ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

MASTER UNIVERSITARIO EN TECNOLOGÍAS INFORMÁTICAS

Título del proyecto:

AGRICULTURA DE PRECISIÓN MEDIANTE REDES INALÁMBRICAS DE SENSORES

Alumno: Castillejo Erviti, Mikel

Tutor: Astrain Escola, Jose Javier

Co-Tutor: Villadangos Alonso, Jesús

Pamplona, 27 de Septiembre de 2013

Contenido

Objeto y desarrollo

Con este Trabajo Final de Máster se pretende diseñar e implantar una infraestructura de red inalámbrica de sensores que permitan obtener datos en un entorno agrario para ayudar en la toma de decisiones.

Específicamente, se pretende usar una serie de placas de sensores dotados de comunicación inalámbrica y sobre ellas implementar una serie de algoritmos de recogida de datos que nos permitan disponer en un punto centralizado de la información suficiente para realizar una agricultura de precisión.

La agricultura de precisión permite optimizar las técnicas de gestión de riegos, mejorando la calidad de la producción, aumentando el rendimiento de los cultivos y optimizando el uso de los recursos naturales. Esto permite maximizar la calidad, minimizar el impacto medio ambiental y minimizar los riesgos de la cosecha. El impacto de esta práctica se observa en distintos ámbitos:

- **Agronómico**: adaptación a las necesidades y requerimientos de los cultivos en función de los suelos, las condiciones atmosféricas, las estaciones del año, el empleo de invernaderos…
- **Medioambiental**: minimización del impacto sobre el medio ambiente de las actividades agrícolas.
- **Económico**: mejora de la competitividad por una optimización de las prácticas agrícolas obteniendo mejoras en la calidad y/o cantidad de la producción, optimizando el empleo de maquinaria y de productos fitosanitarios, abonos...
- **Gestión empresarial**: optimización en el proceso de recogida de la producción minimizando los tiempos de recolección, de transporte de la producción, de logística... Permite una capacidad de previsión que posibilita una mejor adaptación a las condiciones de mercado existentes.
- **Trazabilidad de los productos**. Denominaciones de origen.

Por medio del uso de la agricultura de precisión, se persigue alcanzar los siguientes objetivos:

- Reducción de la inversión inicial y de los costes de instalación y de operación.
- Monitorización, con la granularidad elegida, de las parcelas.

- Movilidad de los sensores: autoconfiguración y autoadaptabilidad de los sensores.
- Escalabilidad en número de sensores, superficie monitorizada, y dimensionamiento de la red de sensores con cambios mínimos en la red.
- Reducción de costes de mantenimiento, ya que los dispositivos tienen autonomía de hasta varios años. Al no haber cables, el mantenimiento se reduce considerablemente.
- Ubicuidad. La aplicación de monitorización permitirá el acceso ubicuo a la información.
- Formación del alumno en la aplicación de la investigación a un área productiva.

En este TFM se propone analizar las topologías y distribuciones de las redes de sensores, así como los mecanismos de comunicación entre sensores y la red de datos. Este estudio incluirá la auto-configuración de las redes, los mecanismos de agregación de datos y enrutamiento y la eficiencia en el consumo energético.
La Figura 1 ilustra el marco de trabajo en el que se incluye el trabajo.

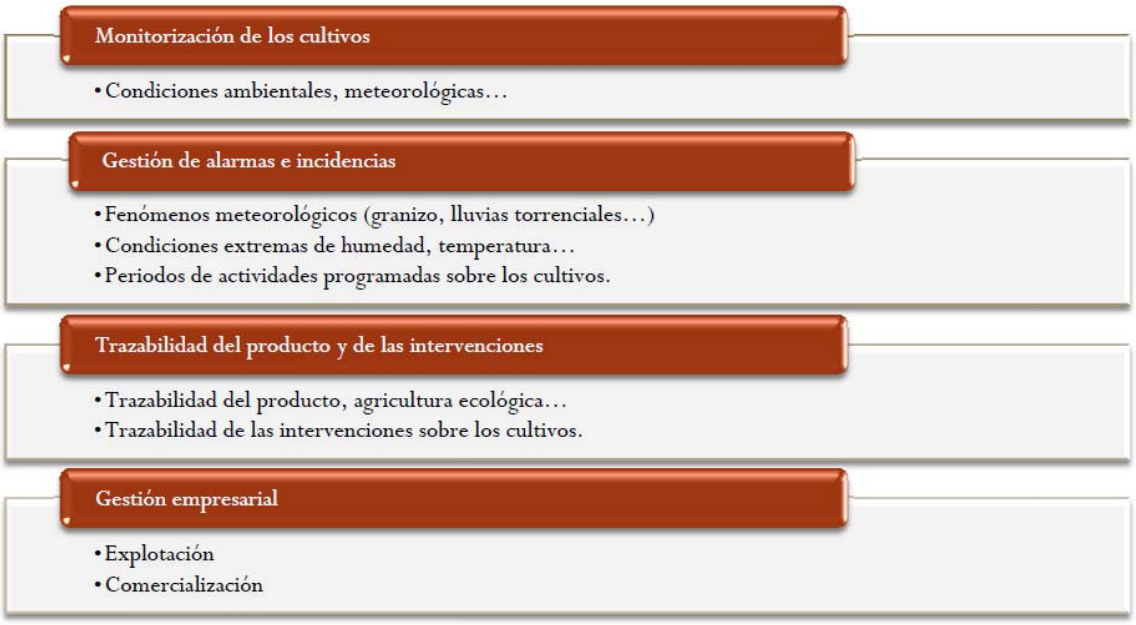


Figura 1. Marco de trabajo de la propuesta.

El alcance que se pretende abarcar con el presente TFM es el siguiente:

- Estudio, análisis y desarrollo de algoritmos de monitorización de parcelas de cultivo mediante redes de sensores.
- Análisis de topologías de despliegue de nodos y de estrategias de agregación que optimicen el consumo energético.
- Estudio de técnicas de autoconfiguración y auto organización de redes de sensores.
- Diseño de un sistema escalable, sin necesidad de conexión de alimentación externa, de fácil instalación y accesible desde cualquier lugar con conexión a internet.

1.- Introducción

1.1.- Formulación del problema a resolver

El sector agrícola cada vez se está volviendo más importante para el ser humano. Cada vez la gente está más interesada en consumir productos naturales que no hayan sido tratados y modificados genéticamente. Con el fin de conseguir cultivos naturales eficientes y productos de calidad es muy importante mantener un control constante sobre los diferentes elementos que pueden afectar a los cultivos como puede ser la humedad, la radiación solar, los diferentes gases que rodean a los cultivos, etc. Por ello, es necesario conocer en cualquier momento las condiciones a las que están sometidas las explotaciones agrícolas. Con este fin, cada vez más se está trabajando en proyectos que busquen incorporar la monitorización por medio de redes de sensores a las explotaciones agrícolas. [1]

El objetivo es desplegar una red de sensores en una explotación agrícola que nos permita obtener información constante y actualizada sobre las características que afectan al cultivo con el fin de poder actuar lo más rápido posible ante las adversidades (falta de agua, gran concentración de gases perjudiciales, etc.). Al tener un flujo constante de información que nos permite responder lo antes posible ante amenazas externas y que nos permiten tener siempre controlados y en valores óptimos los aspectos que afectan a nuestra explotación agrícola, podemos decir que estamos realizando una agricultura de precisión. Según [2], La agricultura de precisión es un concepto agronómico de gestión de parcelas agrícolas, basado en la existencia de variabilidad en campo. Requiere el uso de las tecnologías de Sistemas de Posicionamiento Global (GPS), sensores, satélites e imágenes aéreas junto con Sistemas de Información Geográfica (SIG) para estimar, evaluar y entender dichas variaciones. La información recolectada puede ser usada para evaluar con mayor precisión la densidad óptima de siembra, estimar fertilizantes y otras entradas necesarias, y predecir con más exactitud la producción de los cultivos.

Sobre esta red de sensores habrá que diseñar e implementar un algoritmo de recogida de datos lo más eficiente posible en todos los sentidos, sobre todo en el aspecto energético. El aspecto energético es muy importante en las redes de sensores ya que habitualmente, como en este caso, lo sensores no están conectados a una fuente continua de energía si no que disponen de una batería. Al ser esta de una capacidad limitada, cuanto menor sea el consumo energético

de los sensores mayor será el tiempo que pueden estar la red de sensores funcionando antes de que los sensores se empiecen a quedar sin batería y haya que remplazarlas o recargarlas. En el escenario que vamos a plantear, una explotación agrícola, el cambio o recarga de baterías es más o menos sencillo, al ser de fácil acceso, pero si extrapolamos las redes de sensores a otros ámbitos, por ejemplo a la detección de incendios en bosques, puede que el acceso a los sensores no sea tan sencillo por lo que cobra aún más importancia el aspecto energético y el tiempo que los sensores pueden estar trabajando hasta que sea necesario actuar sobre la batería de los mismos.

La función de implantación de la red de sensores y de diseño del algoritmo de recogida de datos es sólo la primera parte de un sistema completo de toma de decisión que puede convertir una explotación agrícola normal en una explotación totalmente inteligente. Como se puede ver en la Figura 2, el objetivo completo sería el de implantar una red de sensores y que la información obtenida por la misma se integre dentro del sistema de información de la explotación. Por último, emplearíamos una serie de mecanismos de ayuda a la toma de decisión que nos permitiría obtener la mejor decisión a tomar en cada momento con la información constante que vamos obteniendo. En este trabajo sólo no vamos a encargar de la primera parte de este esquema aunque conviene tener una visión global del mismo para ver lo que se puede llegar a alcanzar.

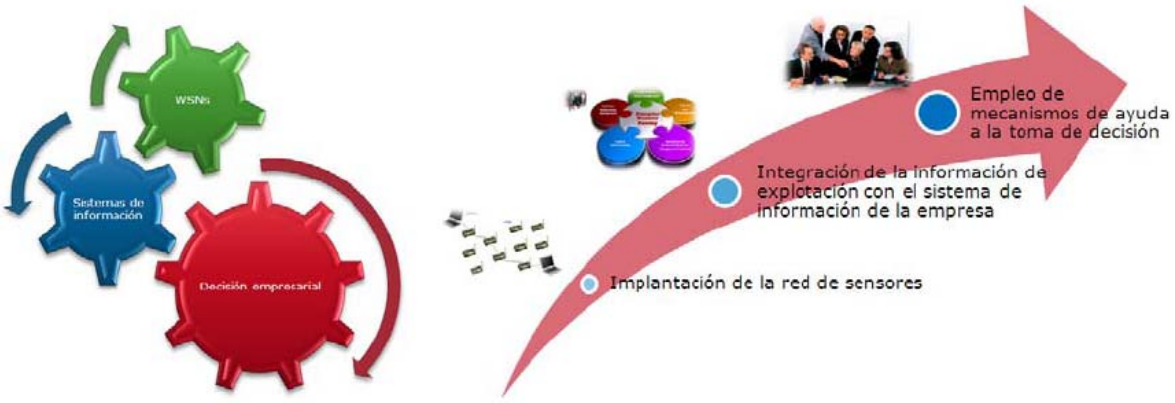


Figura 2. Esquema en el que se sitúa el presente trabajo.

1.2.- Antecedentes y estado del arte

El estudio e implementación de redes de sensores para la recogida de información en general, y relacionados con el mundo de la agricultura en particular, cada vez está cobrando más información. Prueba de ello es la cantidad de proyectos y artículos que se pueden encontrar relacionados con este tema [3-22].

Dentro de los estudios relacionados con la agricultura de precisión podemos encontrar dos tipos de estudios. La mayoría de estudios y trabajos que nos encontramos se centran en la creación de sistemas de información o implementación de redes de sensores que permitan obtener y explotar la información obtenida por los sensores sin tener en cuenta los algoritmos utilizados para recoger la información [3-10, 12, 14, 16, 18, 20-22]. Por otro lado, otros estudios se centran en la creación de nuevos algoritmos de recogida de datos [11, 13, 15, 17, 19]. A continuación se presentan algunos artículos relacionados con la agricultura de precisión que resultan de especial interés en este marco de trabajo.

En [4] se desarrolla una plataforma web para dar soporte a los diferentes estudios del Proyecto del Consorcio de la Toscana en materia de viticultura de precisión. El objetivo es crear una arquitectura Web donde los diferentes actores puedan subir información y comparar y consultar datos. Este proyecto consistiría en la creación de un sistema de información que ayude en la toma de decisión. También en [14] se crea una plataforma que utiliza sistemas GIS (sistemas de información geográfica) para crear una plataforma que permita acceder a datos de explotaciones agrícolas desde cualquier lugar.

En [20] los autores realizan un estudio básico de las redes de sensores y su uso en la agricultura, en especial en invernaderos. Los autores realizan un estudio de los diferentes elementos que forman una red de sensores, tanto a nivel físico con el estudio de los sensores y actuadores, como a nivel lógico con el estudio de los diferentes módulos que forman la red de sensores. También realizan un pequeño estudio de los diferentes aspectos que afectan a una red de sensores como las condiciones climáticas, la eficiencia energética, el coste de los sensores, etc.

En [3] se centran en crear un *framework* que permita crear una red de sensores por medio de *plug-and-play*.

La mayoría de los artículos [5-10, 12, 16, 18, 21, 22] se centrar en el diseño, implementación y prueba de redes de sensores para casos concretos como puede ser para invernaderos [7] [9] o para sistemas de riego [10].

En general en estos artículos me parece que la explicación de las pruebas realizadas y los métodos para obtener los resultados son bastante pobres.

Entre los artículos que hablan de nuevos algoritmos para la recogida de datos, podemos encontrar artículos que desarrollan algoritmos en cadena o en cluster. Entre los primeros, uno de los más conocidos es el algoritmo PEGASIS [15] y [19]. Entre los segundos, podemos encontrar el algoritmo LEACH [11] y [19]. Además de estos, podemos encontrar los algoritmos [13] y [17] en los que se basa en el estudio, desarrollo, implantación y validación de este trabajo en el marco de la agricultura de precisión. En el Capítulo 2 explicaremos el funcionamiento, tanto de LEACH como de PEGASIS brevemente, como de los algoritmos que vamos a implementar en este trabajo. En [19] se explican varias arquitecturas y algoritmos que podrían servir para desarrollar o probar nuevos algoritmos de recogida de datos.

Este trabajo va a contribuir al tema en la implementación de una red de sensores en explotación de vides usando un tipo concreto de sensores que luego veremos que son los sensores Waspmote. Además, también vamos a desarrollar en el lenguaje de estos sensores los algoritmos de arquitectura en cadena y en cluster presentados en [13] y [17], y los vamos a probar en una red real, en vez de en un simulador como algunos otros artículos, y vamos a comparar los resultados obtenidos con ellos.

1.3.- Objetivos y solución propuesta

En este trabajo se van a centrar los esfuerzos en el estudio, análisis, desarrollo, implantación y evaluación de distintos algoritmos para la recolección de información mediante redes inalámbricas de sensores en las vides de la finca de prácticas de la Universidad Pública de Navarra.

El objetivo del proyecto va a ser el de, sobre una explotación agrícola, implantar una red de sensores y, sobre ellos, implementar una serie de algoritmos de recogida de datos con objeto de permitir la aplicación de la agricultura de precisión. Una vez implementados los algoritmos, realizaremos las mediciones oportunas y compararemos los resultados obtenidos por unos algoritmos y otros.

Dentro de los algoritmos de recogida de datos nos encontramos con dos tipos diferenciados. Aquellos en los que los sensores trabajan en cadena pasando la información de unos a otros hasta llegar a la base de recogida de datos y aquellos en los que los sensores forman un cluster y todos los sensores envían directamente su información a un punto común o base sin pasar por el resto. Las peculiaridades de ambas estrategias serán abordadas con mayor detalle en el Capítulo 2 de esta memoria. Concretamente, en este trabajo vamos a probar un algoritmo basado en cadena y tres algoritmos basados en cluster. En el Capítulo 2 plantearemos más específicamente el problema que queremos resolver y los algoritmos que vamos a tratar como solución.

1.4.- Introducción a los elementos que se van a utilizar

El primer paso antes de empezar a diseñar una posible solución a nuestro problema es decidir los elementos con los que vamos a trabajar. Hay dos elementos importantes sobre los que hay que tomar una decisión y que van a afectar a las posibles soluciones que podemos obtener.

El primero de ellos es la estrategia de comunicación que vamos a utilizar ya que va a influir en las topologías de red que podemos emplear, el protocolo de comunicaciones a emplear, la distancia a la que podemos colocar los sensores, etc.

El segundo de los elementos sobre el que tenemos que tomar una decisión es las placas de sensores que vamos a utilizar ya que nos va a marcar el lenguaje de programación que vamos emplear así como las diferentes restricciones de memoria, etc. que tengan las propias placas.

1.4.1.- Estándar de comunicaciones

La primera decisión que tenemos que tomar a la hora de diseñar una red inalámbrica es el estándar de comunicaciones que vamos a utilizar. Debido a que los sensores no disponen de una fuente de alimentación interrumpida, sino que se alimentan de una batería de duración finita, necesitamos elegir un estándar de comunicaciones inalámbrico que nos asegure un alcance considerable a la vez que un bajo consumo eléctrico. Se va a hacer uso de un estándar de comunicaciones inalámbrico ya que todos los sensores encontrados hacen uso de estos estándares en vez de estándares cableados. Esto permite ahorrar costes al no ser necesario desplegar cables y además permite una mayor movilidad de los sensores, como por ejemplo embarcarlos en medios móviles, cosa que no es posible con un estándar de comunicaciones inalámbrico. Vamos a ver las ventajas e inconvenientes del uso de tres estándares de comunicación inalámbricos que son los siguientes:

- WiFi/802.11
- Bluetooth/802.15.1
- ZigBee/802.15.4

1.4.1.1.- WiFi/802.11

WiFi es un estándar de comunicación inalámbrico inicialmente utilizado en redes de área local y luego convertido en un medio para acceder a Internet de banda ancha. La norma IEEE 802.11 es un estándar internacional que describe las características de una red de área local inalámbrica. WiFi es el nombre dado inicialmente a esta certificación por la WECA (Wireless Ethernet Compatibility Alliance), el organismo encargado de certificar que los equipos cumplan la norma 802.11. Debido a un abuso de lenguaje y por razones de marketing el nombre de la norma se

confunde hoy en día con el nombre de la certificación. De este modo una red WIFI es en realidad una red conforme a la norma 802.11. [23]

La principal característica que nos va a ofrecer este estándar frente a los demás es una gran velocidad de transmisión de datos a costa de un gran consumo eléctrico. Permite el crear redes de cualquier topología (ad-hoc, estrella, árbol, etc.) y tamaño. También permite cifrar los datos por medio de sistemas de cifrado como WEP, WPA o WPA2.

1.4.1.2.- Bluetooth/802.15.1

Bluetooth es un estándar para redes inalámbricas que permite la transmisión de voz y datos entre distintos dispositivos mediante un enlace de radiofrecuencia en la banda ISM de los 2,4GHz. El término Bluetooth es la denominación comercial y popular del estándar de comunicación inalámbrica IEEE 802.15.1**.**

El estándar Bluetooth nos ofrece una velocidad de transmisión y un consumo eléctrico inferior a WiFi pero superior a ZigBee. Permite realizar una encriptación de datos de 64 o de 128. A la hora de crear redes de dispositivos estamos limitados a redes Ad-Hoc de 8 dispositivos momo máximos, las llamadas piconets.

1.4.1.3.- ZigBee/802.15.4

ZigBee es el nombre de la especificación de un conjunto de protocolos de alto nivel de comunicación inalámbrica para su utilización con radiodifusión digital de bajo consumo, basada en el estándar IEEE 802.15.4. La encargada de especificar y mantener ese estándar es la ZigBee Alliance, formado por un gran número de empresas. Opera en las frecuencias 868 MHz en Europa, 915 en Estados Unidos y 2,4 GHz en todo el mundo.

El protocolo ZigBee nos ofrece la menor tasa de transferencia de datos de los tres protocolos pero en cambio también nos ofrece el menor consumo eléctrico de los tres. Nos permite crear redes Ad-Hoc, en estrella, etc. y estamos limitados a 65535 dispositivos divididos en sub-redes de 255 dispositivos. Estas sub-redes se comunicarán entre ellas por medio del coordinador de la red. Permite realizar una encriptación de datos de 128 bits.

1.4.1.4.- Comparación de las tecnologías

Primeramente vamos a ver los datos de una serie de dispositivos reales para ilustrar a qué nos referimos cuando afirmamos que una tecnología consume más energía que otra o si tiene mayor o menor alcance.

Dispositivo MRF24J40MA (ZigBee) [24]

- Tasa de Transmisión: 250 Kbps
- Rango:
  - Interior: 38m
  - Exterior o con línea de vista: 120m
- Consumo:
  - Transmisión: 23mA
  - Recepción: 19mA
  - Dormido: 2uA

XBEE (ZigBee) [25]

- Tasa de Transmisión: 250 Kbps
- Rango:
  - XBEE:
    - Interior: 40m
    - Exterior o con línea de vista: 120m
  - XBEE PRO:
    - Interior: 90m
    - Exterior o línea de vista: 3200m
- Consumo:
  - XBEE:
    - Transmisión: 35 mA – 45 mA
    - Recepción: 38mA – 40 mA
    - Dormido: 1uA
  - XBEE PRO:
    - Transmisión: 205 mA – 220 mA
    - Recepción: 47mA – 62 mA
    - Dormido: 4uA

RN-42 (Bluetooth clase 2) [26]

- Tasa de Transmisión: Máximo 3Mbps
- Rango: 10 – 20m
- Consumo:
  - Transmisión: 30mA
  - Dormido: 26uA

RN-131C (WiFi) [26]

- Tasa de transmisión:
  - 1 – 11Mbps usando 802.11b
  - 6 – 54Mbps usando 802.11g
- Rango: 100 m
- Consumo:
  - Transmisión: 210 mA
  - Recepción: 40 mA
  - Dormido: 4uA

Por medio de los datos obtenidos a partir de diferentes dispositivos que implementan los protocolos indicados, vistos anteriormente, y otras fuentes consultadas [27] [28] [29] podemos elaborar la Tabla 1 que hace una comparativa básica entre protocolos.

	ZigBee	**Wifi**	**Bluetooth**
Rango	>1Km	>1Km	>100m
Velocidad	<256Kbps	<300Mbps	<24Mbps
Consumo	Bajo (Posibilidad de reposo)	Muy alto	Bajo-Medio
Encriptación	128 bits	WPE, WPA, WPA2	64 y 128 bits

Topologías	Ad-Hoc, P2P, Árbol, Estrella, Malla Hasta 65535 divididos en sub-redes de 255	Ad-Hoc, Árbol, Estrella, etc.	Ad-Hoc Hasta 8 dispositivos

Tabla 1. Comparación entre protocolos

A partir de los datos que tenemos, vamos a decidir el protocolo de comunicaciones a emplear.

El protocolo WiFi lo descartamos ya que el consumo eléctrico empleado es muy superior al de los otros protocolos y, como hemos dicho, el consumo es una de las características más importantes al no disponer de una fuente de alimentación ininterrumpida.

Tanto ZigBee como Bluetooth presentan unas características muy parecidas aunque, en general, ZigBee presenta mejores características al tener un consumo menor y permitir la creación de redes con mayor número de dispositivos ya que este es uno de los puntos débiles de Bluetooth al permitir únicamente la creación de una red con 8 dispositivos. La única característica donde Bluetooth es superior a ZigBee es en que presenta una tasa de transferencia mayor pero al no ser este un factor determinante de nuestra red, ya que vamos a mandar relativamente pocos datos y el tiempo de transferencia no es crucial al podernos permitir un retraso de unos pocos segundos en la toma de decisiones, optamos por usar el protocolo ZigBee/802.15.4 en nuestra red inalámbrica de sensores.

1.4.1.5.- ZigBee o 802.15.4

Anteriormente hemos dicho que ZigBee es un protocolo que implementa el estándar IEEE 802.15.4 y hasta ahora hemos hablado de ellos como uno solo pero a la hora de diseñar e implementar nuestra red inalámbrica tenemos que decidir hacer uso de ZigBee o de 802.15.4. Para ellos, vamos a ver las diferencias entre uno y otro y vamos a tomar una decisión.

El protocolo de comunicaciones IEEE 802.15.4 se encarga principalmente de definir una comunicación en la capa 2 del modelo OSI, la capa de enlace de datos. Esto permite que dos dispositivos se comuniquen entre sí de manera sencilla.

El protocolo ZigBee implementa las capas superiores del modelo OSI lo que permite ofrecer diferentes características añadidas.

En la Figura 3 podemos observar las capas de las que se encarga cada protocolo, 802.15.4 y ZigBee, y cómo los dispositivos Waspmote que vamos a utilizar, y de los que hablaremos más adelante, así como los dispositivos SquidBee pueden trabajar con ambos protocolos por medio de los módulos de comunicación correspondientes. [30]

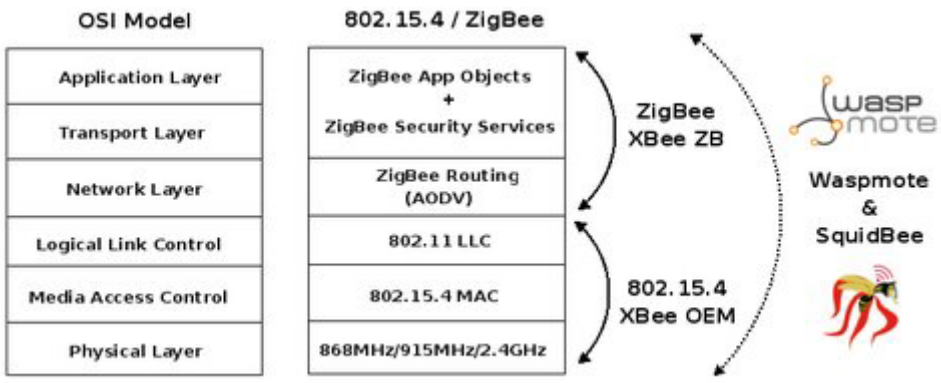


Figura 3. Relación entre los protocolos y las capas OSI. [30]

La implementación de capas superiores del modelo OSI permite que ZigBee ofrezca una serie de características extra como son las siguientes:

- Servicios de cifrado.
- Funciones de asociación y autenticación.
- Protocolos de enrutamiento.

Como contrapartida, ZigBee no permite que los dispositivos "hablen" entre ellos si no que siempre tiene que haber al menos un coordinador de la red al que todos tienen que dirigirse si quieren unirse a la red. Además, al ser necesario la inclusión en los paquetes de una cabecera ZigBee se reduce la cantidad de datos que podemos incluir en cada paquete. Por último, al menos el coordinador de la red tiene que estar conectado a una fuente de alimentación ininterrumpida ya que su consumo energético es mucho mayor y, en caso de agotarse la batería, no se permite el cambio de coordinador de la red si no que habría que crear una nueva.

Considero que la complejidad que añade el protocolo ZigBee, al obligar a crear redes para poder comunicar dispositivos entre sí, unido a la reducción en la cantidad de datos que se puede enviar en cada paquete, no compensa las mejoras que ofrece ya que el cifrado no es necesario al no ser crítica la confidencialidad de los datos que vamos a manejar, el enrutamiento no es necesario ya que los algoritmos que vamos a utilizar transmiten punto a punto y las funciones de asociación y autenticación son un inconveniente más que una ventaja ya que, como acabo de decir, los algoritmos van a comunicarse punto a punto, algo que no es posible con ZigBee al obligar a la creación de una red. Es por esto por lo que he decidido utilizar en el trabajo el protocolo 802.15.4 frente al protocolo ZigBee.

