TRABAJO DE FIN DE CARRERA

TÍTULO DEL TFC: Localización de nodos dentro de una red inalámbrica de sensores.

TITULACIÓN: Ingeniería Técnica de Telecomunicaciones, especialidad Telemática.

AUTOR: Alberto Herrero García.

DIRECTORES: José Polo y Jorge Higuera.

FECHA: 22 de Junio de 2.009

Resumen

Este proyecto consiste en la implementación de un algoritmo de localización de nodos dentro de una red inalámbrica de sensores (WSN). Para ello hemos desarrollado unas rutinas en NesC utilizando el sistema operativo TinyOS orientado a redes de sensores WSN.

La técnica de localización empleada se denomina DV hop y permite localizar un nodo desconocido por medio del numero de saltos de un mensaje enviado a la red de sensores donde se han empleado nodos repetidores de primer nivel y nodos repetidores de segundo nivel. El hardware utilizado son los nodos sensores denominados motas que trabajan bajo la el estándar IEEE802.15.4 y además disponen de una interfaz USB, que a través de ella pueden ser conectados a un ordenador para ser configurados o para poder visualizar los datos recibidos en un hiperterminal de Linux o Windows.

Este proyecto lo hemos dividido en tres capítulos bastante bien diferenciados.

En primer lugar se hará una introducción de una serie de conceptos teóricos relacionados con las redes WSN tales como sus características principales, utilidades, dispositivos que se utilizan, y las principales técnicas de localización de nodos, centrándonos en el algoritmo DV hop utilizado en la implementación practica de nuestra WSN

Posteriormente, se explicará el funcionamiento de cada una de las aplicaciones desarrolladas mediante diagramas de flujo y configuraciones realizadas y como han sido preparadas para su correcto funcionamiento.

Por ultimo, una vez implementadas las aplicaciones necesarias, a partir de dos escenarios propuestos para comprobar su funcionamiento, se mostrarán los resultados obtenidos en las pruebas realizadas.

Overview

This project consists on the implementation of an algorithm of nodes localization in a wireless sensor network (WSN). To do this, we have developed some algorithms with NesC, using the operative system TinyOS which is suitable for programming in wireless networks.

The algorithm of localization used is called DV hop and allows locating an unknown node by the number of jumps in a sent message in the sensor network where first and second level nodes have been used. The hardware used are sensor nodes called “motas” that work on the 802.15.4 standard and in addition they have an USB interface and across it, can be connected to a computer to be configured or to visualize the received information in a Linux o Windows hyper terminal.

This project has been divided in three well enough differentiated chapters.

First of all, an introduction of some theoretical concepts related to WSN such as its principal features, utilities, used dispositives, and the main techniques of nodes localization inside a WSN, centring on the DV hop algorithm used in the practice implementation of our WSN.

Later, the functioning of each one of the applications developed through flow charts and configurations and how they have been prepared for his correct functioning will be explained.

Finally, once implemented the necessary applications, we have proposed two stages to verify its functioning, and the obtained results in the proofs realized will be shown.

ÍNDICE

INTRODUCCIÓN

Hoy en día, existe una gran variedad de dispositivos y sistemas electrónicos, los cuales utilizan sensores. Muchos de estos sensores carecían de la capacidad de procesar y analizar los datos que detectan, limitándose su funcionamiento a un transductor en que realizaba la medición de una variable física acondicionándola y enviando dicha información a un procesador central. El desarrollo de transceptores han permitido a los nodos sensores (motas), la capacidad de transmisión de datos vía RF. Dichos nodos pueden ser distribuidos en un área determinada operando en diferentes topologías de redes inalámbrica que permita el intercambio de información y sensado.

La creación de estándares globales orientados a redes de área personal (PAN-LR) como el estándar IEEE 802.15.4, permiten incrementar la interoperabilidad de las redes de sensores y el surgimiento de nuevas aplicaciones. Este estándar, está orientado a redes de sensores de baja velocidad, bajo coste y consumo de energía. Este tipo de redes, generalmente no requieren ningún tipo de i infraestructura fija ni administración centralizada, de tal forma que los nodos pueden operar tanto como dispositivo sensor terminal como de router, y si es necesario retransmiten la información hacia otros nodos.

