc -lm -c aplicacion.c aplicacion.h
>> gcc -lm -c send.c
>> gcc -lm -c test.c

El parámetro *–lm* sirve para que las funciones de la librería *math.h* funcionen correctamente.

La aplicación se ejecuta con:
./aplicación opc1 opc2

Donde opc1 y opc2 son los modos de la aplicación, siendo:

- opc1: "s" para el modo de envío, "t" para el modo de test.
- opc2: "l" para el modo *listen* o "de escucha activa", "o" para el modo de aviso de pérdida.

Las aplicaciones *BaseStationMod* y *findmemote* se compilan de la misma manera, ejecutando:
make micaz

Y se instalan con el comando:
*make micaz reinstall.*N *mib510,/dev/*puerto

O bien se puede ejecutar el comando siguiente, que compila e instala en el nodo en un mismo paso:
*make micaz install.*N *mib510,/dev/*puerto

Donde *N* es el número que queremos asignar al nodo (si se omite, asigna 0), y *puerto* es la designación del puerto de conexión serie para programar.

Cuando un nodo se conecta a un equipo, lo hace por dos puertos de serie virtuales, normalmente consecutivos, el de número más bajo sirve para programar el nodo, mientras que el más alto se utiliza para la comunicación serie.

La nomenclatura para estos puertos es, habitualmente:

- ttyS*n* para sistemas Windows (utilizando el simulador de entorno Linux Cygwin) y algunas distribuciones de Linux, donde *n* es el número del puerto.
- ttyUSB*n* para la mayoría de las distribuciones de Linux, donde *n* es el número del puerto.

Referencias

[1] M. Soledad Escolar García, Wireless Sensor Networks: Estado del Arte e investigación, http://arcos.inf.uc3m.es/~sescolar/index_files/presentacion/wsn.pdf.

[2] Miguel Castán Artal, Ana García Armada, Estudio de las redes inalámbricas de sensores aplicadas a la localización de objetos, Octubre 2013

[3] David Gay, Philip Levis, David Culler, Eric Brewer, nesC 1.3 Language Reference Manual:
https://github.com/tinyos/nesc/blob/master/doc/ref.pdf?raw=true, Julio 2009.

[4] TinyOS Wiki, Stanford University:
http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Overview, Octubre 2013.

[5] Wikipedia, Artículo sobre Zigbee:
http://es.wikipedia.org/wiki/ZigBee, Octubre 2013.

[6] Portal oficial de Zigbee:
http://www.zigbee.org/About/UnderstandingZigBee.aspx, Marzo 2014.

[7] Comparativa Zigbee y Bluetooth:
http://es.wikipedia.org/wiki/ZigBee#ZigBee_vs_Bluetooth, Octubre 2013.

[8] Estándares Zigbee:
http://www.zigbee.org/About/AboutTechnology/ZigBeeTechnology.aspx, Octubre 2013.

[9] Zigbee en el mercado:
http://www.zigbee.org/About/AboutTechnology/MarketLeadership.aspx, Octubre 2013.

[10] Productos Zigbee:
http://www.zigbee.org/Products/ByFunction/Closures.aspx, Noviembre 2013.

[11] Especificación MIB510:
http://www.memsic.com/wireless-sensor-networks/MIB520, Noviembre 2013

[12] Acerca de TinyOS,
http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Overview, Noviembre 2013.

http://www.tinyos.net/, Noviembre 2013.
http://en.wikipedia.org/wiki/TinyOS#History, Noviembre 2013.

[13] Acerca de Cygwin:
www.cygwin.com, Noviembre 2013.

[14] Acerca de nesC:
http://en.wikipedia.org/wiki/NesC, Noviembre 2013.

[15] Características del transmisor CC1000:
http://prochild.com/board/files/tb_6/Home_Automation_systems_1_0.pdf, Febrero 2014.

[16] Características de la mota mica2dot:
http://www.datasheetarchive.com/dl/Datasheet-026/DSA00462856.pdf, Febrero 2014.

[17] Especificación MSP430:
http://www.ti.com/product/msp430f123, Febrero 2014.

[18] Características de la familia TelosB:
http://www.willow.co.uk/TelosB_Datasheet.pdf, Febrero 2014.

[19] Características de la familia MicaZ:
http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf, Febrero 2014.

[18] Safe TinyOS:
http://tinyos.stanford.edu/tinyos-wiki/index.php/Safe_TinyOS, Marzo 2014.

[19] Estructura de un mensaje en TinyOS:
http://tinyos.stanford.edu/tinyos-wiki/index.php/Mote-PC_serial_communication_and_SerialForwarder, Marzo 2014.

[20] Plantilla de presupuesto para un PFC:
http://portal.uc3m.es/portal/page/portal/administracion_campus_leganes_est_cg/proyecto_fin_carrera, Abril 2014.

[21] Tutorial de instalación de TinyOS:
http://tinyos.stanford.edu/tinyos-wiki/index.php/Installing_TinyOS, Abril 2014.

[22] Características de la familia mica2:
http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf, Febrero 2014