1.4.2.- Placa de sensores

Para la realización de nuestro Trabajo Final de Master hemos decidido utilizar placas de sensores de la compañía Libelium [31], una empresa creada por dos ingenieros de Zaragoza, Alicia Asín y David Gascón, que ha recibido numerosos premios y el apoyo de empresas importantes como Telefónica, IBM, Sony Ericsson o Vodafone.

Esta empresa está especializada en el diseño y construcción de placas de sensores siendo los que nos importan las placas Waspmote, lanzadas al mercado en 2009**.**

En su página web [31] podemos encontrar diferentes proyectos que se han llevado o que se están llevando a cabo usando productos Libelium, en especial Waspmote y Meshlium. Algunos de estos proyectos, que nos pueden dar una idea del uso que se le puede dar a las placas de sensores, son los siguientes [32]:

- Detección de niveles de radiación en Fukushima.
- Aparcamiento inteligente y monitorización del entorno en ciudades en Santander
- Monitorización de viñas para agricultura inteligente en Galicia
- Detección de incendios en bosques de Asturias

1.4.2.1.- Waspmote

Waspmote es el nombre que recibe la placa de sensores que vamos a utilizar. La placa básica consta de los elementos mínimos necesarios para poder ejecutar programas (microprocesador, memoria, etc.) al mismo tiempo que nos da la posibilidad de añadirle módulos y elementos que amplíen sus posibilidades como placas de sensores, tarjetas micro SD, baterías auxiliares, etc. Como tal, la placa Waspmote sólo dispone de un sensor de temperatura, no extremadamente

preciso, y de un acelerómetro. Más adelante vamos a ver cómo hay placas de sensores propiamente dichas que están formadas únicamente por sensores, sin procesador o memoria, y que se pueden añadir a la placa Waspmote básica. Por comodidad, cada vez que hablemos de placa de sensores o de sensor estaremos hablando de la placa básica y de los módulos añadidos que le podemos poner como estas placas de sensores o el módulo de comunicaciones.

En la Tabla 2 podemos observar las características de las placas Waspmote.

Microprocesador	ATmega1281
Velocidad	8MHz
SRAM	8KB
EEPROM	4KB
FLASH	128KB
Tarjeta micro SD	Hasta 2GB
Peso	20gr
Dimensiones	73.5 x 51 x 13 mm
Rango de temperatura	-20ºC - +65ºC
Reloj	RTC (32KHz)

Tabla 2. Características de la placa Waspmote.

En la Figura 4 podemos observar una placa Waspmote, versión 1.1 que es la que vamos a utilizar, la cual tiene conectada un módulo de comunicaciones GPRS y un módulo de comunicaciones ZigBee que es el que tiene antena.

Figura 4. Placa Waspmote.

En la Figura 5 y en la Figura 6 podemos observar la parte superior e inferior de la placa, respectivamente, con sus partes más importantes señaladas.

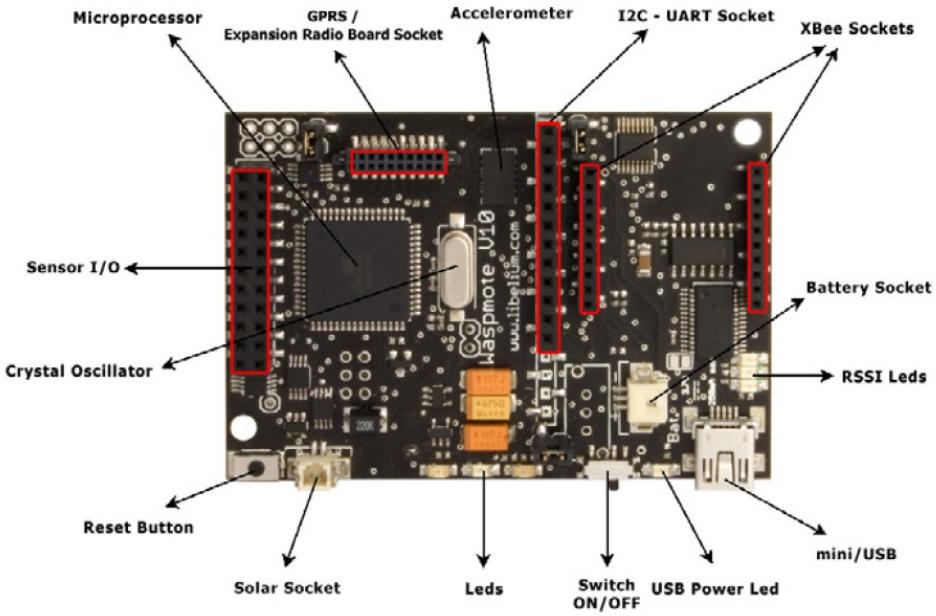


Figura 5. Parte superior de Waspmote

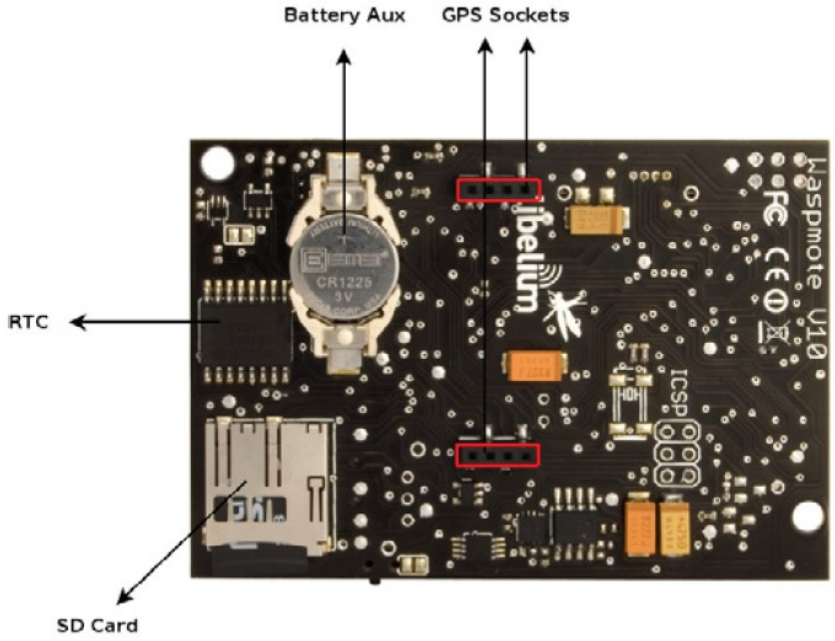


Figura 6. Parte inferior de Waspmote

Algunas de los componentes que podemos destacar de la placa Waspmote son los siguientes:

- Interruptor de encendido-apagado.
- Puerto mini USB para poder conectarlo a un PC.
- Socket Xbee para poder conectar el módulo de comunicaciones.
- Batería auxiliar para poder usar las funciones de apagado y alarma del Waspmote.
- Socket para tarjetas micro SD.
- Sensor de temperatura y acelerómetro incorporado.

Además de la placa Waspmote, Libelium cuenta con una serie de placas de sensores compatibles con la placa Waspmote. Por ejemplo, dispone de placas de detección de gases,

detección de radiación, cámaras, monitorización de cultivos, etc. Además, Libelium permite crear placas de sensores a medida. En la Figura 7 podemos observar de izquierda a derecha una placa de gases, de monitorización de cultivos y una cámara.

Figura 7. Placa de gases, monitorización de cultivos y cámara

Otro de los productos que ofrece Libelium relacionados con las placas Waspmote son los módulos de comunicaciones. Dentro de las opciones que podemos encontrar tenemos 802.15.4/ZigBee, WiFi, 3G+GPS, GSM/GPRS, Bluetooth, RFID/NFC, así como una placa de expansión que nos permite conectar dos módulos de comunicaciones sobre un Waspmote al mismo tiempo. En la Figura 8 podemos observar el módulo 802.15.4, el WiFi y el GSM/GPRS.

Figura 8. Módulo 802.15.4, WiFi y GSM/GPRS

Dentro de los módulos 802.15.4/ZigBee, que son los que vamos a utilizar, podemos encontrar módulos que trabajen con el protocolo 802.15.4 o con el protocolo ZigBee, tanto en versión normal como Pro, que aumenta el rango de emisión.

Además, Libelium dispone de un *Gateway* que podemos conectar a un puerto USB de nuestro equipo y nos permite realizar una captura de los paquetes que se están transmitiendo. En la Figura 9 podemos observar el *Gateway* con un módulo 802.15.4 montado.

Figura 9. *Gateway* con módulo 802.15.4

El alcance de las comunicaciones de estos módulos viene más o menos fijado dependiendo del protocolo utilizado y de la versión, por ejemplo los módulos normales o PRO en ZigBee, como hemos visto anteriormente. Aunque esto es cierto, algunos de estos módulos, como los ZigBee,

WiFi o Bluetooth no tienen la antena integrada sino que se acopla por medio del puerto especificado para ello. El uso de diferentes antenas puede hacer que el alcance en las comunicaciones varíe al usar un mismo módulo. Aunque las variaciones no son drásticas, no vamos a pasar de un alcance de metros a kilómetros, sí que conviene tenerlo en cuenta en algunos casos.

En nuestro caso, vamos a usar los módulos XBee 802.15.4 normales, fabricados por la empresa "Digi International" [33]. Según la página web de Libelium, estos módulos pueden alcanzar con línea de visión hasta 500 metros. Según la página web de Digi, los módulos pueden alcanzar hasta 30 metros en interior y 90 en exterior. Tomando como referencia la menor distancia de todas, 30 metros, veremos que es superior a la distancia máxima a la que vamos a necesitar enviar que luego veremos que es de aproximadamente 15 metros.

Recientemente, en Febrero de 2013, Libelium ha sacado al mercado la versión 1.2, también llamada versión PRO, de las placas Waspmote. Esta nueva versión de las placas ha recibido cambios tanto a nivel hardware como a nivel software. A nivel hardware nos encontramos por ejemplo con que el procesador ha aumentado su frecuencia de 8MHz a más de 14MHz, los jumpers que tenía la versión 1.1 se han eliminado para que no se rompan, la batería auxiliar se ha eliminado y su función lo cumple la batería principal, y, un cambio muy importante, el conector de la batería principal se ha movido a una posición más accesible. En la versión 1.1 era muy incómodo conectar y desconectar la batería principal, lo que había que hacer para cargar los programas en la placa, lo que causaba que fuese muy fácil que se rompiesen alguno de los cables de la batería. Por el contrario, la mayoría de las placas de sensores de gases, de agricultura, etc. han dejado de ser compatibles con la versión PRO habiendo sacado Libelium nuevas versiones de etas placas. Las nuevas versione de las placas sí que presentan compatibilidad con la versión 1.1 de Waspmote. En las Figuras 10 y 11 podemos observar la parte superior e inferior de la versión PRO de las placas Waspmote donde se observan algunos de los cambios como el del conector de la batería en la parte inferior central.

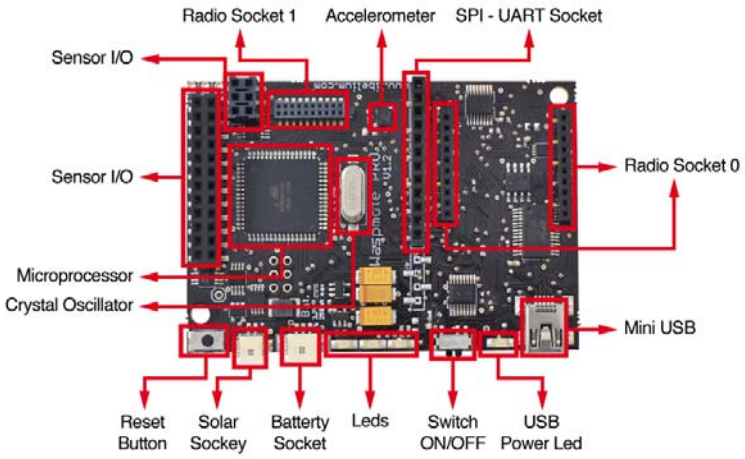


Figura 10. Parte superior de la placa Waspmote 1.2 (PRO)

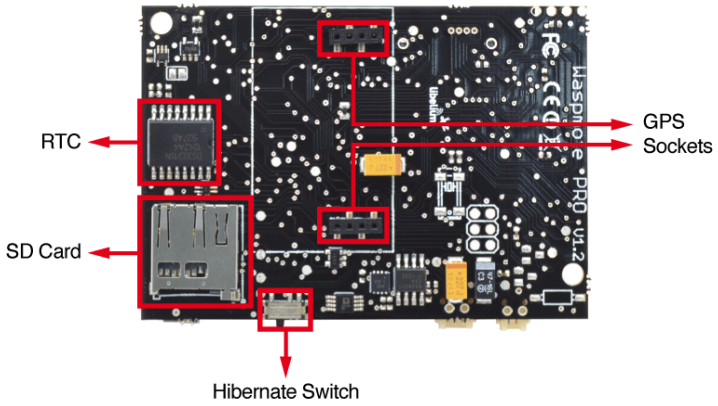


Figura 11. Parte inferior de la placa Waspmote 1.2 (PRO)

Además de los cambios a nivel hardware, la versión 1.2 también ha traído cambios a nivel software. En general, han realizado un cambio en la forma de manejar los archivos de la API con el fin de reducir el tamaño de los programas y evitar gasto innecesario de memoria RAM. Ahora es posible encontrar librerías para cada módulo, por ejemplo el de uso del protocolo 802.15.4, y es necesario cargar cada librería que se desea usar al contrario que antes donde todas las librerías estaban disponibles por defecto. Se ha pasado de una estructura monokernel a otra de microkernel. También se han cambiado algunas de las clases, añadiendo, eliminando o cambiando funciones.

En [34] se puede encontrar un resumen los cambios realizados entre la versión 1.1 y 1.2, tanto a nivel hardware como software.

1.4.2.2.- Otros productos Libelium

La empresa Libelium, además de las placas Waspmote y los productos relacionados con estas, como las placas de sensores o los módulos de comunicaciones, dispone de otros productos relacionados con los sensores.

Cabe destacar el producto Meshlium. A grandes rasgos, Meshlium es un pequeño ordenador que está preparado para conectarle múltiples interfaces. Por ejemplo, le podemos dotar de una antena ZigBee/802.15.4, Bluetooth, WiFi o 3G/GPRS. Además, dispone de un GPS y de un puerto Ethernet. También dispone de métodos de alimentación ininterrumpidos como puede ser la conexión a la red eléctrica, un panel solar o la conexión al mechero de un coche.

Como ordenador, Meshlium dispone de un procesador a 500MHz, 256MB de memoria RAM DDR y entre 8 y 32GB de almacenamiento dependiendo de la opción de Meshlium elegida. Además, utiliza un sistema operativo Linux como es Debian lo que nos permite montar un sistema de base de datos, un servidor Web o cualquier otra cosa que podríamos hacer con un ordenador con Linux en general.

Todo esto hace que sea un complemento ideal para actuar como pasarela entre nuestros sensores Waspmote y otros dispositivos/medios como pueden ser smartphones o Internet. Por ejemplo, el Meshlium podría recoger los datos de la red de sensores que vamos a implementar y presentar estos datos en forma de página Web o suministrar datos a una aplicación Android o iOS de forma que estos datos pudiesen ser consultados desde cualquier lugar en cualquier momento.

En la Figura 12 podemos ver un dispositivo Meshlium junto a una serie de Waspmotes que nos permite hacernos una idea de su tamaño.

Figura 12. Meshlium junto a Waspmotes.

2.- Análisis y diseño

2.1.- Análisis

Como hemos comentado en el Capítulo 1, el objetivo de este trabajo es el de desplegar una red de sensores en una explotación agrícola e implementar una serie de algoritmos de recogida de datos. Una vez realizado esto, realizaremos diferentes mediciones con cada algoritmo y los compararemos para ver cuál puede ser el más adecuado en según qué casos.

En un principio, el tipo de explotación agraria no nos debería influir mucho en las mediciones tomadas siempre que podamos colocar los sensores en una zona más o menos despejada. Por ejemplo, si tuviésemos una plantación de tomates podríamos tener problema de comunicaciones de los sensores si los colocásemos en la raíz de la planta, bajo las hojas, ya que se podría dar que algunos sensores se quedasen sin comunicación con el resto, sobre todo tras regarlos o llover ya que la humedad en general afecta de manera negativa a las comunicaciones. Sobre la misma explotación agraria, si colocásemos los sensores en postes sobre las plantas estos problemas quedarían mitigados o resueltos.

Para nuestro trabajo, hemos elegido una explotación agraria de vides. Las vides están distribuidas sobre la explotación en forma de malla y de forma uniforme sobre el terreno. La explotación no presenta valles o colinas pronunciadas que puedan afectar a las comunicaciones de los sensores sino que la podemos considerar plana.

No hay elementos externos que afecten a las comunicaciones y que nos obliguen a elegir un tipo de algoritmo u otro. Por ejemplo, si tuviésemos una explotación agraria rodeando un bosque o arboleda podría suceder que algunos sensores no se pudiesen comunicar directamente con otros lo que nos obligaría a usar algoritmos donde no todos los sensores tuviesen que tener una comunicación directa.

Como ya hemos dicho anteriormente, en nuestro trabajo vamos a implementar algoritmos que trabajan con sensores dispuestos en una topología en cadena y con algoritmos que trabajan con sensores dispuestos en una topología tipo cluster.

Los algoritmos de cadena se caracterizan porque los datos de los sensores van pasando de un sensor a otro hasta legar a la estación base. Por otra parte, en los algoritmos de cluster se elige un líder, que puede ir variando en el tiempo, al que cada sensor le envía directamente los

datos. Este líder se encargaría, en caso de necesario, de enviar los datos a otro lugar donde serían procesados.

En una topología en cadena, cada sensor recibe paquetes, ya sean de datos, configuración o cualquier otro tipo, de un solo sensor que podemos considerar el nodo o sensor anterior, y envía paquetes a un solo sensor que podemos considerar el nodo o sensor posterior. En general, el orden de los nodos de la cadena se define al iniciar el algoritmo, ya sea por medio de un algoritmo de ordenación y fijando el orden manualmente, y no cambia el orden de los nodos a no ser que alguno de ellos falle y sea necesario rehacer la cadena para que no se produzcan fallos. Dentro de los algoritmos de cadena, uno de los más conocidos es el algoritmo PEGASIS [15] [19]. Este algoritmo trata de crear una cadena desde todos los nodos a un nodo base que será el encargado de recoger los datos. Este nodo base cambia con el tiempo. Para formar la cadena, se intenta que la distancia entre los vecinos sea mínima. Para ello, utiliza la intensidad de señal entre los nodos y elige como vecino aquel con el cuál se comunique usando la menor intensidad de señal posible. De esta forma también se reduce el consumo de batería al necesitar menos potencia para transmitir los mensajes y de esta forma se aumenta la vida de los nodos.

Por otra parte, en las topologías en cluster cada sensor o nodo puede enviar y recibir paquetes de cualquier nodo. El algoritmo inicia con una fase de elección de líder donde intercambian mensajes entre todos los nodos. Una vez elegido el líder, cada nodo enviará sus datos a ese nodo líder. En cada iteración del algoritmo, si este detecta que el líder ha fallado, por ejemplo porque no se reciben paquetes suyos o por que el contador de fallos asociado a ese nodo ha aumentado, el algoritmo elegirá a un nuevo líder. Esto lo hace por medio de un algoritmo de elección de líder. Normalmente se hace llegar de alguna forma a los nodos el número de veces que cada nodo ha fallado y se ha recuperado y se elige al nodo con menor número de fallos usando el identificador para desempatar. Una vez seleccionado el líder, en cada iteración todos los nodos envían sus datos a ese líder. Uno de los algoritmos más conocidos de cluster es el algoritmo Leach [11] [19] [35] [. En este algoritmo, en cada ronda, o cada cierto tiempo, se elige de forma aleatoria P líderes de cluster usando algún algoritmo estocástico. Los nodos se unen al líder que tengan más cerca, por ejemplo haciendo uso de la potencia de transmisión como en PEGASIS. Los diferentes nodos unidos a un mismo líder forman un cluster. El líder les asigna a cada uno una franja de transmisión y cuando ha recibido los datos de los nodos los comprime y los envía a un sumidero o recolector de datos.

Un algoritmo que trabaje con una topología de cadena tiene como ventaja que el número de paquetes enviados y recibidos es mínimo. A no ser que haya que realizar una reconfiguración de la cadena por algún motivo, cada nodo sólo recibe un paquete del nodo anterior y envía un paquete al nodo posterior. Incluso en el caso de tener que reconfigurar la cadena porque un nodo falle o vaya a fallar, sólo es necesario avisar al nodo anterior y posterior. Esto hace que el consumo energético usado en las comunicaciones, que es el factor que más energía consume, sea muy pequeño en comparación con otros algoritmos. Como inconveniente, una topología en cadena es muy sensible a fallos en los nodos. Si un nodo falla va a causar que la cadena se rompa y que los datos no lleguen hasta el final de la cadena causando que no se recoja ningún dato. Sería necesario implementar algún tipo de mecanismo de detección de fallos en nodos y de reconfiguración de la cadena. Veremos una idea de cómo hacer esto en el Capítulo 5. También es más difícil añadir nodos a la topología ya que habría que avisar al menos al nodo anterior, en caso de situarlo al final de la cadena, de que tiene un nuevo vecino y que no es el final de la cadena.

A diferencia de las topologías en cadena, el número de paquetes enviados en cada iteración de un algoritmo que trabaje con una topología en cluster es, por regla general, mayor lo que va a causar un mayor consumo energético. Aunque esto puede variar dependiendo del algoritmo de cluster elegido, en general va a ser necesario en cada iteración que cada nodo envíe un paquete y reciba paquetes del resto de nodos para elegir al líder y además deberá enviar un

paquete al líder con los datos del nodo correspondiente. Por el contrario, la topología en cluster es mucho más robusta ante fallos. Aunque un nodo falle, el resto de nodos va a seguir trabajando normalmente. Sólo en el caso en que el nodo que falle sea el líder va a afectar al resto de la topología ya que en esa iteración no se recogerán datos pero en la siguiente iteración se elegirá a un nuevo nodo como líder y el algoritmo seguirá funcionando con normalidad. Además, la adición de un nuevo nodo a la topología es más sencilla ya que simplemente, en la siguiente ronda de elección de líder se dará a conocer al resto de nodos pudiendo ser inmediatamente elegido como líder, en caso de cumplir con los requisitos

2.2.- Diseño

Una vez que hemos analizado el problema a resolver, el siguiente paso es diseñar los diferentes elementos que nos van a permitir resolver el problema planteado. Analizando el problema hay dos elementos sobre los que hay que tomar decisiones a la hora de diseñar la solución que queremos emplear. El primer elemento es la colocación de los sensores sobre la explotación agraria. El segundo son los diferentes algoritmos de recogida de datos que vamos a implementar y comparar.

2.2.1.- Topología y colocación de los sensores

Como hemos visto en el apartado anterior, las características de la explotación agraria no nos limitar en ningún sentido a la hora de diseñar cómo implantar nuestra red de sensores y de implementar el algoritmo de recogida de datos.

Cuanta mayor es la distancia entre sensores mayor es la energía necesaria para enviar los mensajes. Con el fin de que el gasto energético sea similar en todos los sensores, he decidido situar los sensores en una malla uniforme tipo Manhattan. La estación base estará situada aproximadamente en el centro de la malla para estar situada lo más cerca de todos los sensores ya que, en los algoritmos de cluster, no sabemos cuál va a ser el sensor encargado de recoger los datos y transmitirlos a la estación base. En la Figura 13 podemos ver un esquema de cómo quedarían repartidos los sensores y la numeración que le damos a cada uno de ellos. En un principio la numeración de los sensores podría ser cualquiera siempre y cuando no se repitan ya que, como veremos más adelante, los algoritmos de cluster obligan a que el identificador de cada nodo sea único.

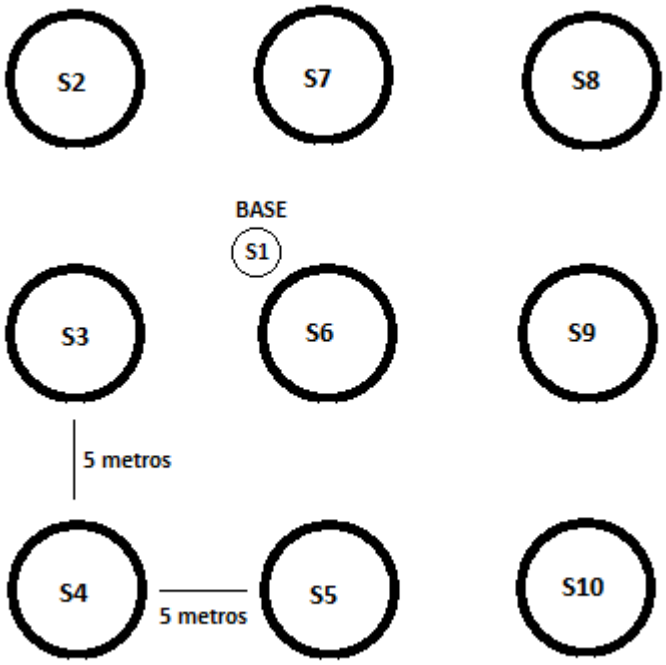


Figura 13. Esquema de la disposición de los sensores.

Como ya hemos dicho anteriormente, los sensores hay que colocarlos en una zona de la explotación donde no estén especialmente tapados.

2.2.2.- Algoritmos a implementar

Una vez que hemos decidido cómo colocar los sensores dentro de la explotación agraria, vamos a presentar los algoritmos de recogida de datos que vamos a implementar en nuestra red de sensores. Los algoritmos van a estar separados en dos grupos, los algoritmos de cadena y los algoritmos de cluster; usando uno y tres de cada tipo respectivamente.

2.2.2.1.- Algoritmos de cadena

En los algoritmos en cadena, cada sensor recibe una trama con datos del sensor anterior, agrega sus propios datos a la trama, en este caso el valor de la batería que, como ya hemos dicho, es el dato que vamos a recoger, y envía la trama al sensor siguiente. La forma de elegir el sensor anterior y siguiente, es decir, la forma de definir la cadena, es lo que diferencia a unos algoritmos de otros. Para la realización de este trabajo, hemos elegido usar el algoritmo S-DAWIN presentado en el artículo "S-DAWIN: A Self-adapted Distributed Algorithm for data gathering in Wireless sensor Networks" [17].

Como se puede deducir del título del artículo, el nombre del algoritmo proviene de las siglas en inglés de "algoritmo distribuido auto-adaptable para la toma de datos en redes de sensores inalámbricas". Aunque en el artículo hacen una referencia a cómo poder hacer que cada nodo conozca sus vecinos de forma autónoma, al no disponer de un pseudo-algoritmo, a la hora de implementar nuestro algoritmo he asumido que cada sensor conoce previamente a sus vecinos y que no se pueden añadir más nodos una vez iniciada la ejecución del algoritmo.

Teniendo en cuenta que, dada la naturaleza para la que se va a utilizar la red de sensores, normalmente una vez desplegada la red de sensores no se añaden más sensores a la misma, la restricción no parece muy importante. En el artículo podemos encontrar el modelo de sistema que se va a utilizar tomando, entre otras, las siguientes asunciones, teniendo en cuenta que vamos a usar sensor o nodo de la cadena como sinónimos:

1. Los sensores se pueden comunicar directamente unos con otros y con la estación base.
2. Los sensores envían sus datos al siguiente nodo usando sólo su información local.
3. La información de cada sensor se tiene que recoger en cada iteración finalizada del algoritmo.
4. Los sensores tienen control de energía.

La asunción número 4 es muy importante ya que, como veremos más adelante, en caso de detectar que un sensor se va a quedar sin batería, el propio algoritmo es capaz de reconfigurar la cadena para que el apagado del sensor no implique el corte en la cadena.

Los diferentes algoritmos de cadena configuran la cadena de sensores de forma diferente. En la Figura 14 podemos ver la configuración de la cadena en tres algoritmos como son SFC, PEGASIS y DAWIN. En [36] podemos encontrar una sencilla explicación de las curvas de Peano y Hilbert. Estas curvas son un caso concreto de las *space-filling curves* que son las que se implementan en los algoritmos SFC. En la curva de Peano, cada segmento será remplazado por otros. Comenzando con un intervalo, este se sustituye por una curva poligonal autointersecante formada por nueve segmentos iguales. Este proceso se repite en cada uno de estos nueve segmentos continuando el proceso indefinidamente. Para construir la curva de Hilbert se procede así: partimos del cuadrado unidad dividido en cuatro partes iguales y se unen sus centros formando una U invertida. Seguidamente, se divide cada uno de los cuadrados en cuatro partes y se repite el proceso; se conectan sus centros, comenzando siempre por el cuadrado inferior izquierdo y terminando en el cuadrado inferior derecho. Este proceso se repite indefinidamente y se obtiene la curva de Hilbert.

En el caso de LEACH, la estación base elige de forma aleatoria un nodo que a ser el inicializador de la cadena. Este nodo comprueba que nodo tiene más "cerca". Para ello comprueba cuál es el nodo para el que necesita utilizar menos intensidad de señal, lo que implica menor uso energético, para enviarle mensajes y lo elige como siguiente nodo. Este a su vez hace lo mismo hasta completar una cadena que implique a todos los nodos de la red. Una vez formada la cadena, la estación base envía un mensaje inicializador al nodo elegido como primero para iniciar la recogida de datos. El mensaje va pasando por todos los nodos, siguiendo la cadena que se ha formado anteriormente, hasta llegar al último nodo que envía el mensaje con los datos de todos los nodos a la estación base. Cada cierto tiempo, la estación base elige a otro nodo como inicializador de la cadena y se repite el proceso de creación de la cadena, con lo que se obtiene una nueva cadena.

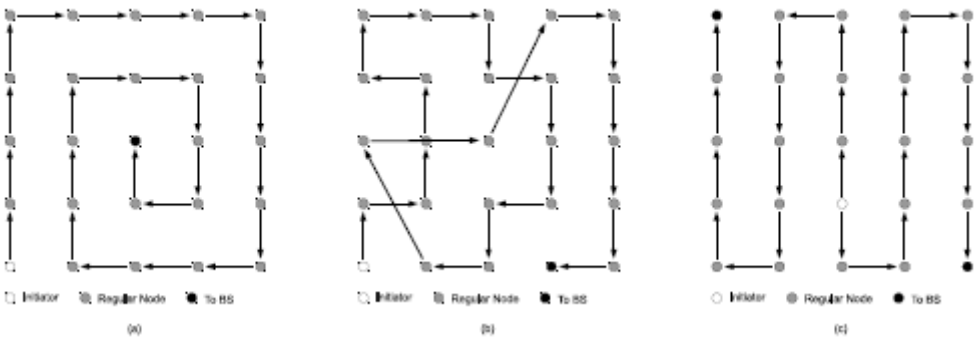


Figura 14. Cadena para los algoritmos (a) SFC, (b) PEGASIS y (c) DAWIN

La principal diferencia del algoritmo DAWIN con la definición que hemos visto de PEGASIS es que, mientras que PEGASIS rehace la cadena cada cierto tiempo, DAWIN sólo la rehace en caso de que se detecte que algún nodo se va a quedar sin batería y sólo implica a sus nodos vecinos en vez de a todos los nodos de la red como sí hace PEGASIS. Además, en ningún momento se espera a que un nodo se quede completamente sin batería sino que se rehace la cadena antes de llegar a esa situación y se realiza un apagado controlado del nodo.

En cada iteración del algoritmo DAWIN, la estación base envía una trama "inicializadora" a un nodo de la cadena de su elección que puede ser cualquiera entre los nodos que forman la red de sensores. Este nodo se encarga de enviar un mensaje o trama *forward* y otro *backward*. En este mensaje, de alguna forma, tiene que hacer saber al siguiente nodo la dirección en la que va la cadena, *forward* o *backward* y de qué nodo se envía el mensaje, el nodo de arriba, abajo, izquierda o derecha. A partir de ese momento, y con la información que hemos indicado, cada nodo decidirá cuál es el siguiente nodo al que tiene que enviar el mensaje. Añadirá su propia información al mensaje y lo enviará al siguiente nodo. Cuando un nodo decida que no tiene un nodo siguiente al que enviar la información, enviará el mensaje con los datos de todos los nodos a la estación base donde se podrán procesar.

Como ya podemos ver, la elección del siguiente nodo al que enviar el mensaje es una de las partes más importantes del algoritmo. En las Tablas 3 y 4 podemos ver las tablas de decisión de los nodos. Dependiendo de dónde haya recibido el mensaje, arriba, abajo, izquierda o derecha, de los vecinos que tenga y de la dirección del mensaje, el nodo tomará la decisión de cuál es su siguiente nodo al que enviar. La Tabla 3 muestra la tabla de decisión *forward* mientras que la Tabla 4 muestra la tabla de decisión *backward*. Como resumen podemos decir que en la dirección *forward* se intenta ir hacia abajo o a la derecha mientras que *backward* se intenta ir hacia arriba o a la izquierda.

BS	up_i	$down_i$	$left_i$	$right_i$	$next_i$
$source$	-	$down_i$	-	-	$down_i$
$source$	-	nil	-	$right_i$	$right_i$
$source$	-	nil	-	nil	BS
-	$source$	$down_i$	-	-	$down_i$
-	$source$	nil	-	$right_i$	$right_i$
-	$source$	nil	-	nil	BS
-	up_i	$source$	-	-	up_i
-	nil	$source$	-	$right_i$	$right_i$
-	nil	$source$	-	nil	BS
-	up_i	nil	$source$	-	up_i
-	nil	$down_i$	$source$	-	$down_i$
-	nil	nil	$source$	-	BS
-	-	-	-	$source$	BS
-	-	-	-	$source$	BS
-	-	-	-	$source$	BS

Tabla 3. Tabla de decisión *forward*

BS	up_i	$down_i$	$left_i$	$right_i$	$next_i$
$source$	up_i	-	-	-	up_i
$source$	nil	-	$left_i$	-	$left_i$
$source$	nil	-	nil	-	BS
-	$source$	$down_i$	-	-	$down_i$
-	$source$	nil	$left_i$	-	$left_i$
-	$source$	nil	nil	-	BS
-	up_i	$source$	-	-	up_i
-	nil	$source$	$left_i$	-	$left_i$
-	nil	$source$	nil	-	BS
-	-	-	$source$	-	BS
-	-	-	$source$	-	BS
-	-	-	$source$	-	BS
-	up_i	nil	-	$source$	up_i
-	nil	$down_i$	-	$source$	$down_i$
-	nil	nil	-	$source$	BS

Tabla 4. Tabla de decisión *backward*

Una vez que hemos visto la forma en que cada nodo decide cuál va a ser su siguiente nodo al que enviar el mensaje, en las Figuras 15 y 16 vamos a ver el pseudo-algoritmo para las funciones *rcvBEACON* y *rcvAGGREGATE*. La primera sería la que se ejecutase en el nodo que reciba la trama "inicializadora" de la estación base por lo que se encargaría de enviar los menajes para crear las dos sub-cadenas. La segunda función es la que se ejecuta en cualquier otro nodo de la cadena por lo que simplemente tiene que decidir el siguiente nodo y enviar los datos. Estas funciones están pensadas para poder ejecutarse con los algoritmos SFC y PEGASIS que hemos visto brevemente por lo que se usa la variable *policy* para diferenciar el algoritmo a ejecutar. Como en nuestro caso siempre ejecutamos el algoritmo DAWIN, esta variable se puede obviar.

$rcvBEACON(policy, fwd, bwd)$
{
  IF fwd **THEN**
    $next_i \leftarrow selectNext(BS, policy, fwd, false)$
    IF $next_i \neq nil$ **THEN**
      send $AGM[i, data_i, policy, fwd, false]$ to $next_i$
    END IF
  END IF
  IF $fwd \wedge bwd$ **THEN**
    $wait\ t_{wait}$
  END IF
  IF bwd **THEN**
    $next_i \leftarrow selectNext(BS, policy, false, bwd)$
    IF $next_i \neq nil$ **THEN**
      send $AGM[i, data_i, policy, false, bwd]$ to $next_i$
    END IF
  END IF
  IF $next_i = nil$ **THEN**
    send $AGM[i, data_i, policy, false, bwd]$ to BS
  END IF
}

Figura 15. Función rcvBEACON a ejecutar en el primer nodo

```
rcvAGGREGATE(source, data, policy, fwd, bwd)
{
    // Update message by aggregating local data (data_i)
    // to the data carried by the message
    data ← aggregateData(data_i, data)
    next_i ← selectNext(source, policy, fwd, bwd)
    IF next_i ≠ nil THEN
        send AGM[i, data, policy, fwd, bwd] to next_i
    ELSE
        send AGM[i, data_i, policy, false, bwd] to BS
    END IF
}
```

Figura 16. Función rcvAGGREGATE a ejecutar en el resto de nodos

En todo momento hemos hablado del algoritmo DAWIN y no del algoritmo S-DAWIN. Esto es así ya que tal y como hemos planteado el algoritmo, este no es auto-adaptable. Con el fin de asegurar que un sensor, en caso de tener poca batería y apagarse, corte la cadena, vamos a añadir un mecanismo que permita a la cadena reconfigurarse en este caso para ignorar al sensor con poca batería. El objetivo es que, cuando un sensor detecta que tiene la batería por debajo de un nivel crítico marcado por nosotros, envíe un mensaje a sus vecinos haciéndoles partícipes de que se va a apagar y reconfiguren sus vecinos de forma que la cadena no se rompa. Por ejemplo, si el sensor i que se va a apagar tiene vecino superior e inferior, al superior le haría saber que su nuevo sensor inferior en vez de ser él mismo va a ser el actual vecino inferior de i y viceversa. En la Figura 17 podemos ver dos ejemplos de reconfiguración de la cadena dependiendo de los vecinos que tiene.

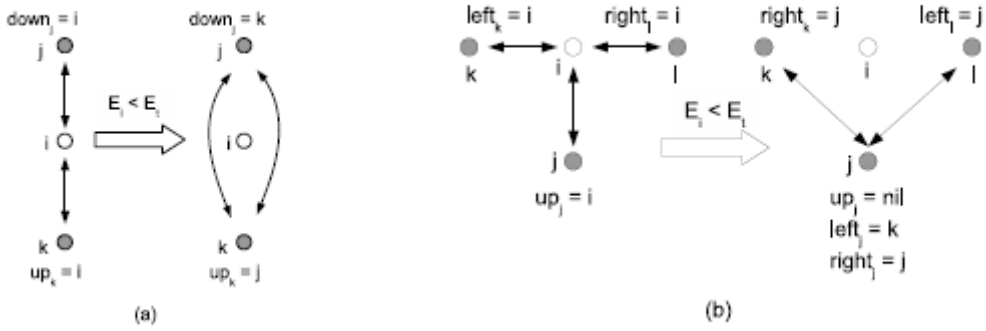


Figura 17. Ejemplos de reconfiguración de la cadena

Una vez que hemos visto la idea que queremos obtener con la reconfiguración, en las funciones de *rcvBEACON* y *rcvAGGREGATE* hay que añadir una simple comprobación del nivel de batería del nodo y, en caso de ser menor a un umbral marcado por nosotros mismos, poner en marcha el proceso de reconfiguración. El proceso de reconfiguración son dos funciones que podemos ver en las Figuras 18 y 19. La primera sería la función que se ejecutaría cuando comprobamos que el nivel de batería es inferior al umbral mientras que la segunda es la función que ejecutaríamos al recibir una trama de reconfiguración de cadena.

$$
\begin{array}{l}
reconfigureCHAIN()\\
\{\\
\quad \textbf{IF } up_i = nil \textbf{ THEN}\\
\quad\quad \text{send } RDN[nil, \perp, left_i, right_i] \text{ to } down_i\\
\quad\quad \text{send } RDN[\perp, \perp, \perp, down_i] \text{ to } left_i\\
\quad\quad \text{send } RDN[\perp, \perp, down_i, \perp] \text{ to } right_i\\
\quad \textbf{END IF}\\
\quad \textbf{IF } down_i = nil \textbf{ THEN}\\
\quad\quad \text{send } RDN[\perp, nil, left_i, right_i] \text{ to } up_i\\
\quad\quad \text{send } RDN[\perp, \perp, \perp, up_i] \text{ to } left_i\\
\quad\quad \text{send } RDN[\perp, \perp, up_i, \perp] \text{ to } right_i\\
\quad \textbf{END IF}\\
\quad \textbf{IF } up_i \neq nil \wedge down_i \neq nil \textbf{ THEN}\\
\quad\quad \text{send } RDN[up_i, \perp, \perp, \perp] \text{ to } down_i\\
\quad\quad \text{send } RDN[\perp, down_i, \perp, \perp] \text{ to } up_i\\
\quad \textbf{END IF}\\
\}
\end{array}
$$

Figura 18. Función de envío de trama de reconfiguración de cadena

$$
\begin{array}{l}
rcvRDN(up, down, left, right)\\
\{\\
\quad \textbf{IF } up_i \neq \perp \textbf{ THEN } up_i = up \textbf{ END IF}\\
\quad \textbf{IF } down_i \neq \perp \textbf{ THEN } down_i = down \textbf{ END IF}\\
\quad \textbf{IF } left_i \neq \perp \textbf{ THEN } left_i = left \textbf{ END IF}\\
\quad \textbf{IF } right_i \neq \perp \textbf{ THEN } right_i = right \textbf{ END IF}\\
\}
\end{array}
$$

Figura 19. Función de recepción de trama de reconfiguración de cadena

A simple vista podemos ver cómo, independientemente del número de nodos de la cadena, cada nodo sólo envía y recibe un mensaje, dos en el caso del nodo "inicializador" de la cadena. Esto es importante en caso de utilizar redes con muchos nodos ya que va a permitir ahorrar en el envío y recepción de mensajes que es uno de los procesos que más energía consumen.

Hemos visto que la cadena se reconfigura cuando un sensor detecta que se va a apagar por falta de batería. La cadena no se reconfigura ante fallos repentinos y permanentes de un nodo, como roturas, lo que podría causar que la cadena quedase rota indefinidamente. En el Capítulo 5 plantearemos cómo poder reconfigurar la cadena ante este hecho.

2.2.2.2.- Algoritmos de cluster

Los algoritmos en cluster tienen la característica de que, en cada iteración, se elige un nodo como líder que puede ser el mismo nodo que en la iteración anterior o un nodo distinto. Así como en el algoritmo en cadena visto anteriormente, el fallo en un sensor puede causar que no se recojan datos, en este caso el fallo de un sensor no va a afectar de manera indefinida a la recogida de datos en el resto. Cada sensor puede fallar y recuperarse un número indeterminado

de veces. Cada sensor va a llevar la cuenta del número de veces que ha fallado y se ha recuperado y se lo hará saber al resto de nodos de una forma u otro y se elegirá como líder al nodo que menos veces haya fallado. La forma de hacer llegar al resto de nodos esta información es lo que va a diferenciar principalmente a unos algoritmos de otros.

En nuestro trabajo, vamos a implementar lo algoritmos presentados en el artículo "Fault-tolerant aggregator election and data aggregation in wireless sensor networks" [13]. Como se puede deducir del título, y como hemos dicho anteriormente, los algoritmos van a ser tolerantes a fallos por lo que el fallo en un sensor no va a afectar de forma definitiva al resto.

Atendiendo al estado de los sensores ante fallos, el artículo presenta tres tipos de sensores:

- *Eventually up* (al final levantado): Nodo que, tras fallar y recuperar un número indeterminado de veces, se mantiene funcionando hasta el final de su batería.
- *Eventually down* (al final caído): Nodo que, tras fallar y recuperarse un número indeterminado de veces, se mantiene caído hasta el final de su batería.
- *Unstable* (inestable): Nodo que, hasta el final de su batería, está continuamente cayendo y recuperándose.

Por definición, los primeros serían nodos correctos mientras que el resto serían incorrectos. Suponemos que al menos un nodo es correcto dentro de la red.

En este artículo se presentan tres algoritmos dependiendo del grado de "visión" entre nodos, tomando como visión el número de nodos al que puede llegar cada nodo de la red. Los tres grados de visión que vamos a presentar y que van a definir cada algoritmo son los siguientes:

- Cada nodo se puede comunicar directamente con el resto de nodos de la red.
- Al menos un nodo correcto se puede comunicar directamente con todos los nodos de la red.
- Existe un camino multi-salto bidireccional entre un nodo correcto y el resto de nodos de la red.

A diferencia del algoritmo en cadena, en los algoritmos en cluster que vamos a presentar no es necesario que cada nodo conozca previamente al resto. Además, se podrían añadir más nodos a la red una vez iniciada la ejecución. En caso de añadirse un sensor "no correcto" esto podría causar una desestabilización temporal en la red y que dejase de recoger datos durante un tiempo pero con el tiempo se volvería a estabilizar y volvería a funcionar correctamente. Sí que es necesario que cada nodo tenga un identificador y que éste sea único dentro de la red. Este identificador se va a usar en algunos casos para elegir al líder del cluster.

Aunque en el artículo dividen los sensores en regiones y presentan algoritmos de comunicación entre regiones, en nuestro trabajo, al no trabajar con muchos nodos ni muy separados, vamos a considerar que todos los nodos están en la misma región y sólo vamos a implementar los algoritmos que se encargan de recoger los datos dentro de una región.

2.2.2.2.1.- Primer algoritmo

El primer algoritmo de cluster que vamos a implementar se caracteriza por que los sensores van a estar hibernando la mayor parte del tiempo y sólo se despertarán en los periodos de tiempo que planifiquemos. Durante el periodo de tiempo que estén despiertos será cuando elijan al líder y le envíen la información. De esta forma vamos a lograr que la batería de los sensores dure más tiempo ya que la mayor parte del tiempo estarán sin actividad. Por el contrario, necesitamos

algún sistema que despierte a los sensores cundo nosotros queramos pero, como veremos en el Capítulo 3, los sensores Waspmote disponen de un sistema que, mediante una pila, podemos hacer que el sensor hiberne y se despierte cuando nosotros queramos. Este algoritmo también necesita que todos los nodos se puedan comunicar directamente entre ellos. Como la distancia entre las esquinas del esquema que hemos planteado en el apartado 2.2.1 es de menos de 15 metros en un entorno abierto, esta distancia es perfectamente alcanzable por el módulo de comunicaciones que vamos a utilizar. En la Figura 20 podemos ver un esquema básico de las operaciones que realiza este algoritmo a lo largo del tiempo donde podemos comprobar que la mayor parte del tiempo el sensor está dormido y sólo se despierta en los tiempos planificados. Cada cierto tiempo, en los momentos planificados, es sensor se despierta y espera un tiempo ε para permitir que el resto de sensores se despierten, con lo que compensamos pequeñas diferencias de desincronización en los relojes y tiempos planificados de actividad del nodo. Pasado ese tiempo, el sensor empieza su tiempo de adquisición de datos donde espera por mensajes donde el resto de nodos le pueden informar del número de veces que han fallado y se han recuperado para de esta forma poder elegir al líder. Una vez pasado este tiempo, el nodo comprueba si es líder del cluster o no. Si lo es, espera a recibir datos del resto de nodos. Si no lo es, envía sus datos al nodo que haya elegido como líder. Una vez hecho esto planifica la próxima vez que se va a despertar y entra en hibernación. Este proceso se repite cíclicamente,

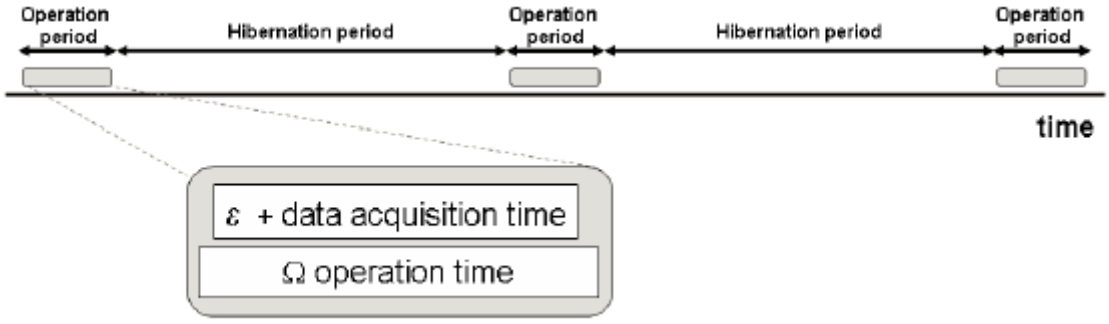


Figura 20. Esquema de tiempo básico del funcionamiento del primer algoritmo de cluster

El funcionamiento básico del algoritmo se basa en que cada sensor guarda en un almacenamiento permanente de forma que no se borre en caso de fallo, que veremos que en nuestro caso será la tarjeta micro SD del sensor, las veces que se ha recuperado de un fallo. En cada iteración los sensores comunican al resto las veces que han fallado y cada sensor elige como líder aquel sensor con menor número de fallos al considerarlo el más estable. En un principio, como todos los nodos se comunican con todos, en cada ronda todos los sensores eligen al mismo líder y le envían sus datos. El sensor líder se encargaría, en caso necesario, de enviar la información recogida de todos los nodos a una estación base donde poder procesarlos. En la Figura 21 podemos ver el pseudo-algoritmo de este primer algoritmo de cluster.

Every sensor p executes the following:

procedure *GoToHibernation*()
(1) write ($incarnation_p$, $leader_p$, $incarnation_{leader}$, $Timeout_p$) in stable storage
(2) $scheduled\_wakeup_p \leftarrow$ TRUE
(3) *hibernate*()
end procedure

Initialisation:
(4) read ($incarnation_p$) from stable storage
(5) if [$scheduled\_wakeup_p = FALSE$] then
(6) $incarnation_p \leftarrow incarnation_p + 1$
(7) write $incarnation_p$ in stable storage
(8) end if
(9) $scheduled\_wakeup_p \leftarrow FALSE$
(10) read ($leader_p$, $incarnation_{leader}$, $Timeout_p$) from stable storage
(11) if [$leader_p = p$] then
(12) start tasks 1 and 2
(13) else
(14) reset $timer_p$ to $Timeout_p + 2\epsilon$
(15) start tasks 2 and 3
(16) end if

Task 1
(17) wait ϵ time units
(18) broadcast (I-AM-ALIVE, p, $incarnation_p$)
(19) receive data from sensors during $\Delta_{DATA\_ACQUISITION}$ time
(20) *GoToHibernation*()

Task 2:
(21) upon reception of message (I-AM-ALIVE, q, $incarnation_q$) such that
[$incarnation_q < incarnation_{leader}$] or
[($incarnation_q = incarnation_{leader}$) and ($q \leq leader_p$)] do
(22) $leader_p \leftarrow q$
(23) $incarnation_{leader} \leftarrow incarnation_q$
(24) $data_p \leftarrow$ acquire sensed data
(25) send ($data_p$) to $leader_p$
(26) *GoToHibernation*()

Task 3:
(27) upon expiration of $timer_p$ do
(28) $leader_p \leftarrow p$
(29) $incarnation_{leader} \leftarrow incarnation_p$
(30) $Timeout_p \leftarrow Timeout_p + \Delta_{TIMEOUT}$
(31) *GoToHibernation*()

Figura 21. Pseudo-código del primer algoritmo

La tarea de inicialización se encarga de configurar las variables a los valores adecuados cuando se inicia el algoritmo y cada vez que el sensor se recupera de un fallo. La tarea 1 se ejecuta cuando el sensor es líder y se encarga de recoger los datos del resto de nodos. La tarea 2 se encarga de comprobar si hay algún nodo que merezca convertirse en líder. La tarea 3 se ejecuta cuando no encontramos ningún candidato a convertirse en líder pasado un tiempo por lo que convertimos al propio nodo en líder para que el algoritmo no deje de funcionar. En el Capítulo 3 veremos que los sensores Waspmote no admiten la ejecución de hilos por lo que habrá que modificar el algoritmo para poder implementarlo en los sensores Waspmote. En el artículo se demuestra que el pseudo-código es correcto.

2.2.2.2.2.- Segundo algoritmo

Este segundo algoritmo parte de la misma idea básica de elegir como líder a aquel nodo que menos falle, es decir, al más fiable. A diferencia del primer algoritmo, en este caso no todos los nodos van a ser candidatos a convertirse en líderes sino que lo van a ser sólo aquellos nodos que tengan comunicación directa con el resto de nodos de la red. Además, en este caso los nodos no van a dormirse por lo que, a priori, el consumo energético de los nodos va a crecer, y por lo tanto la duración de la batería va a ser menor que en el algoritmo primero.

Para asegurar que sólo los nodos con comunicación directa con el resto puedan ser líderes, cada nodo mantienen una tabla con tantas posiciones como nodos hay en la red. Esta tabla en un principio está iniciada a 0 para todos los nodos y se puede considerar una tabla de fallos. Los nodos están constantemente enviando mensajes indicando que están vivos. Si pasado un tiempo determinado un nodo no ha recibido mensaje de viveza de otro, incrementa el valor en 1 en su tabla para ese determinado nodo. Junto con el mensaje de viveza, cada nodo envía su tabla al resto y, al recibir el mensaje, actualizan su taba para cada nodo quedándose con el máximo entre el valor que ya tenían y el recibido. Se puede ver como si un nodo i no tiene comunicación con un sólo nodo de la red, por ejemplo k, aunque si tenga comunicación con el resto, el nodo k cada vez que pase el tiempo establecido aumentará el fallo para i en su tabla. De esta forma, rápidamente aumentará el número de fallos para i en k y, al enviar el nodo k esta

información al resto de nodos, estos actualizan su información del número de fallos de i al valor enviado por k haciendo que no sea un buen candidato a ser elegido líder. Es por esto que a la larga sólo vamos a elegir como líderes a aquellos nodos que tengan comunicación con todos los nodos de la red. Cada vez que el nodo se recupera de un fallo, o la primera vez, aumentamos en 1 el número de fallos del propio nodo y lo guardamos en almacenamiento permanente. De esta forma nos aseguramos que si un nodo está continuamente fallando no sea elegido como líder y no desestabilice el comportamiento del algoritmo. Con el fin de ahorrar mensajes, cada nodo envía sus datos junto con el mensaje de viveza de forma que el nodo líder pueda recoger estos datos sin tener que esperar a recibir un mensaje específico con los datos. En la Figura 22 podemos ver el pseudo-código para este segundo algoritmo de cluster.

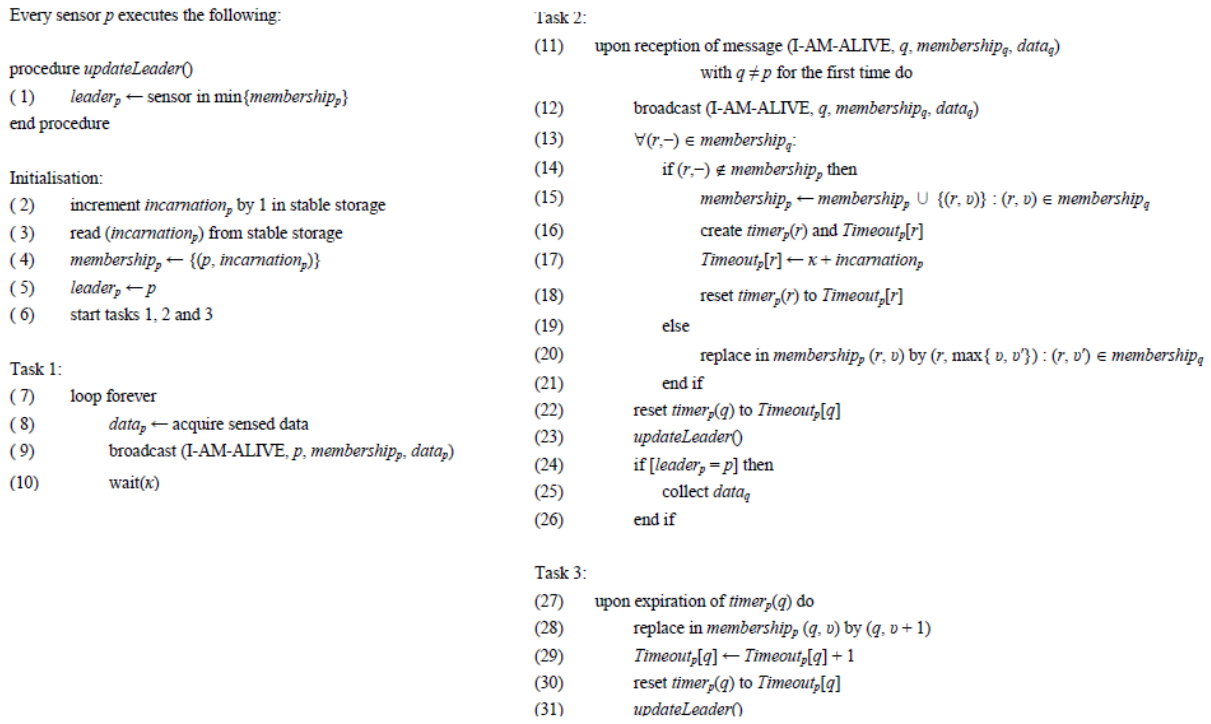
```
Every sensor p executes the following:

procedure updateLeader()
(1)   leader_p ← l such that counter_p[l] = min{counter_p}
end procedure

Initialisation:
(2)   increment incarnation_p by 1 in stable storage
(3)   read (incarnation_p) from stable storage
(4)   ∀q ≠ p : Timeout_p[q] ← η + incarnation_p
(5)   ∀q ≠ p : reset timer_p(q) to Timeout_p[q]
(6)   ∀q ≠ p : counter_p[q] ← 0
(7)   counter_p[p] ← incarnation_p
(8)   leader_p ← p
(9)   start tasks 1, 2 and 3

Task 1:
(10)  loop forever
(11)      data_p ← acquire sensed data
(12)      broadcast (I-AM-ALIVE, p, counter_p, data_p)
(13)      wait(η)

Task 2:
(14)  upon reception of message (I-AM-ALIVE, q, counter_q, data_q) do
(15)      reset timer_p(q) to Timeout_p[q]
(16)      ∀r : counter_p[r] ← max{(counter_p[r], counter_q[r])}
(17)      updateLeader()
(18)      if [leader_p = p] then
(19)          collect data_q
(20)      end if

Task 3:
(21)  upon expiration of timer_p(q) do
(22)      counter_p[q] ← counter_p[q] + 1
(23)      Timeout_p[q] ← Timeout_p[q] + 1
(24)      reset timer_p(q) to Timeout_p[q]
(25)      updateLeader()
```

Figura 22. Pseudo-código del segundo algoritmo

Al igual que antes, la tarea de inicialización configura las variables que vamos a utilizar. La tarea 1 se encarga de enviar los mensajes de viveza al resto de nodos. La tarea 2 se encarga de actualizar la tabla de fallos con los datos recibidos y de recoger datos en caso de ser considerado líder. La tarea 3 se encarga de aumentar el contador de fallo para cada nodo en caso de no recibir un mensaje de viveza dentro del tiempo determinado. Al igual que antes, en el artículo se demuestra la corrección del algoritmo.

2.2.2.2.3.- Tercer algoritmo

Este tercer algoritmo es bastante similar al segundo pero en este caso no es necesario que algún nodo alcance directamente al resto si no que es necesario con que exista un camino muti-salto desde un nodo al resto de nodos. Para conseguir esto, cada vez que un nodo recibe un mensaje de viveza de otro por primera vez, renvía este mensaje al resto de nodos para intentar alcanzar a todos. La idea de guardar el número de fallos y recuperaciones de cada nodo y de aumentar el número de fallos de un nodo cuando ha pasado un determinado tiempo funciona de forma similar a lo que lo hacía en el algoritmo tercero. En la Figura 23 podemos ver el pseudo-código de este tercer algoritmo de cluster.



```
Every sensor p executes the following:

procedure updateLeader()
( 1)    leader_p ← sensor in min{membership_p}
end procedure

Initialisation:
( 2)    increment incarnation_p by 1 in stable storage
( 3)    read (incarnation_p) from stable storage
( 4)    membership_p ← {(p, incarnation_p)}
( 5)    leader_p ← p
( 6)    start tasks 1, 2 and 3

Task 1:
( 7)    loop forever
( 8)        data_p ← acquire sensed data
( 9)        broadcast (I-AM-ALIVE, p, membership_p, data_p)
(10)        wait(x)

Task 2:
(11)    upon reception of message (I-AM-ALIVE, q, membership_q, data_q)
                with q ≠ p for the first time do
(12)        broadcast (I-AM-ALIVE, q, membership_q, data_q)
(13)        ∀(r,–) ∈ membership_q:
(14)            if (r,–) ∉ membership_p then
(15)                membership_p ← membership_p ∪ {(r, v)} : (r, v) ∈ membership_q
(16)                create timer_p(r) and Timeout_p[r]
(17)                Timeout_p[r] ← x + incarnation_p
(18)                reset timer_p(r) to Timeout_p[r]
(19)            else
(20)                replace in membership_p (r, v) by (r, max{ v, v'}) : (r, v') ∈ membership_q
(21)            end if
(22)        reset timer_p(q) to Timeout_p[q]
(23)        updateLeader()
(24)        if [leader_p = p] then
(25)            collect data_q
(26)        end if

Task 3:
(27)    upon expiration of timer_p(q) do
(28)        replace in membership_p (q, v) by (q, v + 1)
(29)        Timeout_p[q] ← Timeout_p[q] + 1
(30)        reset timer_p(q) to Timeout_p[q]
(31)        updateLeader()
```

Figura 23. Pseudo-código del tercer algoritmo

La tarea de inicialización y las tareas 1 y 3 funcionan de forma similar a como lo hacían en el anterior algoritmo. La tarea 2, además de realizar la misma función que antes, también renvía el mensaje recibido en caso de ser necesario. De nuevo, en el artículo podemos encontrar la demostración de corrección del algoritmo.

3.- Desarrollo

Una vez que hemos presentado los algoritmos que vamos a implementar, a continuación vamos a ver el software que vamos a utilizar para implementar estos algoritmos y la forma de implementarlos.

3.1.- Software utilizado

A la hora de implementar y probar los algoritmos vistos anteriormente vamos a utilizar principalmente dos programas. El primero es el IDE de programación de Waspmote que podemos descargar desde la propia página Web de Libelium [31]. El segundo es el programa X-CTU que nos va a permitir monitorizar los sensores al permitir hacer una lectura de los puertos COM que es el que usan los sensores al estar conectados a un PC. Además, en caso de usar el protocolo ZigBee en vez del 802.15.4 nos permitiría configurar ciertos parámetros del mismo. Este programa lo podemos descargar de la página Web de Digi International [33].

3.1.1.- IDE Waspmote

El IDE de programación de Waspmote es un software basado en Java que nos va a permitir crear y compilar programas para nuestros Waspmote, así como subir los programas a los sensores y monitorizar el puerto COM seleccionado. Actualmente, en la página Web de Libelium podemos encontrar dos versiones del IDE de programación, la versión normal que nos permite desarrollar programas para Waspmote v1.1 y la versión PRO que nos permite desarrollar para los Waspmote v1.2 o PRO. En el siguiente enlace se puede descargar la versión normal para Windows, que es la que hemos utilizado en este trabajo [37]

He comprobado que el IDE de programación funciona en Windows XP y Windows 7 de 64 bits. Junto con el IDE de programación, en el archivo comprimido que nos descargamos viene también una carpeta llamada "Drivers". En esta carpeta están los drivers para Waspmote tanto para arquitecturas AMD64 como i368 de forma que si al conectar un sensor a nuestro PC no instala el driver automáticamente, simplemente le tenemos que decir que el driver se

encuentra en esta carpeta. He podido comprobar que en Windows XP es necesario indicarle el driver mientras que en Windows 7 lo instala automáticamente.

Lo primero que nos vamos a encontrar al abrir el programa es una pantalla similar a la de la Figura 24 que, como podemos ver, está en inglés.

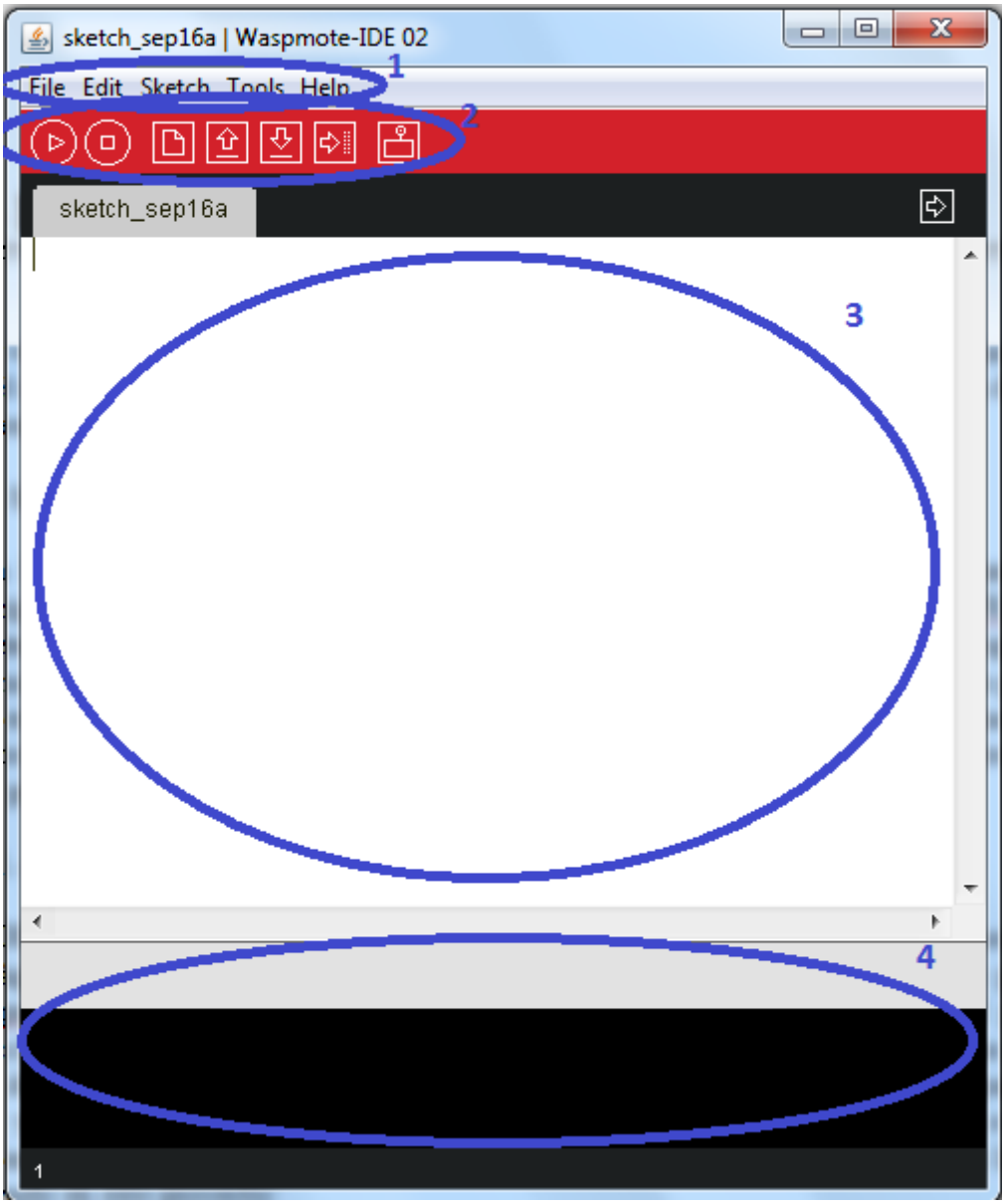


Figura 24. IDE de programación Waspmote

Como podemos ver, el programa consta de 4 partes bien diferenciadas:

1. Barra de menús
2. Acceso rápido a algunas funciones
3. Área de escritura de programas
4. Barra de notificaciones

En la barra de menús podemos encontrar las opciones que nos van a permitir manejar el programa. Dispone de 5 menús como son *File*, *Edit*, *Sketch*, *Tools*, y *Help*. En el menú *File* vamos a encontrar las opciones para abrir, guardar o crear nuevos documentos, imprimirlos, etc. También vamos a encontrar una serie de programas de ejemplo sobre las diferentes partes de la

programación de Waspmote como son el uso de los módulos 802.15.4 o ZigBee, el acelerómetro, el uso del RTC que nos va a permitir dormir y despertar al sensor, el uso de la tarjeta microSD, etc. Esto nos va a permitir iniciarnos en la programación de Waspmote de forma fácil. En el menú *File* vamos a encontrar las opciones de manejo del programa abierto como son copiar, pegar, buscar, etc. En el menú *Sketch* podemos encontrar las opciones de compilar, abrir la carpeta donde se encuentra el programa que estamos tratando, etc. En el menú *Tools* encontramos opciones que nos permiten seleccionar la versión de la API que queremos usar, el sensor sobre el que queremos trabajar o el panel de monitorización del puerto COM. Por último, en el menú *Help* podemos encontrar información y ayuda sobre el IDE y la programación en Waspmote.

En la barra de accesos rápidos podemos encontrar accesos para, de izquierda a derecha, compilar un programa, parar la compilación, crear un nuevo programa, abrir uno existente, guardar el actual, subir el programa al sensor y abrir el panel de monitorización del puerto COM. Hay que indicar que cada vez que subamos el programa a un sensor, primeramente el programa se compile aunque ya lo hayamos hecho previamente y no hayamos realizado ningún cambio. Esto puede ser un inconveniente ya que el proceso de compilación puede llevar entre 30 segundos y 1 minuto más o menos. Además, una vez que hemos iniciado la compilación o subida de un programa a un sensor no es posible parar el proceso.

En el área de escritura será donde realicemos los cambios sobre el programa que tenemos actualmente abierto. El programa detecta las funciones que tiene incorporadas y las marca en un color amarillo aunque no tiene una inteligencia que nos vaya mostrando las funciones de una clase o los campos como si hacen otros IDEs de programación como NetBeans. A la hora de escribir un programa, todo programa Waspmote consta de tres partes que son la declaración de variables, una función *setup* que sólo se inicia una vez cuando arranca el sensor y la función *loop* que se está ejecutando constantemente. Además, podemos declarar las funciones que consideremos oportunas. En los programas Waspmote no se pueden crear ni destruir hilos, sólo hay un hilo de ejecución. Esto puede ser un problema ya que, como hemos visto anteriormente, en los algoritmos en cluster hay que ejecutar varias tareas al mismo tiempo. En el apartado de implementación de los algoritmos veremos cómo solucionar este problema. Para crear funciones, se hace de forma similar al lenguaje C. Se crea una función escribiendo "tipoDatoDevolver nombreFuncion(tipoParametro1 nombreParametro1, tipoParametro2 nombreParametro2, …)". Para salir utilizaremos la sentencia "return datoDevolver". Para llamar a la función simplemente escribiremos su nombre y entre paréntesis, separados por ";" añadiremos los parámetros que queramos pasar a la función.

Por último, nos encontramos con la barra de notificaciones donde, como su nombre indica, nos aparecerán los mensajes de notificación como que un programa ya ha acabado de subirse al sensor, los errores producidos en la compilación, los errores de sincronización a la hora de subir un programa al sensor, etc.

El primer paso a la hora de empezar a programar con el IDE es seleccionar la versión de la API que vamos a utilizar. En caso de no indicarle ninguna versión, sólo es necesario la primera vez porque el programa guarda la configuración automáticamente, cuando intentemos compilar un programa nos dará errores y no lo compilará. Para seleccionar la versión de la API tenemos que ir a *Tools* → *Board* y en el desplegable nos aparecerán las diferentes versiones que tenemos instaladas como se puede ver en la Figura 25.

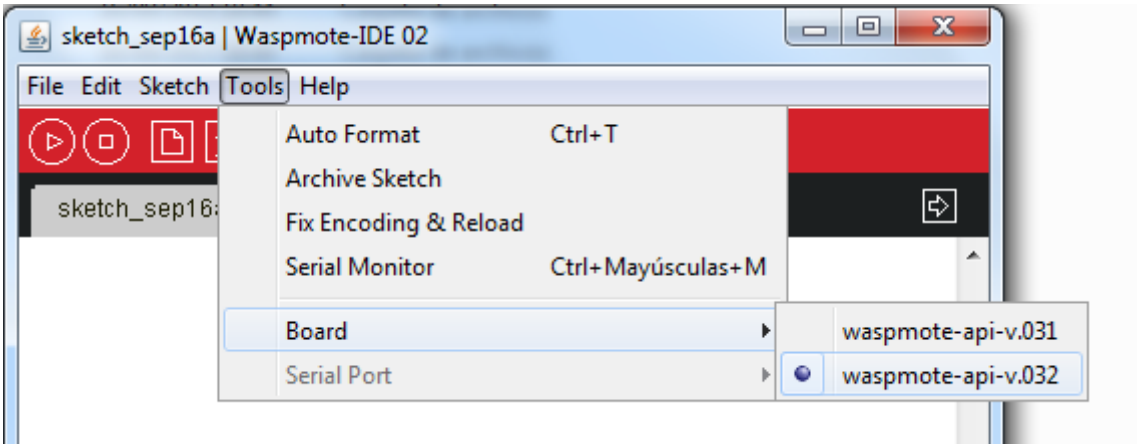


Figura 25. Seleccionar versión de API

Aunque en un principio no es necesario ya que el IDE viene con la versión 31 de la API y actualmente sólo hay hasta la 32 que no presenta muchos cambios, es posible añadir nuevas versiones de la API al IDE. Para ello sólo tenemos que descargarnos la nueva versión de la Web de Libelium, descomprimirla en la carpeta "hardware\cores" dentro de la carpeta del IDE y modificar el fichero "hardware\boards.txt". En un principio este fichero sólo incluye la versión 31 de la API. Habrá que copiar el texto que hay y modificarlo por cada versión que queramos incluir cambiando el número de la API, el nombre y poniendo la versión de la API adecuada. En el Código 1 podemos ver un fichero boards.txt modificado para añadir la versión 32 de la API. El texto original es el situado hasta las primeras almohadillas mientras que el resto es la modificación realizada para que detecte la versión 32 de la API.

```
##############################################################

wasp01.name=waspmote-api-v.031

wasp01.upload.protocol=stk500
wasp01.upload.maximum_size=122880
wasp01.upload.speed=38400

wasp01.bootloader.low_fuses=0xdf
wasp01.bootloader.high_fuses=0xd0
wasp01.bootloader.extended_fuses=0xf5
wasp01.bootloader.path=wasp_v8.1
wasp01.bootloader.file=ATmegaBOOT_v0003.hex
wasp01.bootloader.unlock_bits=0x3F
wasp01.bootloader.lock_bits=0x0F

wasp01.build.mcu=atmega1281
wasp01.build.f_cpu=8000000L
wasp01.build.core=waspmote-api-v.031


##############################################################

wasp02.name=waspmote-api-v.032

wasp02.upload.protocol=stk500
wasp02.upload.maximum_size=122880
wasp02.upload.speed=38400

wasp02.bootloader.low_fuses=0xdf
wasp02.bootloader.high_fuses=0xd0
wasp02.bootloader.extended_fuses=0xf5
wasp02.bootloader.path=wasp_v8.1
wasp02.bootloader.file=ATmegaBOOT_v0003.hex
wasp02.bootloader.unlock_bits=0x3F
wasp02.bootloader.lock_bits=0x0F
```

```
wasp02.build.mcu=atmega1281
wasp02.build.f_cpu=8000000L
wasp02.build.core=waspmote-api-v.032
```

Código 1. Fichero boards.txt para detectar las versiones 31 y 32 de la API

Una vez que hemos indicado la versión de la API ya podemos empezar a implementar programas en Waspmote. A la hora de subir el programa, tenemos que seleccionar el sensor al que lo queremos subir. Para ello nos dirigimos a *Tools* → *Serial Port*. Aquí nos aparecerán los sensores que tenemos conectados. Los sensores aparecerán nombrados como COMxx. La primera vez que se conecta un sensor a un PC éste le asigna un número progresivamente y lo mantiene aunque apaguemos el equipo. No hay manera de saber cuál es el número asociado a cada sensor más que, por ejemplo, observando los sensores COM que hay, desconectando uno y viendo cuál es el COM que ha desaparecido. Si lo volvemos a conectar veremos que vuelve a aparecer el COM anterior en la lista. De esta forma podemos identificar el número de COM asignado a un sensor en un equipo aunque este número cambiará cuando lo conectemos en otro equipo. En la Figura 26 podemos ver una lista de sensores conectados a un ordenador.

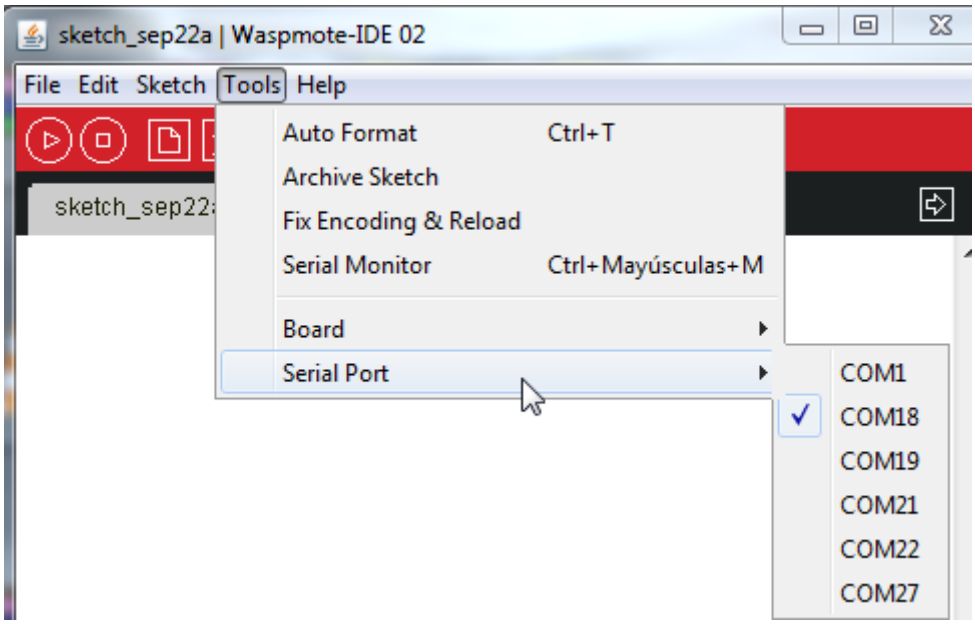


Figura 26. Lista de sensores conectados a un ordenador

Una vez seleccionado el sensor podemos subir el programa que tengamos abierto bien por medio del botón de acceso rápido, con la combinación CTRL+U o por medio del menú *File*. A la hora de subir un programa a un sensor hay que tener en cuenta las siguientes consideraciones para que la subida no de errores:

- El Jumper de programación tiene que estar puesto. En la Figura 27 podemos observar en la parte superior izquierda cuál es ese Jumper.
- El sensor tiene que estar encendido por medio del switch de encendido/apagado.
- El módulo de comunicaciones debe estar quitado del sensor.

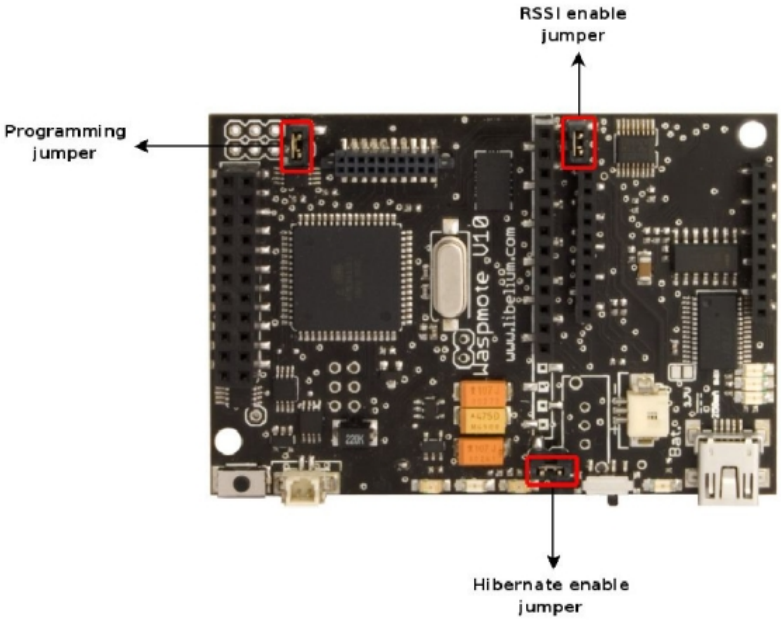


Figura 27. Jumpers del Waspmote

En caso de no cumplirse alguno de estos requisitos, en la barra de notificaciones aparecerá un error de sincronización y el programa no se subirá al sensor. El error que nos dará lo podemos ver en la Figura 28.

Figura 28. Error al subir un programa a un sensor.

Una vez realizado estos pasos, ya podemos subir los programas al sensor. Si la compilación y subida es correcta, en la barra de notificaciones aparecerá el mensaje "Done uploading" y el tamaño del programa subido como se puede observar en la Figura 24 con el texto "*Binary skecth size*".

Una vez cargado el programa podemos apagar el sensor, conectar el módulo de comunicaciones en caso de ser necesario y volver a encender el sensor para ejecutar el programa cargado. En caso necesario, el IDE dispone de un monitor que nos permite ver todo aquello que mande el sensor por medio del puerto USB. En general, esto será aquello que queramos imprimir por medio de la clase USB o XBEE aunque se recomienda usar la primera al ser éste su uso específico. Para poder monitorizar un determinado sensor primero lo tenemos que seleccionar de la misma forma que hacemos para cargar un programa. El IDE sólo nos permite monitorizar un sensor cada vez por lo que deberemos buscar otro programa, que veremos a continuación, en caso de ser necesario monitorizar varios sensores al mismo tiempo. Una vez seleccionado el sensor que queremos monitorizar, abriremos el monitor por medio del botón de acceso rápido, por medio de la combinación CTRL+Mayúsculas+M o por medio de la opción del menú *Tools*. De esta forma nos aparecerá una pantalla similar a la de la Figura 29 donde irá apareciendo todo aquello que se imprima por el puerto USB. Para que se muestre el texto

correctamente, los baudios los tenemos que dejar en 38400 que es cómo funcionan los Waspmote. En caso de no hacerlo, en vez del texto enviado por el puerto USB veremos símbolos ilegibles. Además, la batería debe estar quitada ya que si no he comprobado que no se muestra nada ni por medio de este monitor ni por medio del programa que veremos luego.

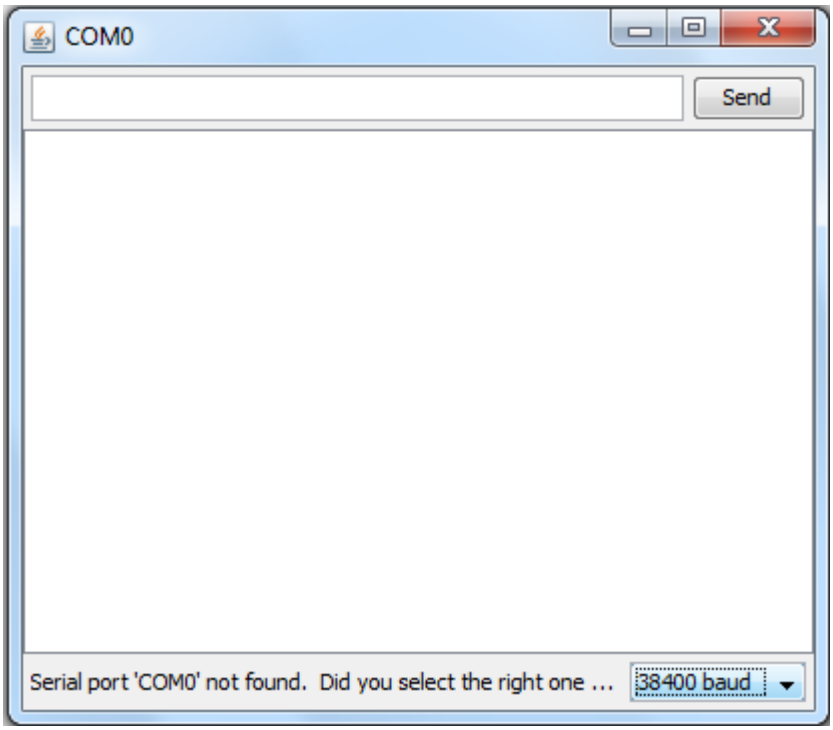


Figura 29. Monitor

El IDE de programación no dispone de opciones de depuración de programas por lo que si nuestro programa falla, la única manera que vamos a tener de encontrar el fallo es por medio de impresiones de texto para ver en qué punto el programa no hace lo que nosotros esperamos. Esto hace que la tarea de depuración de programas sea muy costosa en tiempo ya que cada vez que queremos modificar algún error encontrado o algo que no funcione como debería, debemos cambiarlo en el programa, compilarlo y subirlo al sensor, lo que para los programas implementados en este trabajo cuesta alrededor de un minuto y medio, aproximadamente dos entre quitar y volver a poner el módulo de comunicaciones, y volver a ejecutarlo para ver donde falla. Esto, unido a que hay muy poca información fuera de la página de Waspmote sobre cómo programar para Waspmote, hace que cada problema encontrado cueste mucho tiempo solucionarlo.

3.1.2.- X-CTU

X-CTU es un software de la empresa Digi International [33] que nos permite monitorizar y operar sobre los módulos de comunicaciones 802.15.4, y ZigBee en caso de usarlo, desarrollados por la empresa Digi que vamos a utilizar en nuestro trabajo. En nuestro caso nos va a permitir monitorizar los puertos COM por donde los sensores nos van a enviar la información. La diferencias con el monitor proporcionado por el IDE de Waspmote es que en este caso vamos a poder tener abiertas varias instancias de X-CTU al mismo tiempo lo que nos va a permitir monitorizar varios sensores al mismo tiempo. En caso de utilizar módulos ZigBee, X-CTU también nos permitiría configurar los módulos para indicarles el identificador de red sobre el que van a trabajar, si van a actuar como coordinadores de la red o no, etc.

Lo primero que vamos a ver al arrancar el programa es una pantalla similar a la de la Figura 30.

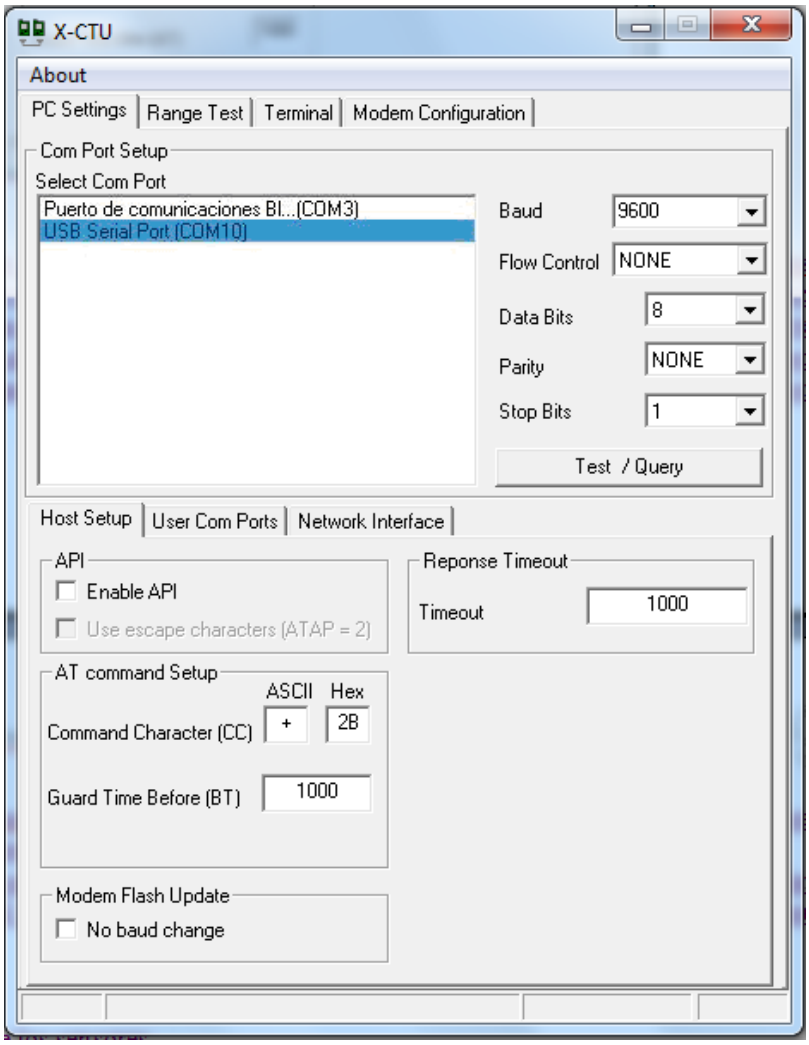


Figura 30. Pantalla inicial de X-CTU

En ella nos aparece una lista de los puertos que permite monitorizar. Los sensores Waspmote serán los que vengan marcados como USB Serial Port y el número COM coincidirá con el del IDE. Una vez arrancado X-CTU la lista de puertos no se actualiza aunque conectemos o desconectemos algún sensor por lo que si queremos trabajar sobre un sensor conectado después de arrancar X-CTU lo tendremos que reiniciar o abrir otra instancia. A la hora de monitorizar un sensor nos tenemos que asegurar de que sus baudios están puestos a 38400 para que el texto que envíe por el puerto USB lo podamos leer correctamente, como en el IDE.

En caso de querer monitorizar un sensor, una vez seleccionado en la lista nos dirigimos a la pestaña *Terminal* y nos encontraremos con algo similar a lo que podemos ver en la Figura 31.

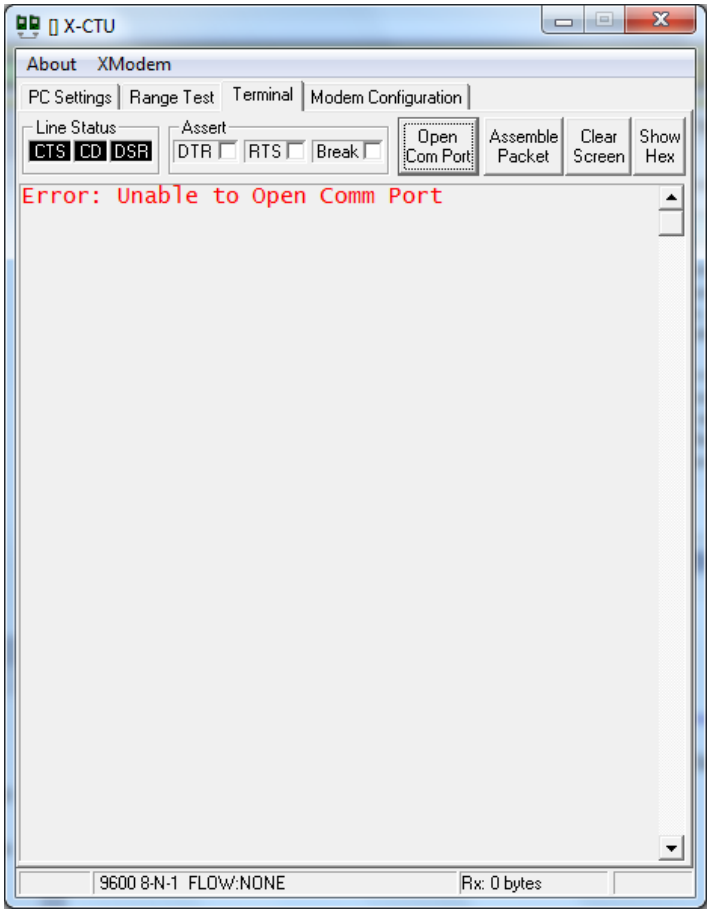


Figura 31. Pantalla de *Terminal*

En el recuadro inferior irá apareciendo el texto que el sensor imprima por el puerto USB. En caso de que el sensor seleccionado esté apagado, o lo hayamos desconectado, en la pantalla se mostrará un mensaje como el de la Figura 31. Si estamos trabajando al mismo tiempo con el IDE de programación y con X-CTU, a la hora de subir un programa al sensor tenemos que cerrar X-CTU o ir a alguna pestaña que no sea la de *Terminal*. Si intentamos subir un programa a un sensor mientras que lo estamos monitorizando con la pestaña *Terminal* de X-CTU nos aparecerá un mensaje similar al de la Figura 28 indicando que se ha producido un error de sincronización y que no se ha podido subir el programa al sensor. Esto es debido a que no podemos usar el puerto USB como lectura y escritura al mismo tiempo. Además, en caso de tener la batería conectada al sensor X-CTU no muestra nada. Esto creo que es debido a que utiliza el puerto USB para cargar la batería y no puede al mismo tiempo enviar datos.

En nuestro trabajo no vamos a utilizar X-CTU más que para monitorizar los sensores. En caso de usar módulos ZigBee, también lo podríamos usar para configurarlos. Para ello, si seleccionamos el sensor y vamos a la pestaña *Modem Configuration* nos encontraremos con algo similar a la Figura 32.

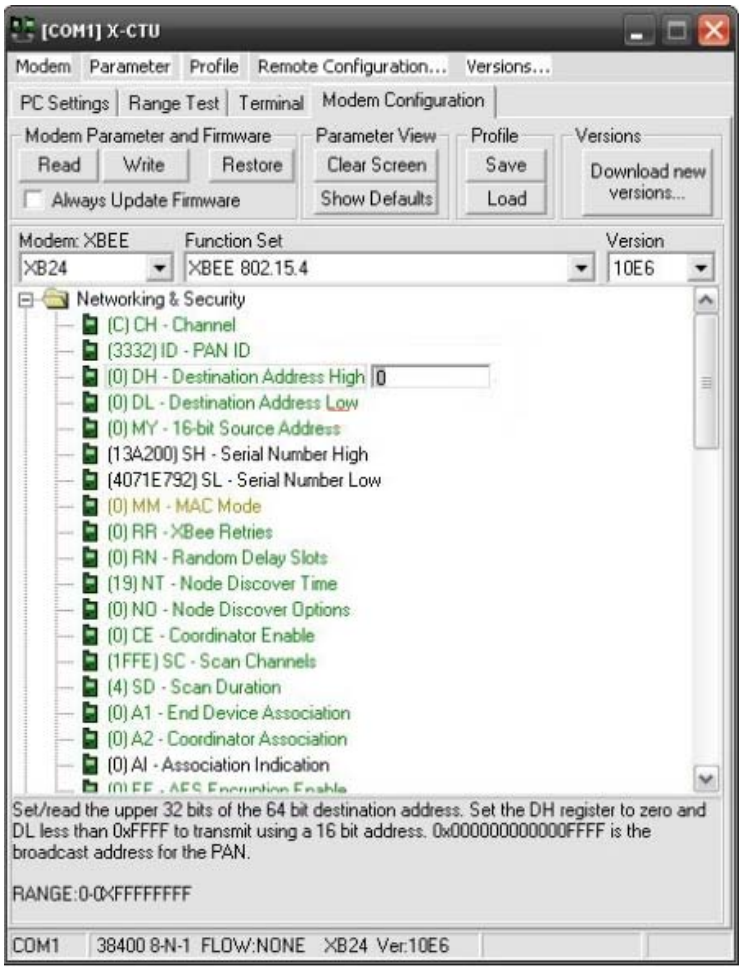


Figura 32. Pantalla de configuración de los módulos

En esta pantalla, tras pulsar el botón *read*, X-CTU nos muestra la actual configuración del sensor. Una vez leída la configuración la podemos modificar cambiando los parámetros que consideremos oportunos de los que se nos muestran. Además, con el desplegable *Funcion Set* podemos cambiar la funcionalidad del módulo. Normalmente nos serviría para decirle si queremos que actúe como coordinador o no.

3.2.- Implementación

Una vez que hemos visto el software que vamos a utilizar para implementar y probar los sensores vamos ver las consideraciones que tenemos que tener en cuenta y el desarrollo de los algoritmos.

3.2.1.- Consideraciones

Sólo dispongo de 10 sensores. Uno lo voy a usar como estación base para recoger los datos por lo que sólo dispongo de 9 sensores para formar la red de sensores. Como veremos más adelante, uno de los algoritmos que voy a implementar es un algoritmo en cadena donde cada sensor envía sus datos al siguiente.

Con esto, y teniendo en cuenta el despliegue de red que hemos indicado anteriormente que vamos a realizar, se formaría una malla de tamaño 3x3 con cada sensor separado aproximadamente 5 metros de sus vecinos. Esto haría que, en el caso de los sensores de los algoritmos en cluster que pueden necesitar enviar entre cualquier par de nodos, los sensores de las esquinas estarían situados como mucho unos 15 metros, distancia bastante inferior al rango máximo de los módulos de comunicación.

Como ya hemos visto, en nuestro trabajo vamos a utilizar tanto algoritmos que trabajan con topologías en cluster como en cadena. Los algoritmos en cluster, como veremos luego, nos obligan a que cada nodo tenga un identificador único ya que, en caso de empatar en el método de elección de líder, van a utilizar este identificador para elegir al líder. Es por eso que, manualmente, damos una numeración del 2 al 10 a cada uno de los sensores y los colocaríamos tal y como se ven en la Figura 9. El número 1 correspondería a la estación base a donde el líder de cada iteración enviaría los datos para tenerlos centralizados y poder trabajar con ellos en caso de ser necesario. Esta numeración va a ser estática durante toda la ejecución del algoritmo.

Por otra parte, en el algoritmo en cadena, al no tener un método de construcción automático, hay que construir la cadena manualmente. Por simplicidad mantenemos la misma numeración de sensores que en los algoritmos en cluster. Aunque en este caso no es necesario tener identificados de forma única, el que cada nodo tenga un número nos va a permitir trabajar de forma interna en el algoritmo con este identificador en vez de con la MAC del sensor, lo que nos va a facilitar el trabajo al ser un identificador más corto. Como veremos, cada nodo puede tener un vecino superior, inferior, a la izquierda y a la derecha. Por simplicidad, colocamos los sensores en una malla 3x3 y elegimos como vecinos los que se pueden ver en la Figura 13. Es decir, el sensor 2 tendría como vecino inferior al 3, éste como superior al 2 y como inferior al 4. El 4 tendría como vecino superior al 3 y como vecino a la derecha al 5. De esta forma, la cadena que se va a formar en un inicio es sensor 2, 3, 4, 5, 6, 7, 8, 9 y 10. Los identificadores no van a cambiar a lo largo de la ejecución del algoritmo pero la composición de la cadena sí que podría cambiar ya que, como veremos, en caso de que un sensor detecte que se va a quedar sin batería avisaría a los sensores vecinos para que reconfiguren sus vecinos y de esta forma la cadena. Por ejemplo, si el sensor 3 detecta que se va a quedar sin batería, avisaría al 2 que su vecino inferior pasa a ser el 4 y viceversa con el 4. De esta forma, la cadena se reconfiguraría con los sensores 2, 4, 5, 6, 7, 8, 9 y 10. En caso de producirse algún fallo permanente en la cadena, con el algoritmo que hemos implementado la cadena no se reconstruiría sino que se quedaría cortada produciendo que una parte de los datos no se recojan. Como hemos comentado anteriormente, en el Capítulo 5 plantearemos una posible solución a este problema.

En nuestros algoritmos vamos a usar el nivel de batería como dato a recoger ya que es uno de los datos a los que tenemos acceso de forma integrada en la placa de sensores. Los otros serían la temperatura o el valor del acelerómetro. El acelerómetro lo descartamos por razones obvias ya que, en un principio, no debería haber cambios significativos en estos valores. Podíamos usar la temperatura como dato a recoger pero el valor de la batería nos va a servir para comparar la eficiencia energética de los algoritmos al permitirnos comprobar cómo va disminuyendo su valor con el paso del tiempo.

3.2.2.- Implementación de DAWIN

El algoritmo DAWIN tiene como partes importantes la recepción del mensaje y comprobar de dónde se ha recibido, el añadir los datos propios del sensor al mensaje, la elección del siguiente nodo y envío del mensaje.

El primer paso es definir una política que nos permita diferenciar si los paquetes que recibe un nodo son de datos, de reconfiguración de cadena por nodo con baja batería u otro

paquete recibido por error. Para ello, usaremos los datos enviados dentro del paquete. A la hora de diferenciar los tipos de paquetes y que el nodo que los recibe sepa qué hacer con ellos, seguiremos el siguiente formato en los paquetes, "TipoPaquete#Direccion#IDPaquete#Datos". Es decir, el primer dígito indica el tipo de paquete donde un 1 indica que es un paquete de datos y un 2 es un paquete de reconfiguración de la cadena ya que el nodo que envía el paquete se va a quedar sin batería. Si el nodo es de reconfiguración, después del primer # vendrán los datos para que el nodo siguiente sepa cómo reconfigurar la cadena. Si el nodo es de datos, el siguiente dígito indica la dirección donde un 1 indica que vamos hacia delante y un 2 que vamos hacia atrás. Después del segundo # añadimos un contador del paquete, para poder controlar pérdidas, y luego irían los datos de cada nodo. Cada nodo añade a la trama los datos de la forma "IDNodo*Datos#". A continuación podemos ver una trama real recibida en la estación base "1#1#1#0002*43#0003*92#0004*86#0005*88#0006*86#0007*91#0008*94#0009*91#0010*44#". El primer 1 significa que es una trama de datos, el segundo 1 que hemos ido hacia delante, el tercer 1 significa que es el primer paquete y luego vendrían los datos de cada nodo, su ID y separado por * el valor de la batería. Una vez que hemos definido esto, vamos a empezar a ver cómo desarrollar el pseudocódigo para el algoritmo DAWIN visto en el apartado 2.2.2.1.

Lo primero que vamos a hacer en nuestro algoritmo es declarar una serie de variables globales tal y como se recomienda en la guía de programación de Libelium [38]. En el Código 2 podemos ver la declaración de estas variables. En estas variables guardamos los nodos vecinos que tiene cada nodo, donde usaremos un 1 para la estación base, 2 para el vecino superior, 3 para el inferior, 4 para el de la izquierda y 5 para el de la derecha. En caso de tener un 0 quiere decir que el nodo no tiene un vecino en esa dirección. También guardamos la MAC de los vecinos, la MAC de la estación base y el identificador del nodo en el que estamos. También añadimos un umbral que luego nos servirá para, si el valor de la batería es inferior a este valor, reconfigurar la cadena y apagar ese nodo antes de que la batería se acabe. Este umbral lo ponemos en un 10% de la batería ya que he comprobado al hacer pruebas que la lectura del valor de la batería no es muy precisa al comprobar que hay veces que da un valor y en la siguiente iteración del algoritmo da un valor mayor. Podríamos poner en las pruebas un valor menor a este, por ejemplo 5%, pero se podría dar que se apagase el sensor por baja sin antes detectarlo lo que causaría que no se reconfigurase la cadena. Además añadimos las MACs de todos los sensores ya que nos facilitará algunas tareas.

```
char* BS ="0013A2004071E415";
char* ID = "0002";
char* MAC = "0013A20040693815";
char* upMAC;
char* downMAC = "0013A2004071E44D";
char* leftMAC;
char* rightMAC;
int source=0;
int up=0;
int down=3;
int left=0;
int right=0;
int thres=10;
char* MAC1 = "0013A20040693815";
char* MAC2 = "0013A2004071E44D";
char* MAC3 = "0013A2004071E41B";
char* MAC4 = "0013A2004071E591";
char* MAC5 = "0013A2004071E442";
char* MAC6 = "0013A2004071E548";
char* MAC7 = "0013A200406FAD42";
char* MAC8 = "0013A2004071E592";
char* MAC9 = "0013A2004071E58F";
```

Código 2. Declaración de variables

Una vez declaradas las variables, en la función *setup* inicializamos los módulos que vamos a utilizar que en este caso sólo son el módulo 802.15.4 que nos va a permitir enviar y recibir paquetes y el módulo USB que nos va a permitir imprimir por el puerto USB. En el Código 3 podemos ver la función *setup*.

```
void setup(){
  // Inits the XBee 802.15.4 library
  xbee802.init(XBEE_802_15_4,FREQ2_4G,NORMAL);

  // Powers XBee
  xbee802.ON();

  USB.begin();
}
```

Código 3. Función *setup* de DAWIN

Una vez inicializado los módulos del sensor que vamos a utilizar, tenemos que hacer que se quede esperando paquetes. Cuando recibimos un paquete comprobamos por medio del ID del origen del paquete el emisor. Si el ID es igual a 0001 quiere decir que el paquete proviene de la base y que somos el primer nodo de la cadena. En este caso añadiremos nuestros datos al paquete y enviaremos un paquete hacia delante y otro hacía atrás. En el Código 4 podemos ver cómo concatenamos los datos que vamos a enviar. En este caso sería el paquete que vamos a enviar hacía adelante ya que el segundo dígito es un 1. Si estuviésemos creando el paquete para ir hacia atrás, en la primera instrucción sustituiríamos "1#1#" por "1#2#". En el caso de ser un nodo intermedio, a los datos recibidos del nodo anterior concatenaríamos el ID del nodo en el que estamos y el valor de la batería.

```
dataB = strcat(dataB, "1#1#");
//Concatemos el número de paquete en el que estamos
sprintf(auxA, "%d", packID);
dataB = strcat(dataB, auxA);
dataB = strcat(dataB, "#");
//Concatenamos el ID del nodo
dataB = strcat(dataB, ID);
dataB = strcat(dataB, "*");
//Concatenamos el valor de la batería
sprintf(auxA, "%d", PWR.getBatteryLevel()); //Leemos el valor de la batería
dataB = strcat(dataB, auxA);
dataB = strcat(dataB, "#");
```

Código 4. Creación de datos que vamos a enviar

Para saber cuál es el siguiente nodo al que hay que enviar, creamos dos funciones, *selectNextFwd* y *selectNextBwd*, al cual le pasamos el lado nodo del que hemos recibido el paquete siguiendo la regla que hemos definido antes, en este caso un 1 al ser la estación base, un 2 sería el nodo de arriba, etc. y a partir de ahí implementamos las Tablas 3 y 4 de decisión de siguiente nodo que hemos visto anteriormente. En el Código 5 podemos ver la función *selectNextFwd* que implementa la tabla de elección del siguiente nodo cuando vamos hacia adelante. La función *selectNextBwd* sería muy similar pero implementando la tabla de decisión cuando vamos hacia atrás.

```
int selectNextFwd(int source){
  //mensaje que viene de una estacion base
  if (source == 1){
    if (down == 3){
      return down;
    }
```

```
    if ((down == 0) && (right == 5)){
      return right;
    }
    if ((down == 0) && (right == 0)){
      return 1;
    }
  }
  //mensaje que viene de arriba
  if (source == 2){
    if (down == 3){
      return down;
    }
    if ((down == 0) && (right == 5)){
      return right;
    }
    if ((down == 0) && (right == 0)){
      return 1;
    }
  }
  //mensaje que viene de abajo
  if (source == 3){
    if (up == 2){
      return up;
    }
    if ((up == 0) && (right == 5)){
      return right;
    }
    if ((up == 0) && (right == 0)){
      return 1;
    }
  }
  //mensaje que viene de la izquierda
  if (source == 4){
    if (up == 2){
      return up;
    }
    if ((up == 0) && (down == 5)){
      return down;
    }
    if ((up == 0) && (down == 0)){
      return 1;
    }
  }
  //mensaje que viene de la derecha
  if (source == 5){
    return 1;
  }
}
```

Código 5. Implementación de la función *selectNextFwd*

Una vez que sabemos a qué nodo tenemos que enviar el paquete, pasamos este dato y los datos que vamos a enviar en el paquete a la función *sendAGM*. Esta función se encargará de crear el paquete que vamos a enviar y configurarlo con los datos de origen y destino. Luego enviará el paquete. Entre el envío del paquete hacía adelante y hacía atrás introducimos una espera para intentar evitar que los paquetes colisionen y se produzcan errores. En Código 6 podemos ver esta función. La función *direccionNodo* que aparece, simplemente devuelve la MAC asociada al nodo que hemos elegido como siguiente, el superior, inferior, izquierdo, derecho, o la estación base.

```
void sendAGM(char* data2, int next){

  //Preparamos paquete para envio
  paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
  paq_sent->mode=UNICAST;
  paq_sent->MY_known=0;
  paq_sent->packetID=packID;
  paq_sent->opt=0;
```

```
  nodo = direccionNodo(next);

  //Configuramos datos de envío y enviamos
  xbee802.hops=0;
  xbee802.setOriginParams(paq_sent, ID, MY_TYPE);
  xbee802.setDestinationParams(paq_sent, nodo, data2, MAC_TYPE, DATA_ABSOLUTE);
  xbee802.sendXBee(paq_sent);

//Comprobamos si el envío es correcto
if( !xbee802.error_TX )
  {
    XBee.println("ok");
  }
  else{
    XBee.println("error");
  }

//Liberamos el paquete enviado
  free(paq_sent);
  paq_sent=NULL;
  return;
}
```

Código 6. Función de envío de paquetes

Una vez que se han enviado los paquetes hacía delante y hacía atrás, el algoritmo vuelve a quedarse esperando a recibir un paquete aunque antes de eso tenemos que liberar el paquete recibido al principio ya que si no, al volver a comprobar si hay algún paquete disponible, volveríamos a leer el mismo paquete. Los paquetes recibidos se van almacenando en una tabla. Lo que hay que hacer es liberar el paquete, liberar esa posición de la tabla de paquetes recibidos y reducir el contador de paquetes tratados. En Código 7 podemos ver las instrucciones necesarias para liberar el paquete ya tratado.

```
free(xbee802.packet_finished[xbee802.pos-1]);
xbee802.packet_finished[xbee802.pos-1]=NULL;
xbee802.pos--;
```

Código 7. Liberación de los paquetes ya tratados

Si el paquete recibido no proviene de la estación base, sino que proviene de cualquier otro nodo, comprobamos a partir de los datos si es un paquete de datos o de reconfiguración de la cadena. En el Código 8 podemos ver como por medio de la función "*strExplode*" troceamos los datos usando de limitador el carácter "#". Nos quedamos con el primer trozo y si es un 1 sabemos que es una trama de datos, si es un 2 de reconfiguración de la cadena y cualquier otra cosa un error.

```
Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
if (atoi(Utils.arguments[0]) == 1){
   //Paquete de datos
}else if (atoi(Utils.arguments[0]) == 2){
   //Paquete de reconfiguración
}else{
   //Paquete erroneo
}
```

Código 8. Comprobación del tipo de paquete

Si el paquete es de datos, a partir del ID del nodo que ha enviado el paquete comprobamos la MAC que corresponde con ese identificador de nodo y se la pasamos a la función *rcvAGGREGATE* junto con los datos recibidos del nodo anterior. Esta función se

encargará, a grandes rasgos, de comprobar de donde viene el paquete, adjuntar los datos del propio nodo y seleccionar el siguiente nodo y enviar el paquete. En Código 9 podemos ver la implementación de esta función.

El primer paso para ello es saber si vamos hacia adelante o hacía atrás. Para ello seguimos un patrón similar al que hemos usado para saber si es un paquete de datos o de reconfiguración pero en este caso leemos el segundo dígito del paquete recibido. Si es un 1 vamos hacia adelante y si es un 2 vamos hacia atrás. Una vez que sabemos la dirección, lo siguiente que tenemos que saber es de dónde ha venido el paquete. Para ello comprobamos la MAC de origen con la MAC de los vecinos del sensor que hemos almacenado en las variables globales al inicio. Con estos datos llamamos a la función *selectNextFwd* o *selectNextBwd* y el vecino del que ha venido el paquete para poder decidir el siguiente nodo al que vamos a enviar el paquete.

Una vez que sabemos el siguiente nodo al que vamos a enviar el mensaje, añadimos los datos propios del nodo en el que estamos a los datos que hemos recibido del nodo anterior. Pasamos estos datos y el del siguiente nodo a la función *sendAGM* para realizar el envío de los datos.

Por último, comprobamos que el nivel de la batería no es inferior al umbral que hemos marcado en las variables globales. En caso de ser inferior, llamamos a la función que se encargará de enviar a los vecinos los datos necesarios para que realicen la reconfiguración de la cadena y apagaremos el sensor.

```
void rcvAGGREGATE(char* aux, char* data){

//Comprobamos si vamos Forward o Backward
  Utils.strExplode(data, '#');
  if (atoi(Utils.arguments[1]) == 1){
    //1 = Vamos Forward
    fwd = 1;
  }
  else{
    //2 = Vamos Backward
    fwd = 2;
  }

  //Elegimos siguiente nodo depende de donde haya venido el paquete anterior
  if (fwd == 1){
    if (aux == upMAC){
      next = selectNextFwd(2);
    }
    else if (aux == downMAC){
      next = selectNextFwd(3);
    }
    else if (aux == leftMAC){
      next = selectNextFwd(4);
    }
    else{
      next = selectNextFwd(5);
    }
  }
  else{
    if (aux == upMAC){
      next = selectNextBwd(2);
    }
    else if (aux == downMAC){
      next = selectNextBwd(3);
    }
    else if (aux == leftMAC){
      next = selectNextBwd(4);
    }
    else{
      next = selectNextBwd(5);
    }
  }
```

```
  //Concatemos los datos que vamos a enviar
  data = strcat(data, ID);
  data = strcat(data, "*");
  sprintf(aux, "%d", PWR.getBatteryLevel());
  data = strcat(data, aux); //De momento leemos el valor de la batería. Habrá que cambiarlo por leer de sensor
  data = strcat(data, "#");

  //Llamamos a la function de envío de paquetes
  if (next != 1){
    sendAGM(data, next);
  }
  else{
    sendAGM(data, 1);
  }

  //Reconfiguramos cadena si el nivel de bateria es muy bajo
  if (PWR.getBatteryLevel() < thres){
    reconfigureCHAIN();
    xbee802.OFF();
  }
}
```

Código 9. Función rcvAGGREGATE

En el algoritmo de DAWIN tiene que haber un nodo que inicie la ejecución en el resto de nodos enviando un mensaje al primer nodo. Este nodo será el que consideremos como estación base. La estación base simplemente tiene que enviar un mensaje a un nodo de la red, este nodo será el que consideremos inicializador de la cadena, y quedarse esperando a recibir los mensajes que le envíen los nodos al final de las dos sub-cadenas que se van a formar. La elección del nodo inicializador al que enviar el primer mensaje se podría hacer de forma aleatoria, variando el nodo siguiendo un orden, de forma fija, etc. Nosotros hemos elegido enviar siempre el mensaje de inicialización al mismo nodo para comprobar si el hecho de tener que enviar dos mensajes en vez de uno afecta al consumo de batería Una vez enviado el mensaje inicializador, que puede tener cualquier dato ya que lo que miramos para ver si es un mensaje inicializador es el origen del nodo que envía el mensaje, es decir, si viene de la estación base que hemos marcado con el identificador 1, la estación base se queda esperando mensajes. En general se queda esperando por dos mensajes, que procederían de las dos sub-cadenas y añadiríamos un *timeout* de forma que si pasado un tiempo que fijemos no ha recibido mensajes seguirá la ejecución. Una vez que ha recibido los dos mensajes, o ha recibido uno y ha saltado el *timeout* o ha saltado el *timeout* en los dos casos, se quedará esperando un tiempo antes de enviar el siguiente mensaje para iniciar la siguiente recogida de datos. Este tiempo lo iremos variando en las pruebas para ver cómo afecta en los resultados obtenidos.

3.2.3.- Implementación del primer algoritmo de cluster

Una vez que hemos visto cómo implementar el algoritmo DAWIN, vamos a ver cómo implementar el primer algoritmo de cluster cuyo pseudo-código vimos en la Figura 21. La implementación que vamos a ver da un error indeterminado, Waspmote no indica el tipo de error, al intentar enviar mensajes. No he sido capaz de determinar el porqué del error ni de solucionarlo por lo que el algoritmo no funciona aunque sí que el funcionamiento general creo que es el correcto ya que las diferentes partes del código utilizadas, como la lectura o escritura en la tarjeta SD, el uso de las alarmas, etc. se ha probado de manera individual y funciona correctamente.

Como hemos visto en la Figura 21, este algoritmo ejecuta varias tareas simultáneamente y, como hemos dicho anteriormente, Waspmote no permite la creación de hilos si no que sólo puede ejecutar una tarea al mismo tiempo por lo que ese es el primer problema que vamos a tener que solucionar.

Además, en este algoritmo vamos a tener que utilizar la tarjeta SD como almacenamiento permanente y el uso de la batería auxiliar para poder despertar el sensor una vez dormido.

Como variables globales, la única variable que tenemos que declarar es el número de nodo en el que estamos y lo almacenaremos en la variable *p*. Este número, también le llamaremos identificador del nodo, tiene que ser único dentro de la red ya que lo utilizaremos en la elección del líder para romper empates. Una vez que declaremos esa variable, podemos empezar a implementar la función de *setup*. A diferencia del algoritmo DAWIN, en este caso sí que vamos a realizar operaciones en la función *setup* además de inicializar los módulos necesarios. En concreto, vamos a inicializar los valores de las variables *Timeout*, *incTimeout* (ΔTimeout), *adquisitionTime*, *incarnation*, *leader* e *incarnationleader*. Estas variables corresponden con las que aparecen en el pseudo-código de la Figura 21. Aunque también vamos a utilizar la variable *eps* (Epsilon o ε), de momento no la inicializamos ya que la vamos a usar más adelante para saber si es la primera vez que pasamos por la función *loop* o no. La variable *incarnation* e *incarnationleader* la iniciamos a cero ya que en caso de almacenarlas será porque es la primera ejecución del algoritmo y por lo tanto no ha fallado ninguna vez y la variable *leader* al identificador del nodo en el que estamos. Una vez hecho esto, comprobaremos si el archivo *cluster.txt*, donde vamos a guardar las variables de este algoritmo en almacenamiento permanente, existe o no. Si este archivo no existe es porque es la primera vez que ejecutamos el algoritmo y no una recuperación del sensor tras un fallo. Esto nos va a obligar a borrar este archivo de la tarjeta SD cada vez que queramos lanzar el algoritmo pero es la única solución que se me ha ocurrido. Si el archivo no existe, almacenamos en él las variables *incarnation*, *leader*, *incarnationleader* y *Timeout*. Como veremos luego, estas variables las vamos a tener que leer de almacenamiento permanente más adelante en el algoritmo por lo que las almacenamos ya que, al estar recién creado el archivo, no se encuentran almacenadas y así evitar que el algoritmo falle cuando las tengamos que leer. En el Código 10 podemos ver la implementación que acabamos de comentar.

```
void setup(){ //Funcion de configuracion. Solo se ejecuta cuando arrancamos la placa de sensores
  // Inits the XBee 802.15.4 library
  xbee802.init(XBEE_802_15_4,FREQ2_4G,NORMAL);

  // Powers XBee
  xbee802.ON();

  // USB begin
  USB.begin();

  //SD ON
  SD.ON();
  SD.init();

  //RTC ON
  RTC.ON();

  //Inicializamos variables
  Timeout = 0;
  incTiemout = 2000;
  eps = 0;
  adquisitionTime = 10000;
  incarnation = 0;
  leader = p;
  incarnationleader = 0;

  if ((SD.isFile("cluster1.txt") == 0)){ //No esxiste el fichero
    SD.create("cluster1.txt");
    variables = "";
    Utils.long2array(incarnation, aux1);
    //sprintf(aux, "%d", incarnation);
    variables = strcat(variables, aux1);
```

```
    variables = strcat(variables, "#");
    delay(100);
    Utils.long2array(leader, aux1);
    //sprintf(aux, "%d", leader);
    variables = strcat(variables, aux1);
    variables = strcat(variables, "#");
    delay(100);
    Utils.long2array(incarnationleader, aux1);
    //sprintf(aux, "%d", incarnationleader);
    variables = strcat(variables, aux1);
    variables = strcat(variables, "#");
    delay(100);
    Utils.long2array(Timeout, aux1);
    //sprintf(aux, "%d", Timeout);
    variables = strcat(variables, aux1);
    variables = strcat(variables, "#");
    SD.writeSD("cluster1.txt", variables, 0);
  }

}
```

Código 10. Función *setup* del algoritmo cluster1

Además de los módulos 802 y USB que hemos utilizado en el algoritmo DAWIN, también vamos a utilizar, y por lo tanto inicializar, los módulos SD y RTC. El primero nos va a permitir usar la tarjeta SD y el segundo las funciones de alarmas lo que nos va a permitir despertar el sensor cuando lo hayamos dormido. Luego inicializamos las variables como hemos comentado, comprobamos si existe el fichero *cluster1.txt* y en caso de no existir lo creamos y almacenamos en él las variables que vamos a necesitar leer más adelante.

Para que el algoritmo funcione, los relojes de todos los nodos tienen que estar sincronizados y es lo primero que vamos a hacer en la función *loop*. Para sincronizar de forma fácil los relojes de todos los nodos, el primer nodo enviará un paquete en modo *broadcast* y el resto de nodos se quedará esperando por ese paquete. Cuando llegue, todos los nodos sincronizarán los relojes. Al estar los nodos relativamente cerca unos de otros, podemos considerar que el mensaje llega más o menos al mismo tiempo a todos los nodos. La desviación de tiempo que le puede costar al mensaje de sincronización llegar a los diferentes nodos quedaría compensada con el tiempo ε que pasa el líder esperando antes de empezar su funcionamiento y por el tiempo que se pasan los nodos esperando por paquetes *IAmAlive*. Esto es algo teórico y habría que comprobarlo al realizar pruebas.. En Código 11 podemos ver cómo se envían paquetes en modo *broadcast*. Para ello, en el modo de envío tenemos que poner BROADCAST en vez de UNICAST como vimos en al algoritmo DAWIN y en la MAC de destino hay que poner "000000000000FFFF. Aunque este método es el que se indica en la página de Libelium, he podido comprobar que si en el modo de envío ponemos BROADCAST el paquete también se envía a todos los nodos. Aunque esta segunda opción funciona, he decidido utilizar la primera al ser la recomendada por Libelium para evitar posibles comportamientos indeseados. Para saber si es la primera vez que entramos en la función *loop* usaremos la variable *eps*. Como hemos visto anteriormente, en la función *setup* le hemos dado el valor de 0 por lo que si continua con ese valor es la primera vez que ejecutamos *loop* ya sea la primera vez que ejecutamos el algoritmo o tras una recuperación tras un fallo. Además de sincronizar los relojes, también le daremos a *eps* el valor que consideremos adecuado para que la siguiente vez que ejecutemos *loop* no se vuelvan a sincronizar los relojes.

```
paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
paq_sent->mode=BROADCAST;
paq_sent->MY_known=0;
paq_sent->packetID=packID;
paq_sent->opt=0;
```

```
xbee802.hops=0;
xbee802.setOriginParams(paq_sent, ID, MY_TYPE);
xbee802.setDestinationParams(paq_sent, "000000000000FFFF", data, MAC_TYPE, DATA_ABSOLUTE);
xbee802.sendXBee(paq_sent);
```

Código 11. Envío de paquetes en modo BROADCAST

Cada vez que un nodo se despierta tras estar dormido, hay que ejecutar la función *Initialisation* vista en la Figura 21. Esta función la vamos a implementar dentro de la función *loop* y se ejecutará cuando el valor de *eps* sea diferente a 0 lo que indicará que ya se han sincronizado los relojes. En Código 12 podemos ver la implementación de esta función. Lo primero que hacemos es leer las variables *incarnation*, *leader*, *incarnationleader* y *Timeout* del almacenamiento permanente. Para hacer esto leemos los primeros 50 caracteres del fichero *cluster1.txt*, caracteres suficientes para leer todos los datos que tenemos que almacenar. En caso de que *scheduled_wakeup* valga cero quiere decir que no hemos pasado por la función *GoToHibernation* sino que el sensor se ha recuperado de un fallo por lo que aumentamos el valor de *incarnation* en 1 y lo volvemos a almacenar en el almacenamiento permanente. En vez de almacenar sólo la variable *incarnation* volvemos a almacenar todas por comodidad ya que es más sencillo hacer esto que rescribir sólo el lugar donde está almacenada la variable *incarnation*. Para hacer esto concatenamos todas las variables en una variable de tipo texto, convirtiendo los datos de tipo *long* a *string* en caso de ser necesario, y escribimos esa variable en el fichero *cluister1.txt* al principio del mismo para sobrescribir los valores que había antes. Al no poder ejecutar varias tareas al mismo tiempo, creamos las funciones *SoyLider* y *NoSoyLider* las cuales ejecutaremos dependiendo si el nodo en el que estamos es el líder o no. En la función *SoyLider* ejecutaremos las tareas 1 y 2 de la Figura 21 mientras que en la función *NoSoyLider* ejecutaremos las tareas 2 y 3. Más adelante veremos cómo simular la ejecución simultánea de dos tareas en una sola.

```
if (SD.isSD() == 1){ //Hemos pasado por Setup y esta la SD puesta
  aux = SD.cat("cluster1.txt", 0, 50);
  Utils.strExplode(aux, '#');
  incarnation = Utils.array2long(Utils.arguments[0]);
  leader  = Utils.array2long(Utils.arguments[1]);
  incarnationleader = Utils.array2long(Utils.arguments[2]);
 Timeout = Utils.array2long(Utils.arguments[3]);
}

if (scheduled_wakeup == 0){
  incarnation = incarnation + 1;
  //Vamos a escribir todas las variables en vez de sólo incarnation para seguir la coherencia
  variables = "";
  Utils.long2array(incarnation, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  delay(100);
  Utils.long2array(leader, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  delay(100);
  Utils.long2array(incarnationleader, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  delay(100);
  Utils.long2array(Timeout, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  SD.writeSD("cluster1.txt", variables, 0);
}

scheduled_wakeup = 0;

if (leader == p){
  SoyLider(eps, adquisitionTime, p, leader);
```

```
}
else{
  timer = Timeout + 2*eps;
  NoSoyLider(leader, timer, incTiemout);
}
```

Código 12. Implementación de la tarea *Initialisation* de la Figura 21

La función *SoyLider* se encargará de ejecutar las operaciones indicadas cuando el nodo en el que estamos es el líder. Lo que hay que hacer es enviar un mensaje *IAmAlive*. Este tipo de mensaje lo vamos a utilizar para indicar al resto de nodos que el nodo sigue funcionando correctamente e informar de su valor de *incarnation*, es decir, del número de veces que el nodo ha fallado y se ha recuperado. Además, hay que recoger los datos que envíen otros nodos y comprobar si se reciben paquetes *IAmAlive* de otros nodos con un *incarnation* menos al del nodo en el que estamos. En caso de ser así, cambiaremos el líder por el nodo que ha enviado el paquete. Lo que hacemos para poder simular la ejecución de dos tareas al mismo tiempo, al tener que esperar en dos tareas por mensajes recibidos, ya sean de datos o de mensajes *IAmAlive*, es ejecutar las operaciones hasta llegar a la recepción de mensajes las cuáles simplemente son esperar el tiempo marcado por *eps* y enviar el paquete IAmAlive, y luego nos quedamos esperando paquetes. De forma similar a como hicimos en DAWIN, si lo que enviamos es un paquete de datos, los datos empezarán por "1#" mientras que si es un paquete de tipo *IAmAlive* empezarán por "2#". De esta forma cuando la función *SoyLider* se quede esperando mensajes y reciba uno sabrá de qué tipo es. En caso de ser un paquete de datos mostraremos los datos por el puerto USB, aunque se podría hacer cualquier otra cosa como enviarlos a un nodo centralizado que haga de sumidero, mientras que si es de tipo *IAmAlive* extraerá el valor que adjunta de identificador de nodo y de *incarnation*. Si el valor de *incarnation* es menor al del propio nodo o es igual pero el identificador del nodo es menor o igual, en caso de ser igual el valor de *incarnation* quiere decir que es el mismo nodo que era líder en la ronda anterior y que no ha variado su *incarnation*, el nodo que ha enviado el mensaje se convertirá en el nuevo líder, o se mantendrá el líder anterior, y actualizaremos las variables *leader* e *incarnationleader* con los valores recibidos en el mensaje de identificador de nodo e *incarnation* respectivamente. Esperaremos por más mensajes mientras dure el tiempo marcado por *adquisitionTime* o hasta que el identificador del líder ya no sea el identificador del nodo en el que estamos, lo que querrá decir que el líder ha cambiado. Una vez que acabemos de esperar por más mensajes, por la razón que sea, llamaremos a la función *GoToHibernation* para dormir al nodo el tiempo que hayamos indicado. En Código 13 podemos ver la implementación que acabamos de describir.

```
void SoyLider(int eps, int adquisitionTime, int p, long leader){
  delay(eps);
  sendIAmAlive();
  previous=millis();
  while( ((millis()-previous) < adquisitionTime) && (leader == p))
  {
    if(XBee.available()){ //Paquete recibido
      xbee802.treatData();
      if(!xbee802.error_RX){ //Paquete recibido sin error
        while(xbee802.pos>0){
          Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
          if (atoi(Utils.arguments[0]) == 1){ //Paquete de datos
            for(int f=0;f<xbee802.packet_finished[xbee802.pos-1]->data_length;f++)
            {
              USB.print(xbee802.packet_finished[xbee802.pos-1]->data[f],BYTE);
            }
            free(xbee802.packet_finished[xbee802.pos-1]);
            xbee802.packet_finished[xbee802.pos-1]=NULL;
            xbee802.pos--;
          }
          else if (atoi(Utils.arguments[0]) == 2){ //Paquete de I-AM-ALIVE
```

```
          Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
          aux2 = Utils.array2long(Utils.arguments[2]);
          aux3 = Utils.array2long(Utils.arguments[1]);
          if ((aux2 < incarnationleader) || ((aux2 = incarnationleader) && (aux3 <= leader))){
            leader = aux3;
            incarnationleader = aux2;
            enviarDatos(leader);
            free(xbee802.packet_finished[xbee802.pos-1]);
            xbee802.packet_finished[xbee802.pos-1]=NULL;
            xbee802.pos--;
          }
        }
        else{
          free(xbee802.packet_finished[xbee802.pos-1]);
          xbee802.packet_finished[xbee802.pos-1]=NULL;
          xbee802.pos--;
        }
      }
    }
  }
 }

  GoToHibernation();
}
```

Código 13. Función *SoyLider*

La función *sendIAmAlive* simplemente crea un paquete con los datos "2#IDNodo#*incarnation*" y lo envía por *broadcast* a todos los nodos.

La función *NoSoyLider* se va a encargar de implementar las tareas 2 y 3 de la Figura 21 que son las se ejecutarán cuando el nodo en el que estamos no sea líder. Lo que hay que hacer es esperar por mensajes *IAmAlive* hasta que recibamos alguno cuyo *incarnation* sea menor al que tenemos almacenado del líder que actualmente tenemos asignado, o que sea igual pero con un identificador de nodo menor o igual. En caso de recibir un mensaje con estas características actualizamos el líder y sus datos y salimos. Si el actual nodo no ha aumentado su *incarnation* el paquete que envíe cumplirá las características al tener el mismo *incarnation* que antes y su identificador será igual por lo que volverá a ser elegido líder. Cuando actualicemos el líder, que puede ser el mismo como hemos visto, también le enviamos un paquete con los datos del nodo en el que estamos. Esto lo vamos a hacer durante un tiempo *timer*. Si pasado este tiempo no hemos recibido un paquete *IAmAlive* que cumpla las características, el propio nodo se auto-proclamará líder y actualizará los valores de *incarnationleader* y *leader* con sus propios datos. Una vez hecho esto, llamamos a la función *GoToHibernation* para dormir al nodo. En Código 14 podemos ver la implementación de la *NoSoyLider* descrita.

```
void NoSoyLider(long leader, long timer, long incTimeout){
  int lead = leader;
  USB.println("Funcion: No Soy Lider");
  USB.print("Lider: ");
  USB.println(leader);
  USB.print("timer: ");
  USB.println(timer);
  USB.print("lead: ");
  USB.println(lead);
  int a = 0;
  USB.println("Esperamos por paquetes IAmAlive");
  previous=millis();
  while( ((millis()-previous) < timer ) && (a == 0))
  {
    if( XBee.available() )
    {
      xbee802.treatData();
      if(!xbee802.error_RX){
        //Comprobamos que no hay nadie con un incarnation mejor que el mio
        Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
        aux2 = Utils.array2long(Utils.arguments[2]);
```

```
      aux3 = Utils.array2long(Utils.arguments[1]);
      if ((aux2 < incarnationleader) || ((aux2 = incarnationleader) && (aux3 <= leader))){
        USB.println("Vamos a caqmbiar de lider");
        leader = aux3;
        incarnationleader = aux2;
        USB.print("Lider: ");
        USB.println(leader);
        enviarDatos(leader);
        a = 1;
      }
    }
  }
 }

 //Pasa tiempo=timer sin encontrar un lider por lo que me hago lider
 if ( a == 1 ){
   USB.println("Pasamos el tiempo sin encontrar lider");
   USB.println("Me pongo como lider");
   leader = p;
   incarnationleader = incarnation;
   Timeout = Timeout + incTimeout;
 }

 GoToHibernation();
}
```

Código 14. Implementación de la función *NoSoyLider*

La función *enviarDatos* va a crear un paquete con los datos "1#IDNodo*valorBatería" y lo va a enviar al nodo líder. Para ello, le pasamos el ID del nodo a la función y ésta comprueba la MAC correspondiente a ese nodo y envía el mensaje. Esto lo hacemos así ya que en un principio no se van a agregar nodos a la red. En caso de que se pudiesen agregar nodos, en vez de tener ya fijadas las MACs de los nodos, en los mensajes *IAmAlive* además del identificador del nodo y el *incarnation*, enviaríamos también su MAC y la almacenaríamos en el nodo que reciba el mensaje si sale elegido líder.

La última función que nos queda por implementar es la función *GoToHibernation*. En Código 15 podemos ver su implementación. Esta función se va a encargar de guardar todas las variables en almacenamiento permanente de forma similar a como vimos en la función *loop*, fijar una alarma durante el tiempo que consideremos adecuado, en este caso 30 segundos, y dormir el sensor para que no consuma batería. La función de alarma puede funcionar por *Offset*, saltando cuando haya pasado el tiempo indicado, o saltando en una fecha y hora concretas. Hemos elegido la primera opción al ser más fácil de implementar ya que la segunda implicaría leer la fecha y hora en la que estamos y calculando la fecha y hora a la que queremos que se despierte el sensor para que haya pasado el tiempo que queremos que duerma. Pasado el tiempo que hemos indicado en la alarma, la batería auxiliar despertará al sensor y tendremos que volver a inicializar los módulos que hemos inicializado en la función *setup* ya que al dormir el sensor se desactivan para no consumir batería. Una vez hecho esto volvemos a lanzar la función *loop* para seguir con la siguiente iteración del algoritmo.

```
void GoToHibernation(){
  //Ecribimos las variables que necesitamos guardar en la SD
  variables = "";
  Utils.long2array(incarnation, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  delay(100);
  Utils.long2array(leader, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  delay(100);
  Utils.long2array(incarnationleader, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
```

```
  delay(100);
  Utils.long2array(Timeout, aux1);
  variables = strcat(variables, aux1);
  variables = strcat(variables, "#");
  SD.writeSD("cluster1.txt", variables, 0);

  //Ajustamos la alarma
  RTC.setAlarm1(0,0,0,30,RTC_OFFSET,RTC_ALM1_MODE2);

  scheduled_wakeup = 1;
  //Dormimos
  PWR.sleep(ALL_OFF);
  //Iniciamos los modulos despues de dormir
  xbee802.init(XBEE_802_15_4,FREQ2_4G,NORMAL);
  xbee802.ON();
  USB.begin();
  SD.ON();
  SD.init();
  RTC.ON();
  loop();
}
```

Código 15. Implementación de la función *GoToHinernation*

3.2.4.- Implementación del segundo algoritmo de cluster

La implementación del segundo algoritmo de cluster que hemos planteado anteriormente y cuyo pseudo-código vimos en la Figura 22, nos va a presentar dos problemas que tendremos que ver cómo solucionar. El primero de ellos es cómo enviar y recuperar en un mensaje una tabla. Hasta ahora habíamos enviado sólo variables o datos simples aunque veremos que la solución adoptada es muy similar a lo que hemos utilizado hasta ahora. El segundo problema es que el pseudo-código presenta la ejecución de tres tareas simultáneamente y como ya hemos dicho Waspmote no permite hacer esto. Tendremos que ver cómo simular que las tres tareas se ejecutan al mismo tiempo en una sola tarea. Al igual que el algoritmo primero de cluster, tampoco he logrado hacer que este algoritmo funcione por lo que vamos a ver la idea general de la implementación del mismo y no una implementación completamente funcional aunque sí que las diferentes partes, como el envío y recuperación de una tabla en un mensaje, se han probado de manera individual y funciona correctamente.

La única variable global que vamos a necesitar es el identificador del nodo, que al igual que en el algoritmo anterior, tiene que ser único en toda la red ya que lo vamos a utilizar para desempatar a la hora de realizar la elección del líder. También utilizamos una variable n, similar a η en la Figura 22 que es el tiempo que va a pasar entre envío de mensajes.

En la función *setup* de Waspmote implementaremos la tarea *Initialisation* vista en la Figura 22. Esta tarea se encarga de inicializar las variables cuando un sensor se recupera de un fallo. En este caso los sensores no se duermen por lo que sólo vamos a leer y escribir en almacenamiento permanente el valor *incarnation* del nodo que lleva cuenta del número de veces que el nodo ha fallado y se ha recuperado. En Código 16 vemos la implementación de la función *setup* del segundo algoritmo de cluster. Lo primero que hacemos es inicializar los módulos que vamos a utilizar. En este caso son los módulos 802, USB y SD. En este algoritmo no hacemos uso del módulo RTC al no dormir el sensor. Lo siguiente es leer el valor de *incarnation* almacenado en almacenamiento permanente. Para ello, primero comprobamos si existe en la tarjeta SD el fichero *cluster2.txt*. En este fichero vamos a almacenar los datos relativos a este algoritmo. En caso de no existir el fichero, podemos suponer que es la primera vez que se ejecuta el algoritmo y no que se ha recuperado de un fallo por lo que inicializamos la variable *incarnation* a cero. Al igual que en el algoritmo anterior, esto nos obliga a borrar el fichero *cluster2.txt* de la tarjeta SD cada vez que queramos volver a poner en ejecución el algoritmo

pero es la única solución, o la más sencilla, que se me ha ocurrido. En caso de existir el fichero podemos suponer que el sensor se ha recuperado de un fallo por lo que leemos el valor de *incarnation* almacenado en la tarjeta SD y aumentamos su valor en uno para representar el número de veces que el sensor se ha recuperado de un fallo. Tanto si es la primera vez que lanzamos el algoritmo como si se ha recuperado de un fallo, almacenamos el valor de *incarnation* en la memoria permanente. Una vez hecho esto, inicializamos las variables *timer*, *timerIni*, *Timeout* y *counter* que vamos a utilizar en el algoritmo. Las tres primeras nos servirán para calcular cuánto tiempo puede pasar entre dos envíos de mensaje de un mismo nodo antes de que consideremos que ha fallado. La última variable la utilizaremos para almacenar los valores de *incarnation* del resto de nodos de la red. Todas estas variables son tablas de enteros con tantas posiciones como nodos hay en la red. Por último, auto-denominamos al nodo como líder.

```
void setup(){ //Funcion de configuracion. Solo se ejecuta cuando arrancamos la placa de sensores
  // Inits the XBee 802.15.4 library
  xbee802.init(XBEE_802_15_4,FREQ2_4G,NORMAL);

  // Powers XBee
  xbee802.ON();

  //USB bein
  USB.begin();

  //SD ON
  SD.ON();
  SD.init();

  if ((SD.isFile("cluster2.txt") == 0)){ //No esxiste el fichero
    SD.create("cluster2.txt");
    incarnation = 0;
  }else{
    //Initialisation
    aux = SD.cat("cluster2.txt", 0, 50);
    Utils.strExplode(aux, '#');
    incarnation = Utils.array2long(Utils.arguments[0]);
    incarnation = incarnation + 1;
  }
  Utils.long2array(incarnation, aux);
  data = "";
  data = strcat(data, aux);
  data = strcat(data,"#");
  SD.writeSD("cluster2.txt", data, 0);
  //Anteriores instrucciones, similar a sumar 1 a incarnation en la tabla de almacenamiento y luego leer incarnation

  for (int i = 0; i < 10; i++){
    if (i != p){
      Timeout[i] = n + incarnation;
      resetTimer(timer, timerIni, i, Timeout[i]);
      counter[i] = 0;
    }
  }

  counter[p] = incarnation;
  leader = p;

}
```

Código 16. Función *setup* del segundo algoritmo de cluster

La función *resetTimer*, se encarga de dar a las variables *timer* y *timerIni* del nodo que le indiquemos los valores adecuados que luego nos permita comprobar si se ha superado el *timer* de algún nodo. Estos valores, como podemos ver en Código 17, son el valor del momento actual para *timerIni* y el valor de *Timeout* para *timer*.

```
void resetTimer(long timer[10], long timerIni[10], long i, long Timeout){
  timer[i] = Timeout;
  timerIni[i] = millis();
}
```

Código 17. Implementación de la función *resetTimer*

Una vez que hemos implementado la tarea *Initialisation* de la Figura 22, vamos a ver cómo poder implementar las tareas 1, 2 y 3 teniendo en cuenta que las tenemos que implementar en una sola tarea al no poder usar hilos. La implementación de estas tareas la vamos a hacer sobre la función *loop* de Waspmote, para que se esté siempre ejecutando, y en Código 18 podemos ver cómo hemos decidido implementarla. La tarea 1 implica cada η unidades de tiempo, en nuestro algoritmo usamos la variable n, enviar un mensaje *IAmAlive* en modo broadcast. Dentro de este mensaje se añaden los datos del nodo en el que estamos y la variable *counter* que contiene el valor almacenado en el nodo de *incarnation* de todos los nodos, incluido en el que estamos. De forma que con un solo mensaje decimos al resto de nodos que el nodo en concreto está vivo, informa del número de veces que sabe que cada nodo ha fallado y envía sus datos. Estos datos se podrían enviar por separado y enviar los datos sólo al nodo que sea líder pero de esta forma ahorramos el número de mensajes enviados y recibidos en cada nodo, lo que supone un ahorro de batería. Pasamos un tiempo n en un bucle haciendo las tareas 2 y 3, y pasado ese tiempo volvemos a enviar un mensaje *IAmAlive*. Para calcular el tiempo que pasa, al inicio del bucle que va a implementar las tareas 2 y 3 guardamos el momento en el que estamos por medio de la función *millis*. Cada vez que vuelve a empezar el bucle volvemos a calcular el momento en el que estamos y si este momento menos el valor almacenado de cuando empezó el bucle es mayor a n salimos del bucle para volver a enviar un mensaje *IAmAlive*. Esto hay que hacerlo continuamente para la tarea 1 que es justamente lo que hace la función *loop* por lo que no necesitamos implementar nada más. Para las tareas 2 y 3, una implica comprobar continuamente si se han recibido mensajes de otros nodos y la otra comprobar que no haya expirado ningún *timer* de ningún nodo. Aunque podríamos estar continuamente comprobando si se ha recibido un nuevo mensaje y a continuación si ha expirado algún *timer*, considero que lo que cuesta comprobar los *timers* de todos los nodos podría causar que mientras llegue un mensaje de algún nodo causando que se detecte un falso fallo. Es por esto por lo que sólo compruebo si se ha superado algún *timer* cuando hayamos recibido algún mensaje. Esto también puede causar que un nodo que había superado su *timer* no se detecte a tiempo pero considero que es menos grave este caso que el anterior. Habría que hacer pruebas con ambas implementaciones para ver cuál da mejores resultados.

La tarea 2 consiste en recibir mensajes del resto de nodos y comparar los valores de *counter* con los del propio nodo quedándonos con el mayor valor. Cada vez que recibimos un nuevo mensaje, hayamos actualizado o no algún valor de nuestra propia variable *counter*, actualizamos el líder. Si tras esto detectamos que el nodo en el que estamos es el líder recogemos los datos que nos han llegado con el mensaje y los tratamos. En este caso simplemente los mostramos por el puerto USB pero podríamos hacer lo que considerásemos oportuno como enviarlos a un sumidero. También reseteamos el *timer* del nodo que haya enviado el paquete. Como veremos más adelante, los datos de los mensajes van a tener la siguiente forma "A#IDNodo#counter[1]#counter[2]#...#counter[10]#valorBateria". Es por eso que, al trocear los datos con la función *Utils.strExplode*, en la tabla *Utils,arguments* donde se almacenan los trozos tenemos en la posición 1 el identificador del nodo que envía el mensaje, de la 2 a la 11 los valores de *counter* de ese nodo y en la posición 12 el valor de su batería.

La tarea 3 consiste en comprobar si se ha superado el *timer* de algún nodo. En caso de superarse, consideramos que el nodo ha fallado y aumentamos su *incarnation* lo que supone aumentar el valor de la posición correspondiente al nodo que ha fallado en la variable *counter*.

Para saber si se ha superado algún *timer*, por ejemplo del nodo i, tenemos que comprobar que no ha pasado más de un tiempo *Timeout[i]* para el nodo concreto que estamos comprobando. Como hemos visto anteriormente, cuando reseteamos un *timer* le damos a *timerIni[i]* el valor del momento en el que estamos y a *timer[i]* el valor de *Timeout[i]*. Como el tiempo que ha pasado tiene que ser menor a ese *Timeout[i]*, comprobamos que la suma de *timerIni[i]* más *timer[i]* sea menor el momento en el que estamos. En caso de ser mayor consideramos el nodo i ha fallado por lo que aumentamos su *counter[i]*, reseteamos el *timer* ahora dándole más margen ya que aumentamos *Timeout[i]* y comprobamos si el cambio en *counter[i]* supone un cambio en el líder.

```
void loop(){
  USB.println("Funcion: loop");
  //Task 1
  sendIAmAlive(counter, p);
  //Vamos a esperar a recibir un paquete n millis
  //Cuando se pase el tiempo volvemos a enviar I-AM-ALIVE
  //Similar a wait(n) de la Task1

  previous = millis();
  while((millis()-previous) < n )
  {
    if( XBee.available() ) //      (14) upon reception of message (I-AM-ALIVE, q, counterq, dataq) do
    {
      xbee802.treatData();
      if(!xbee802.error_RX){
        //Task 2
        //Comprobamos que no hay nadie con un incarnation mejor que el mio
        Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
        aux = Utils.arguments[1];
        long q = Utils.array2long(aux);
        resetTimer(timer, timerIni, q, Timeout[q]);
        long counterq[10];
        for (int i = 0; i < 10; i++){
          aux = Utils.arguments[2+i];
          counterq[i] = Utils.array2long(aux);
          delay(100);
        }

        for (int i = 0; i < 10; i++){
          if (counter[i] < counterq[i]){
            counter[i] = counterq[i];
          }
        }

        leader = updateLeader(counter);
        if (leader == p){
          data = Utils.arguments[12];
          for(int f=0;f<data->data_length;f++)
          {
            USB.print(data[f],BYTE);
          }
        }
        //Liberamos el paquete que hemos tratado
        free(xbee802.packet_finished[xbee802.pos-1]);
        xbee802.packet_finished[xbee802.pos-1]=NULL;
        xbee802.pos--;

        //Task 3
        for (int i = 0; i < 10; i++){
          if ((timer[i]+timerIni[i]) <= millis()){ //Similar a "upon expiration of timerp(q)"
            counter[i] = counter[i] + 1;
            Timeout[i] = Timeout[i] + 1;
            resetTimer(timer, timerIni, i, Timeout[i]);
            leader = updateLeader(counter);
          }
        }
      }
    }
  }
```

```
}
```

Código 18. Implementación de las tareas 1, 2 y 3 dentro de la función *setup*

Con la implementación de las funciones *setup* y *loop* habríamos implementado las funciones principales del algoritmo. Sólo nos quedaría ver la implementación de la función *updateLeader,* que realiza la elección del líder, y de la función *sendIAmAlive* que se encarga de enviar los mensajes *IAmAlive* con los datos necesarios.

La función *updateLeader* tiene que elegir como líder a aquel nodo que menos haya fallado, es decir, cuyo *incarnation* sea menor. Al tener almacenados en *counter* los valores de *incarnation* de todos los nodos, tenemos que recorrer esa tabla y quedarnos con el nodo cuya posición en la tabla tenga menor valor. En caso de haber un empate en el valor de *incarnation* nos quedamos con el nodo con un identificador menor aunque podríamos usar cualquier otro método de desempate como el del nodo con mayor identificador siempre y cuando usemos el mismo criterio en todos los nodos. En nuestra implementación, a los nodos les hemos dado los identificadores del 1 al 10 lo que hace que la posición *counter[j]* corresponda al nodo j. En caso de no ser así, crearíamos otra tabla donde en cada posición, por ejemplo la k, almacenaríamos el identificador del nodo cuyo *incarnation* sea *counter[k]*. En Código 19 podemos ver la implementación de *updateLeader*. Las variables *mini* y *mval* hay que iniciarlas a unos valores que nos aseguren que no hay ningún identificador de nodo y valor de *incarnation* respectivamente superior a esos valores. En el caso del identificador es relativamente sencillo al dar de antemano un valor a los identificadores de los nodos lo que nos permite saber el valor máximo. El segundo caso puede ser más complicado al aumentar los valores de *incarnation* continuamente, más bien nunca disminuyen, por lo que, si se ejecuta durante suficiente tiempo el algoritmo, este valor siempre se podría superar. De todos modos, consideramos que un valor de 999 es más que suficiente para las pruebas realizaríamos. Además, se puede implementar una pequeña función que compruebe el valor de *incarnation* del nodo en el que estamos y en caso de superar un valor prefijado apague el nodo al considerar que ese nodo es inestable al fallar y recuperarse muchas veces.

```
int updateLeader(long counter[10]){
  USB.println("Funcion: updateLeader");
  int mini = 999;
  int mval = 999;
  for (int i = 0; i < 10; i++){
    if(counter[i] < mval){
      mval = counter[i];
      mini = i;
    }
  }

  leader = mini;
  return leader;
}
```

Código 19. Función *update Leader*

Por último, la función *sendIAmAlive* es muy similar a la que vimos en el algoritmo primero de cluster. La diferencia es los datos que se envían en el paquete y en Código 20 podemos ver la implementación de la variable *data* que va a contener los datos que se van a enviar. Vemos como a la hora de concatenar los valores de *counter*, y a la hora de leerlos, introducimos un pequeño retraso. Hacemos esto ya que si no, aunque no es lógico, al ejecutar el algoritmo y llegar a ese punto, entra en un bucle infinito y es la única solución que he encontrado.

```
data = "A";
data = strcat(data, "#");
Utils.long2array(p, aux1);
data = strcat(data, aux1);
data = strcat(data, "#");
for (int i = 0; i < 10; i++){
  Utils.long2array(counter[i], aux1);
  delay(100);
  data = strcat(data, aux1);
  data = strcat(data, "#");
  delay(100);
}
sprintf(aux, "%d", PWR.getBatteryLevel());
data = strcat(data, aux);
```

Código 20. Concatenación de datos a enviar en *sendIAmAlive*

3.2.5.- Implementación del tercer algoritmo de cluster

Una vez que hemos visto la idea general de implementación de los dos primeros algoritmos de cluster, a continuación vamos a ver cómo implementar el tercer algoritmo de cluster cuyo pseudo-código pudimos ver en la Figura 23. Al igual que en los dos algoritmos de cluster anteriores, no se ha logrado que el algoritmo funcione por lo que se va a explicar la idea general de implementación aunque sí que se han probado por separado las novedades de programación utilizadas.

Como se puede observar en las Figuras 22 y 23, el pseudo-código del segundo y tercer algoritmo de cluster es muy similar por lo que su implementación también lo va a ser. Necesitamos crear una variable global para almacenar el identificador del nodo en el que estamos y otra llamada k donde almacenamos el tiempo que va a pasar entre envío de mensajes de *IAmAlive*.

La función *setup* que implementa la tarea *Inisitalisation* es prácticamente igual. Inicializamos los módulos 802, USB y SD que vamos a utilizar. Si el fichero donde vamos a almacenar de manera permanente las variables, en este caso *cluster3.txt*, no existe lo creamos e inicializamos la variable *incarnation* a 1. En este caso la inicializamos a 1 en vez de a 0 ya que luego, para saber si tenemos datos de un determinado nodo, comprobaremos que su *incarnation* sea distinto de 0. Si es distinto de 0 quiere decir que, aun en el caso de haber fallado, su *incarnation* valdrá 1 al ser el valor con el que inicializamos la variable al arrancar el algoritmo la primera vez. Guardamos el valor de *incarnation* en almacenamiento permanente, aumentando en 1 su valor en caso de existir ya el fichero *cluster3.txt*lo que indicaría que nos hemos recuperado tras un fallo, e inicializamos la variable *membership*, que es una tabla donde almacenaremos el *incarnation* de cada nodo, a 0, excepto en la posición que corresponde al nodo en el que estamos que la inicializamos al valor que tenga *incarnation*. En los pseudo-códigos, las variables *membership* y *counter* se diferencian en que la primera es una tabla de dos dimensiones donde almacenamos valores de *incarnation* y el nodo al que corresponde ese valor. En nuestro caso, al numerar los nodos del 1 al 10, no es necesario realizar esto ya que suponemos que la posición 1 de la tabla pertenece al nodo 1 y así sucesivamente por lo que la implementación y uso de esta variable es igual al de la variable *counter* del segundo algoritmo de cluster. No voy a mostrar el código al ser prácticamente igual al Código 16 donde hemos mostrado la implementación de la función *setup* del algoritmo dos pero introduciendo los cambios descritos. Aunque en el pseudo-código de la Figura 23 no pone que en la tarea *Initialisation* haya que resetear los *timers* de los nodos ya que los crea en la tarea 2, en nuestra implementación, al saber de antemano cuántos nodos vamos a tener, creamos todos los *timers* en la función *setup* pero le damos un valor de *Timeout* lo suficientemente alto como para que no expiren.

Al igual que en el algoritmo anterior, en la función *loop* vamos a implementar las tareas 1, 2 y 3 del algoritmo. La tarea 1 y la tarea 3 son exactamente iguales en ambos algoritmos por lo que su implementación la hemos realizado de igual manera haciendo las mismas asunciones con respecto a los *timers* y comprobándolos sólo cuando nos llega un nuevo mensaje y no continuamente. En la implementación de la tarea 1, el valor de la batería se lo pasamos a la función *sendIAmAlive* de fuera en vez de leerlo dentro de la función. En la implementación de la tarea 2 veremos el porqué.

La tarea 2 es ligeramente distinta ya que al recibir un mensaje, si es la primera vez que recibimos un mensaje de ese nodo, renviamos el mensaje tal y como lo hemos recibido en modo broadcast. En Código 21 podemos ver la implementación de la tarea 2 del tercer algoritmo. Los datos que vamos a enviar van a seguir el mismo formato que en algoritmo dos de cluster, es decir, "A#IDNodo#membership[1]#membership[2]#...#membership[10]#valorBateria". Cuando recibimos un mensaje, lo primero que hacemos es extraer el valor de identificador de nodo, *membership* y valor de batería. Si el identificador del nodo es, por ejemplo q, comprobamos si *membership[q]* vale 0. En caso de ser así quiere decir que nunca hemos recibido datos de ese nodo ya que tiene el valor con el que lo hemos inicializado. En este caso, vamos a renviar por *broadcast* un mensaje que contenga exactamente los mismos datos que los que hemos recibido. Para ello he modificado ligeramente la función *sendIAmAlive* para que, en vez de leer el valor de la batería dentro de la función le pasemos este valor desde fuera. De esta manera le podemos pasar el valor de la batería que hemos recibido en el mensaje y que corresponde al nodo que haya realizado el envío del mismo. Una vez hecho esto, comprobamos si hay alguna posición que valga 0 en *membership* y que no lo valga en el *membership* que hemos recibido. En caso de darse esta situación, quiere decir que hemos recibido en el mensaje el valor de *incarnation* de un nodo que aún no teníamos. En este caso, guardamos su valor e inicializamos su *timer* y *Timeout*. En caso de ya disponer de un valor para un determinado nodo, nos quedamos con el valor más alto de *incarnation* para ese nodo entre el que tenemos almacenado en *membership* y el que hemos recibido en el *membership* del mensaje. Una vez hecho esto reseteamos el *timer* del nodo q del que hemos recibido el mensaje, actualizamos el líder y, en caso de ser nosotros el líder, realizamos las operaciones necesarias con los datos recibidos que en este caso se limitan a mostrar los datos por el puerto USB.

```
//Task 2
Utils.strExplode(xbee802.packet_finished[xbee802.pos-1]->data, '#');
aux = Utils.arguments[1];
long q = Utils.array2long(aux);
long mem[10];
for (int i = 0; i < 10; i++){
  aux = Utils.arguments[2+i];
  mem[i] = Utils.array2long(aux);
  delay(100);
}

if (membership[q] == 0){
  aux = Utils.arguments[12];
  sendIAmAlive(q, mem, aux);
}

for (int i = 0; i < 10; i++){
  if ((membership[i] == 0) && (mem[i] != 0)){
    membership[i] = mem[i];
    Timeout[i] = k + incarnation;
    resetTimer(timer, timerIni, i, Timeout[i]);
  }else{
    if(mem[i] > membership[i]){
      membership[i] = mem[i];
    }
  }
}
```

```
resetTimer(timer, timerIni, q, Timeout[q]);
leader = updateLeader(membership);
if (leader == p){
  data = Utils.arguments[13];
  for(int f=0;f<data->data_length;f++)
  {
    USB.print(data[f],BYTE);
  }
}
```

Código 21. Implementación de la tarea 3 en el tercer algoritmo de cluster

El resto de funciones implementadas, *updateLeader*, *sendIAmAlive* y *resetTimer*, son iguales a las que hemos visto para el algoritmo dos de cluster con la diferencia ya indicada de que en la función *sendIAmAlive* el valor de la batería se lo pasamos de fuera en vez de leerlo en la función. Al ser el código muy similar, no lo vamos a mostrar si no que se puede ver en Código 19, 20 y 17 respectivamente del algoritmo anterior.

4.- Resultados experimentales

En el Capítulo 3 hemos visto cómo implementar los algoritmos de [13] y [17], aunque los algoritmos de cluster no hemos logrado que funcionen. En los tres algoritmos, al intentar enviar mensajes por *broadcast* los sensores daban un error en el envío y no se recibía nada. No se especifica el error que da y no he sido capaz de solucionarlo. El código para enviar mensajes en *broadcast* lo he probado por separado, sólo realizando envío y recepción de mensajes en *broadcast*, sin realizar nada más, y funciona correctamente. Una vez que tenemos los algoritmos implementados y los programas cargados en los sensores, el siguiente paso es realizar pruebas de rendimiento con los mismos. En este capítulo vamos a ver las pruebas realizadas y comentar los resultados obtenidos.

4.1.- Pruebas realizadas y resultados

En el apartado 1.3 dijimos que el objetivo del trabajo realizar las pruebas de los algoritmos en la finca de prácticas de la Universidad Pública de Navarra. A la hora de realizar las pruebas, no hemos tenido acceso a la finca de prácticas por problemas de horario por lo que hemos tenido que cambiar el lugar de las pruebas. Las pruebas finalmente las hemos realizado en una sala interior, sin paredes bloqueando la línea de visión de los nodos aunque sí con objetos que puedan influir algo en la línea de visión, donde hemos desplegado los sensores de la forma indicada anteriormente, formando una malla de 3x3 pero con una separación de 3 metros entre cada nodo en vez de 5 como habíamos diseñado originalmente. He supuesto que, al no ser la diferencia de distancia entre nodos muy diferente entre lo que se había planeado y lo que finalmente se ha probado, los resultados obtenidos en las pruebas serían muy similares en caso de realizarlos como se había planeado. Además, he realizado pruebas de envío y recepción de mensajes a una distancia aproximada de 15 metros en interior, con paredes en la línea de visión de los nodos, y el envío y recepción de mensajes se realiza de forma satisfactoria por lo que podemos suponer que el envío y recepción de mensajes a esta distancia pero en exterior, que es la distancia máxima entre nodos que habíamos planeado, también se realizaría de forma satisfactoria.

Las pruebas sólo las hemos podido realizar con el algoritmo DAWIN al ser el único finalizado y en funcionamiento. En este algoritmo el único valor que puede variar es el del

tiempo que pasa en la estación base entre envíos de mensajes inicializadores, es decir, el tiempo que pasa entre mediciones. He realizado pruebas fijando ese tiempo en 300 milisegundos, para obtener una cantidad considerable de medidas, y en 1, 10 y 30 minutos que podrían ser tiempos reales entre medidas en una explotación. En el primer caso, el tiempo de *timeout* fijado en la estación por si no recibimos es de 5 segundos mientras que en el resto de casos el tiempo de *timeout* es de 20 segundos. Además, el inicio de la ejecución ponemos a hora 0:00.00 el reloj de la estación base de forma que podamos controlar de forma sencilla en qué momento se reciben mensajes y el tiempo que están los sensores ejecutándose.

Para obtener resultados, monitorizamos el puerto COM de la estación base, para lo que utilizaremos el programa X-CTU, donde imprimimos cuando enviamos un mensaje inicializador, y si se ha enviado correctamente, y los mensajes de respuesta de las dos sub-cadenas, sus datos y el momento en el que se ha recibido el mensaje con los datos. Sólo monitorizamos la estación base por lo que en caso de no recibirse algún mensaje de respuesta con los datos, no sabremos qué nodo es el que ha fallado. Esto lo hemos realizado así ya que, en caso de desplegar los nodos en una explotación real, sería lo que podríamos hacer al no poder tener los sensores conectados a un ordenador y además, en caso de querer monitorizar todos los sensores, los tendríamos que tener conectados a un ordenador por lo que el valor de la batería obtenido no sería el correcto ya que da siempre valores en torno al 90% de batería. En la Figura 33 podemos ver un ejemplo de lo que se mostraría en una iteración del algoritmo. El mensaje inicializador se ha enviado correctamente, de ahí el mensaje “Paquete enviado OK” ya que si no pondría “Paquete enviado error”, y se ha recibido mensajes de respuesta de ambas sub-cadenas, hacia adelante y hacia atrás.

```
.~...RMY.~...RMY...~.}3...}3..@i8....#...1#.
~...RMY...Paquete enviado
.ok
.Paquete A recibido n..mero: 1
.Time: Sunday, 13/01/01 - 00:00.04
.MAC Address Source: 013A2040693815
.Network Address Origin: 02
.MAC Address Origin: 00000000
.Data: 1#2#1#0002*35#
.Paquete B recibido n..mero: 1
.Time: Sunday, 13/01/01 - 00:00.08
.Network Address Source: 00
.MAC Address Source: 013A204071E58F
.Network Address Origin: 010
.MAC Address Origin: 00000000
.Data: 1#1#1#0002*35#0003*91#0004*62#0005*88#
0006*66#0007*64#0008*93#0009*30#0010*88#
.Delay de 10 minutos
```

Figura 33. Ejecución de una iteración de DAWIN en la estación base

Con los resultados obtenido de las pruebas, vamos a analizar el número de rondas que ha estado el algoritmo ejecutándose, para lo cual llevaremos un contador en la estación base, el número de paquetes de inicialización enviados correctamente y el número de paquetes de respuesta de ambas sub-cadenas recibidos en la estación base. Además, de los datos recogidos de los sensores vamos a comprobar el nivel de batería de los nodos al inicio de la ejecución, a mitad y al final. No disponemos de baterías suficientes para todos los sensores por lo que algunos de los sensores están conectados a un ordenador para poder funcionar y el valor de su batería es irrelevante. Sólo disponíamos al realizar las pruebas de 5 baterías que funcionasen por lo que sólo hemos obtenido medidas de batería de 5 de los 9 sensores que forman la cadena. Uno de estos sensores es el sensor que recibe el mensaje inicializador para comprobar si el tener

que enviar dos mensajes en vez de uno como el resto de nodos de la cadena afecta al consumo de batería.

En la Tabla 5 se pueden observar el tiempo de ejecución, el número de rondas y los paquetes enviados y recibidos en la estación base.

Ejecucion	Tiempo ejecución	Rondas	Enviados OK	Rec. Forward	Re. backward
300 msg	34 minutos	240	228	204	204
1 min	1 hora 25 minutos	72	70	63	63
10 min	3 horas	19	19	19	18
10 min 2	1 hora 50 minutos	12	12	12	12
30 min	6 horas	13	13	13	13

Tabla 5. Tiempos y paquetes enviado y recibidos

En la Tabla 6 podemos observar los valores de la batería al inicio (B I), a mitad (B M) y al final (B F) de la ejecución junto al identificador del nodo que ha enviado los datos.

Ejecucion	B I 2	B M 2	B F 2	B I 4	B M 4	B F 4	B I 6	B M 6	B F 6	B I 7	B M 7	B F 7	B I 9	B M 9	B F 9
300 msg	38	35	35	64	59	55	68	63	59	66	61	59	33	30	29
1 min	39	37	35	67	59	56	69	62	59	69	62	59	38	34	33
10 min	43	38	34	81	74	70	83	77	71	84	76	72	42	35	32
10 min 2	45	43	41	70	65	61	72	66	63	73	69	65	42	39	36
30 min	35	28	13	62	49	38	66	53	42	64	52	40	30	16	0

Tabla 6. Valor de la batería

En las pruebas realizadas he podido observar que el algoritmo falla si elegimos al nodo 5, 6 o 7 de la Figura 13 como inicializadores de la cadena. Esto es debido a que tal y como están planteadas las Tablas 3 y 4 para elegir al siguiente nodo, si por ejemplo elegimos al nodo 6 como inicializador, envía el mensaje al 5 al ser su vecino de abajo. Este recibe un mensaje de su vecino de arriba y al no tener ningún vecino abajo o a la derecha envía el mensaje a la estación base dejando los nodos 1, 2 y 3 sin recoger sus datos. Si vamos hacia atrás pasa algo similar al llegar al nodo 7 al recibir un mensaje de abajo y no tener vecino arriba o a la izquierda. El problema es parecido si el nodo inicializador es el 5 o el 7. Para evitar esto habría que pensar un mecanismo que detecte estos nodos problemáticos y en esos caso al recibir un mensaje inicializador de la estación cambiase las tablas de decisión para que la que normalmente es la tabla hacia atrás sea hacia adelante y viceversa.

4.2.- Análisis de los resultados obtenidos

Observando los datos presentados en la Tabla 5, podemos ver cómo a medida que aumentamos el tiempo entre envíos, la probabilidad de que no se envíen mensajes de inicialización disminuye hasta conseguir un 100% de envíos en las pruebas con tiempos de 10 y 30 minutos. En el caso de los envíos cada 300 milisegundos y 1 minuto, se consiguen envíos un 95 y un 97% de las veces respectivamente lo que implica una gran tasa de envíos correctos. Al no indicar el tipo de error, no podemos saber el motivo del fallo aunque en el caso de los envíos cada minuto puede ser debido a que cerca de los sensores había un router WiFi recibiendo tráfico y al trabajar tanto los sensores como el router en la frecuencia de los 2.4GHZ pueden haberse produciendo interferencias lo que explicaría también las perdidas en los mensajes de datos. En cuanto a los mensajes con datos, de nuevo podemos observar que las pruebas de envío cada 10 y

30 minutos nos devuelven una tasa de acierto de casi el 100%. Sólo en un caso no se ha recibido datos de una de las sub-cadenas. Este fallo lo podemos atribuir a un fallo puntual de algún sensor o a una pérdida del mensaje debido a una colisión con otro mensaje. En el caso de envíos cada menos tiempo, hemos recibido mensajes con los datos aproximadamente un 90% de las veces que el mensaje de inicialización de la estación base se ha enviado correctamente. Aunque el número de mensajes recibidos correctamente de ambas sub-cadenas es el mismo en ambos casos, viendo los mensajes recibidos pude observar que había veces que se recibía uno y otro no. Además, en los envíos cada 300 milisegundos había veces que pasaban los 5 segundos de espera sin recibir mensajes y en la siguiente ronda nos mostraba dos mensajes. Este hecho nos puede indicar que algunos de los mensajes que no se han recibido haya podido ser ya que el tiempo de espera sea muy corto y que, al seguir la ejecución del programa, se haya solapado la recepción de un mensaje de datos con el envío de la trama inicializadora del siguiente, causando la perdida de uno de los dos mensajes.

En cuanto a la batería consumida, podemos observar cómo el envío y recepción de mensajes afecta bastante al nivel de la batería ya que por ejemplo en las mediciones con envíos cada 300 milisegundos, cada minuto y la segunda prueba con envíos cada 30 minutos indican que el consumo de batería ha sido muy similar en los tres casos en todos los nodos medidos pero la diferencia es que el tiempo que han estado los nodos funcionando ha sido de 34, 85 y 110 minutos respectivamente. Es decir, la prueba de envío cada 10 minutos ha estado funcionando más de 3 veces la prueba cada 300 milisegundos y ha consumido la misma batería. Por el contrario, en la prueba cada 300 milisegundos los nodos que forman la cadena han enviado y recibido casi 17 veces más mensajes. Además, podemos observar como el elegir siempre el mismo nodo como inicializador, en este caso el 2, lo que implica que envíe dos mensajes en vez de uno, no afecta significativamente a la batería. De hecho, en la mayoría de las pruebas este nodo ha consumido menos batería que el resto aunque la diferencia no es significativa y la podemos achacar al estado de las baterías.

Aunque no hayamos podido obtener resultados de pruebas de los algoritmos de cluster al no funcionar, sí que podemos hacer una comparación teórica del número de mensajes enviados y recibidos, y de la robustez de los algoritmos ante fallos observando los pseudo-códigos. Esta comparación la podemos hacer también de forma teórica con el algoritmo DAWIN. Esto nos puede dar una idea de que algoritmo puede consumir más batería fijándonos sólo en el uso del módulo de comunicaciones que es la principal fuente de consumo de batería en los sensores.

Observando los pseudo-códigos de los cuatro algoritmos, podemos ver que en el algoritmo DAWIN cada sensor sólo envía y recibe un mensaje por ronda, dos mensajes enviados en el caso del nodo inicializador. Si vamos rotando el nodo que recibe la trama inicializadora, todos los nodos consumen a la larga más o menos la misma batería al enviar de media un mensaje y recibir un mensaje. En caso de que un nodo detecte que se va a quedar sin batería enviaría un mensaje a sus vecinos aunque esto no va a influir mucho en el consumo. En el nodo emisor no tiene importancia ya que se va a desconectar y en los vecinos receptores tendría una influencia pequeña al no ser habitual que se produzca este caso y no tener muchos vecinos.

En el primer algoritmo de cluster, si el líder no falla envía un mensaje y recibe tantos mensajes como nodos haya. El resto de nodos envía y recibe un mensaje. En caso de que haya que elegir un nuevo líder, ya sea porque el actual no haya enviado mensaje de *IAmAlive* porque este caído, o porque se haya recuperado de un fallo y su *incarnation* haya aumentado, en la siguiente ronda habrá que elegir un nuevo líder. Para ello todos los nodos enviarán un mensaje y recibirán tantos como nodos hay en la red activos. Luego, una vez elegido el líder, todos los nodos volverán a enviar un mensaje y el líder recibirá uno de cada nodo. Si el nodo líder no

falla, su consumo de energía es mayor que en el resto de nodos ya que, al seguir siendo líder, recibe más mensajes que el resto por lo que su batería se acabará antes.

En el caso del segundo algoritmo de cluster, todos los nodos envían un mensaje y reciben tantos mensajes como nodos estén al alcance del nodo. Este comportamiento se mantiene haya que elegir un nuevo líder o no. Esto hace que los nodos con mayor conectividad con el resto consuman más batería aunque en nuestro caso, al estar todos al alcance de todos, el consumo sería similar.

El comportamiento del tercer algoritmo de cluster es muy similar al segundo ya que, en cada ronda, cada sensor envía un mensaje y recibe un mensaje de cada nodo de la red que tiene al alcance. Además, la primera vez un nodo recibe un mensaje de otro lo renvía. A la larga eso no supondría mucho gasto energético ya que en toda la ejecución del programa como mucho enviará un número de mensajes extra igual al número de nodos de la red. De nuevo, en nuestro caso todos los nodos tienen un comportamiento igual en cuanto al número de mensajes enviados y recibidos por lo que su consumo de batería debería ser similar.

En el caso de los algoritmos de cluster la mayoría de los mensajes se envían en broadcast por lo que se envían con la máxima intensidad que nos permita el módulo de comunicaciones. En el caso del algoritmo DAWIN estos mensajes se envían punto a punto por lo que se podría implementar un mecanismo que detectase en los la intensidad mínima con la que hay que enviar los mensajes para que lleguen a los vecinos. De esta forma, al enviar los mensajes con menos intensidad, se ahorraría batería.

Si realizamos una comparación entre los algoritmo, podemos observar que el algoritmo DAWIN es el que menos mensajes envía ya que por lo general cada nodo sólo envía y recibe un mensaje. En el caso del primer algoritmo de cluster, los nodos que no son líderes también envían y reciben un mensaje pero el líder recibe tantos como nodos haya. Esto hace que, si no hay cambios habitualmente de líder, el número de mensajes enviados y recibidos sea similar al algoritmo DAWIN. En caso de que los nodos fallen constantemente, se puede dar el caso de que haya que hacer muchos cambios de líder lo que implicaría mayor envío y recepción de mensajes en cada nodo y mayor consumo. En el segundo y tercer algoritmo de cluster el número de mensajes es superior a los anteriores ya que, si bien cada nodo sólo envía un mensaje, recibe tantos mensajes como nodos haya en la red a su alcance. Aunque la recepción de mensajes consume menos batería que el envío, el consumo va a ser mayor que en algoritmo DAWIN o el primer algoritmo de cluster.

Podemos concluir que en cuanto a consumo de batería fijándonos únicamente de los mensajes enviados y recibidos el algoritmo DAWIN es el más eficiente, seguido por el primer algoritmo de cluster siempre que no se den cambios habituales de líder y por último el segundo y el tercer algoritmo de cluster más o menos a la par.

Otro factor importante a la hora de evaluar que algoritmo consideramos que es mejor es la robustez del mismo, es decir, la tolerancia que tiene para seguir funcionando ante el fallo de un sensor.

El algoritmo DAWIN tal y como está planteado no presenta ningún mecanismo de detección o recuperación ante un fallo de un nodo por lo que el fallo permanente en un nodo va a producir que la cadena se corte y no se recojan datos de parte de la red. Sí que presenta un mecanismo de detección de batería baja que reconfigura la cadena en ese caso y apaga el sensor para evitar que la cadena quede rota. En el Capítulo 5 veremos una idea de cómo implementar un mecanismo de detección de fallos y reconfiguración de la cadena.

Los algoritmos de cluster presentan una mayor robustez ante fallos ya que la caída de un nodo, ya sea de forma temporal o permanente, no va a influir en el resto de nodos. Como mucho, si el nodo era líder en el momento del fallo, se dejarán de recoger datos en esa ronda pero a la siguiente se elegirá un nuevo líder y el algoritmo seguirá funcionando correctamente.

Una ventaja que tiene el algoritmo en cadena es que, por regla general, los sensores sólo necesitan poder comunicarse con sus vecinos y estos no cambian a no ser que algún nodo falle. Los algoritmos de cluster implican en general una mayor dependencia de comunicación entre los sensores ya que por ejemplo en el primer algoritmo todos tienen que poder comunicarse con todos, en el segundo al menos un sensor se tiene que poder comunicar con todos y en el tercero tiene que existir un camino multi-salto entre al menos un nodo y el resto. Esto hace que, con el mismo número de sensores, el algoritmo en cadena ofrezca mayor flexibilidad a la hora de poder abarcar una explotación mayor con el mismo número de sensores o una explotación con más obstáculos de comunicación al necesitar una menor comunicación entre los nodos de la red.

5.- Conclusiones y líneas futuras

5.1.- Conclusiones

La primera conclusión que puedo sacar de la realización de este trabajo es la dificultad de implementar programas para los sensores Waspmote. Por un lado me he encontrado que la documentación o ayudas para programar estos sensores es muy escasa reduciéndose casi exclusivamente a la página Web de Libelium [31]. Además, el IDE de programación no presenta ningún método de depuración de programas por lo que la única manera de probar un cambio y ver si funciona es subiendo el programa al sensor y viendo su ejecución por medio de impresiones de texto por el puerto USB. Subir un programa al sensor, teniendo en cuenta que implica la depuración del mismo aunque no hayamos realizado ningún cambio en el programa, por ejemplo si queremos subir el mismo programa a dos sensores, cuesta por ejemplo en los programas que hemos realizado en este trabajo más de un minuto y medio. También me he encontrado con que hay veces que el programa falla pero no sabes porque. Ocurre por ejemplo que al intentar enviar un mensaje este puede dar un error pero no hay manera de saber por qué sucede el error ya que no hay más información sobre el mismo. Esto causa que querer implementar algo nuevo y probarlo, aunque sea una cosa mínima, lleve una gran cantidad de tiempo en pruebas realizadas.

Por otra parte, también me he encontrado con que los sensores no son muy robustos ni a la hora de ejecutar programas ni a nivel hardware. Me ha ocurrido varias veces que la ejecución de programas no es la lógica. Por ejemplo, como ya he comentado, a la hora de concatenar varios datos seguidos sobre una variable de texto he tenido que introducir delays en algunos casos entre las operaciones ya que si no el programa entraba en un bucle infinito. Esto también me ha costado bastante tiempo en pruebas hasta dar con la solución. También me ha pasado que, tras subir un programa a un sensor, aunque el IDE de programación indicaba que se había subido correctamente, al encender el sensor éste no hacía nada necesitando volver a cargar el programa, sin cambiar nada, para que funcionase. A nivel hardware hay que tener bastante cuidado con dos elementos bastante frágiles que son los jumpers y las baterías. Con los primeros hay que tener cuidado sobre todo con el jumper de programación ya que en caso de rotura el sensor no se puede volver a programar. Con la batería hay que tener mucho cuidado ya que al tener unos cables muy finos, unido a que el puerto donde se conecta la batería es de muy difícil

acceso al estar en mitad de la placa y al lado del puerto USB, dificulta mucho la tarea de colocar y quitar la batería. Es relativamente sencillo que al intentar poner o quitar la batería se rompa alguno de los cables de la misma obligando a realizar tareas de bricolaje para repararla o quedar inservible.

Relativo al trabajo realizado, he podido comprobar cómo por medio de una red de sensores se puede obtener mucha información que ayude en todo momento a tener monitorizada una explotación agrícola. Habría que considerar si el beneficio económico que esto puede aportar, sabiendo por ejemplo en todo momento los valores de humedad, temperatura, etc. de la explotación, con la posible mejora de calidad en el producto final que esto puede tener, compensa el desembolso económico que hay que realizar a nivel hardware para poder desplegar la red de sensores.

En cuanto a nivel de algoritmo, he podido ver, aunque sea a nivel teórico, la diferencia entre algoritmos de recogida de datos en una estructura de cadena y en una estructura de cluster. Dependiendo de la tasa de fallos de los sensores, de la facilidad para recargar las baterías, lo que implicaría que la misma tenga que durar más o menos tiempo, o de la topología del terreno de la explotación agraria habría que estudiar cuál de los algoritmos planteados se adapta mejor a las exigencias.

5.2.- Líneas futuras

Las líneas futuras de trabajo de este algoritmo las podemos separar en dos dependiendo del elemento sobre el que se quiera trabajar. Estas dos vías de trabajo coinciden con los dos tipos de artículos que vimos en el apartado 1.2. Por una parte podemos trabajar en la creación de un sistema completo que permita desde recoger datos hasta consultarlos en cualquier lado, incluso crear un sistema de toma de decisión que actúe de manera autónoma. Por otro lado podemos trabajar en el algoritmo de recogida de datos ya sea mejorando los que hemos presentado o creando nuevos algoritmos.

5.2.1.- Sistema completo de monitorización

En este trabajo hemos implantado una red de sensores que recojan datos de una explotación agraria. El siguiente paso sería, a partir de estos datos, crear un sistema completo de monitorización que nos permita realizar agricultura de precisión. Para ello, el siguiente paso sería recoger los datos de cada una de las explotaciones agrarias y enviar estos datos a un punto central. Este paso se podría hacer por medio de un Meshlium ya que nos permite, gracias a sus mayores capacidades, realizar un pre-procesamiento de los datos y por ejemplo almacenar un mayor número de datos antes de enviarlos. Además, gracias a que puede trabajar con varios protocolos de comunicación al mismo tiempo, sería capaz de trabajar con 802.15.4/ZigBee para comunicarse con los Waspmote y luego podría enviar estos datos al lugar de procesamiento central por medio de WiFi en caso de estar a una distancia mayor.

Una vez realizado esto, en el lugar de procesamiento central habría que tratar estos datos y presentarlos por medio de un servicio Web o similar para su consulta a través de Internet, o presentarlos en un formato adecuado a una aplicación móvil para poder ser consultado desde un Smartphone.

Por último, se podría crear un sistema de toma de decisión e implantar una serie de actuadores en las explotaciones agrarias de manera que el sistema pueda actuar de forma autónoma cuando detecte una serie de características predefinidas. Por ejemplo, se podría

colocar una serie de actuadores sobre las tomas de riego de la explotación de forma que si, por medio de los sensores, detectamos que la tierra está muy seca, automáticamente se pueda iniciar la fase de riego.

En la Figura 34 podemos ver un esquema simplificado de los que podría ser un sistema completo de toma de decisión.

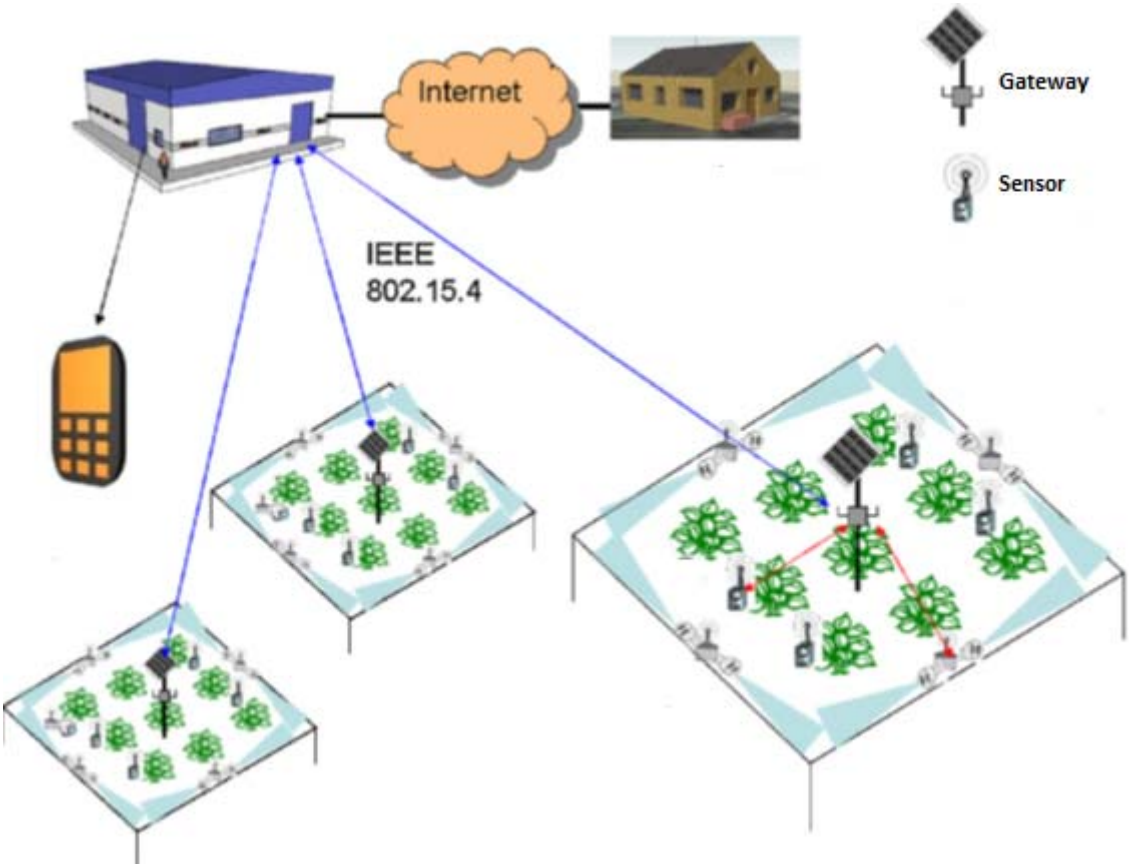


Figura 34. Sistema completo de toma de decisión

5.2.2.- Trabajo sobre los algoritmos

El primer trabajo que habría que hacer en relación a los algoritmos sería acabar de implementar y probar que funcionan los algoritmos de cluster y una vez hecho esto realizar unas pruebas similares a las realizadas con el algoritmo DAWIN para poder obtener datos y compararlos con los ya obtenidos.

Una vez realizado esto el siguiente paso sería implementar mejoras en los algoritmos que permitan por ejemplo un menor consumo de batería o una mayor robustez ante fallos.

Una de esas mejoras ya la he indicado anteriormente y sería arreglar el problema que tiene el algoritmo DAWIN al elegir como nodos inicializadores de la cadena determinados nodos de la red. Para ello, habría que crear un mecanismo que identifique esos nodos que en nuestro trabajo serían los nodos 5, 6 y 7 de la Figura 13. Una vez identificados esos nodos, cuando llegue una trama inicializadora a esos nodos intercambiaríamos las tablas de elección del siguiente nodo usando la que normalmente es la tabla *forward* para ir *backward* y viceversa.

Otra posible mejora que se podría implementar en el algoritmo DAWIN es que, dado que más o menos sabemos cuándo va a llegar el mensaje inicializador, ya que es periódico, se podría hacer que los nodos duerman durante ese determinado tiempo despertándolos un tiempo

antes y volviéndolos a dormir. De esta forma, sobre todo si el tiempo entre recogida de datos es bastante grande como por ejemplo en las pruebas de recogida de datos cada 30 minutos, se ahorraría gran parte de la batería gastada. Esto puede no tener mucho sentido en el caso de que las recogidas de datos se efectúen cada poco tiempo, por ejemplo con las pruebas de recogida cada minuto o cada 300 milisegundos, ya que se podría llegar a gastar más batería durmiendo y despertando los sensores que la que se ahorraría.

Por último, se podría implementar un mecanismo de detección de nodos que fallan y reconstrucción de la cadena. Al principio del algoritmo, la estación base podría pedir a los diferentes nodos una copia del mensaje que estos enviarían en caso de ir a quedarse y los nodos a los que la enviarían almacenaría estos datos. Luego, si la estación base detecta que no se están recibiendo todos los datos, por ejemplo porque durante dos rondas no hemos recibido alguno de los mensaje con los datos, enviaría un mensaje de *estasVivo* a todos los nodos. Si alguno de los nodos no responde al mensaje, la estación base enviaría el mensaje de reconfiguración de ese nodo a sus vecinos para que la cadena se reconfigure y no seguir perdiendo mensajes. Hecho esto volvería a pedir los mensajes de reconfiguración ya que estos han cambiado al cambiar los vecinos de algunos nodos.

6.- Bibliografía

[1] S. Liaghat; S.K. Balasundram, A Review: The Role of Remote Sensing in Precision Agriculture
[2] Agricultura de precisión, [http://es.wikipedia.org/wiki/Agricultura_de_precisi%C3%B3n]
[3] Miguel A. Fernandes; Samuel G. Matos; Emanuel Peres; Carlos R. Cunha; Juan A. López; P.J.S.G. Ferreira; M.J.C.S. Reis; Raul Morais, A framework for wireless sensor networks management for precision viticulture and agriculture based on IEEE 1451 standard, 2013
[4] T. De Filippis; L. Rocchi; E. Fiorillo; L. Genesio, A WebGIS application for precision vitivulture: From research to operative practices, 2000
[5] Manijeh Keshtgari; Amene Deljoo, A Wireless Sensor Network Solution for Precision Agriculture Based on ZigBee Technology
[6] Mauro Martinelli; Luca Ioriatti; Federico Viani; Manuel Benedetti; Andrea Massa, A WSN-Based solution for precision farm purposes
[7] Seong-eun Yoo, Jae-eon Kim; Taehong Kim; Sungjin Ahn; Jongwoo Sung; Daeyoung Kim. A2S: Automated Agriculture System based on WSN
[8] Herman Sahota; Ratnesh Kumar; Ahmed Kamal; Jing Huang, An Energy-efficient Wireless Sensor Network for Precision Agriculture
[9] D.D.Chaudhary; S.P.Nayse; L.M.Waghmare, Application of Wireless Sensor Networks for Greenhouse Parameter Control in Precision Agriculture
[10] Xin Dong; Mehmet C. Vuran; Suat Irmak, Autonomous precision agriculture through integration of wireless underground sensor networks with center pivot irrigation systems
[11] Wendi Rabiner Heinzelman; Anantha Chandrakasan; Hari Balakrishnan, Energy-Efficient Communication Protocol for Wireless Microsensors Networks
[12] Fauzi M. Shubeita; Cesar Missio Marcon; Antônio R. D. De Vit; Roberto Franciscatto; Marco A. Spohn3, Evaluation of a Wireless Sensor Network Applied to Precision Agriculture under Adverse Weather Conditions
[13] Mikel Larrea; Cristian Martín; José Javier Astráin. Fault-tolerant aggregator election and data aggregation in wireless sensor networks. Int. J. Communication Networks and Distributed Systems, Vol. 3, No. 2, 2009

[14] Pau Aragó Galindo; Carlos Granell; Paulo Guilherme Molin; Joaqun Huerta Guijarro, Participative site-specific agriculture analysis for smallholders
[15] Stephanie Lindsey; Cauligi S. Raghavendra; PEGASIS: Power-Efficient Gathering in Sensor Information Systems
[16] Zeldi Suryady; Mohd Hafiz Md Shaharil; Khairina Abu Bakar; Reza Khoshdelniat; Gopinath Rao Sinniah; Usman Sarwar, Performance Evaluation of 6LoWPAN-based Precision Agriculture
[17] M. Goyeneche; J. Villadangos; J.J. Astrain. S-DAWIN: A Self-adapted Distributed Algorithm for data gathering in Wireless sensor Networks. 2011
[18] Atis Elsts; Rihards Balass; Janis Judvaitis; Leo Selavo, SAD: Wireless Sensor Network System for Microclimate Monitoring in Precision Agriculture
[19] Juán Vicente Capella Hernández; Tesis UPV: Redes inalámbricas de sensores: una nueva arquitectura eficiente y robusta basada en jerarquía dinámica de grupos; Abril 2010
[20] Naseer Sabri; S. A. Aljunid; R. B. Ahmad; M.F. Malik; Abid Yahya; R. Kamaruddin; M.S. Salim, Wireless Sensor Actor Networks,
[21] Antonio-Javier Garcia-Sanchez; Felipe Garcia-Sanchez; Joan Garcia-Haro, Wireless sensor network deployment for integrating video-surveillance and data-monitoring in precision agriculture over distributed crops
[22] J.A. López Riquelme; F. Soto; J. Suardíaz; P. Sánchez; A. Iborra; J.A. Vera, Wireless Sensor Networks for precision horticulture in Southern Spain
[23] WiFi, [http://es.kioskea.net/faq/580-wifi-descripcion-y-explicacion-en-relacion-al-tipo-de-antena#]
[24] [http://www.ingeniosolido.com/blog/2010/06/zigbee-vs-bluetooth-vs-wi-fi-parte-1/]
[25] [http://www.ingeniosolido.com/blog/2010/08/zigbee-vs-bluetooth-vs-wi-fi-parte-2/]
[26] [http://www.ingeniosolido.com/blog/2010/09/zigbee-vs-bluetooth-vs-wi-fi-parte-3/]
[27] [http://www.stg.com/wireless/ZigBee_comp.html]
[28] [http://www.sena.com/blog/?p=359]
[29] [http://www.cse.ohio-state.edu/~xuan/courses/694/694_WIFI_BT_ZIGBEE_NFC.pptx.]
[30] [http://www.sensor-networks.org/index.php?page=0823123150]
[31] Libelium, [www.libelium.com]
[32] Proyectos con Libelium, [http://www.libelium.com/case-studies/]
[33] Digi Intrnational, [http://www.digi.com/]
[34] Comparativa Waspmote 1.1 vs 1.2, [http://www.libelium.com/uploads/2013/02/wasp_v11_vs_wasp_v12_guide.pdf]
[35] Algoritmo LEACH [http://en.wikipedia.org/wiki/Low_Energy_Adaptive_Clustering_Hierarchy]
[36] Curvas de Peano y Hibert, [http://www.dma.fi.upm.es/sonia/proyectos/tutorial-mov-brow/curvapeano.htm]
[37] Descarga IDE Wapmote [http://www.libelium.com/v11-files/downloads/waspmote-ide-v.02-windows.zip]
[38] Guía de programación de Waspmote, [http://www.libelium.com/development/waspmote/documentation/programming-guide/?action=download]