En los últimos años, varios laboratorios de investigación involucrados tanto en el sector científico como en el industrial, han mostrado un gran interés por este tipo de redes debido a que pueden ser desplegadas en entornos de difícil acceso y sin infraestructura de red previa. En este tipo de redes existe un amplio rango de aplicaciones, entre ellas automoción, defensa, aplicaciones médicas, predicción medioambiental entre otras. Un ejemplo a esta mención, es DARPA (Defense Advanced Research Projects Agency), institución dependiente del Departamento de Defensa estadounidense que se involucró en su desarrollo mediante el proyecto DSN (Distributed Sensor Network) [3].

Este tipo de redes actualmente, está llevando a una revolución tecnológica similar a la que tuvo la aparición de Internet, y además se habla de redes de vigilancia global del planeta, capaces de registrar seguimiento de personas y mercancías concretas, monitorizar el tráfico, y varias iniciativas y proyectos de investigación que han despertado gran interés para ser aplicados en la práctica.

En este TFC nos centraremos en la descripción de algunos algoritmos para localizar nodos en una red de sensores WSN utilizando la información correspondiente al número de saltos de un mensaje enviado por un nodo sensor.

CAPÍTULO 1. CONCEPTOS TEÓRICOS

Las comunicaciones inalámbricas utilizan las ondas electromagnéticas que se propagan en el espacio sin necesidad de utilizar ningún medio guiado que comunique cada uno de los extremos de la transmisión

Actualmente existe, una gran variedad de dispositivos electrónicos hacen uso de este tipo de comunicación. Entre ellos los teléfonos móviles, ordenadores portátiles, mandos a distancia, etc.

1.1. Estándares de comunicaciones orientados a redes WSN.

Los estándares consisten en una recopilación de especificaciones que regulan la realización de procesos para garantizar la interoperabilidad de diversos productos.

En el entorno de las comunicaciones inalámbricas podemos mencionar algunos como:

- Bluetooth (IEEE802.15.1) [14]: permite comunicaciones radio de 720 kbps (1 Mbps de capacidad bruta) en radios de cobertura de entre 10 y 100 metros con un consumo de corriente de 40 mA. Los datos son sincronizados entre ordenadores, teléfonos móviles y otros periféricos tales como impresoras, PDAs, etc..

- IEEE802.15.4: permite transmisiones de datos de entre 20 a 250 kbps en radios de cobertura de entre 10 y 75 metros soporta bandas de radio de 2400-2483,5 MHz (utilizado en todo el mundo) empleando 16 canales. También es compatible en las bandas de 868-868,8 MHz (Europa) y 902-928 MHz (Norte América), hasta diez canales (2003) extendidos a treinta (2006). Este estándar es el que utilizaremos durante este proyecto y mas adelante describiremos con mas profundidad.

- Wimax (IEEE802.16) [15]: permite trabajar a una tasa de transmisión de 70 Mbps en radios de cobertura de hasta 48 kilómetros a frecuencias de 2,5 y 3,5 Ghz.

- Wifi (IEEE802.11) [28]: permite transmisiones de datos de entre 11 Mbps (IEEE802.11b) y 54 Mbps (IEEE802.11g) y opera en las bandas de radio de 2,4 - 2,5 Ghz.

1.1.1. IEEE802.15.4.

Este estándar [1] define el nivel físico y el control de acceso al medio en redes inalámbricas de área personal (LR-WPAN) y es la base de la tecnología inalámbrica llamada Zigbee.

Las principales características de este estándar son:

- Flexibilidad en la red debido a la facilidad de integración en la red mostrada por sus dispositivos ya que cada nodo puede iniciar su participación en la red, y el intercambio de información se realiza sin demasiado esfuerzo de instalación.

- Bajo coste, debido al uso de componentes de coste reducido.

- Bajo consumo de energía: se trata de uno de los objetivos primordiales de este estándar ya que al tratarse del uso de dispositivos inalámbricos, deberemos utilizar baterías y conseguir un consumo mínimo para evitar de reponer las baterías de manera frecuente. Para ello utilizamos una potencia de transmisión y un radio de alcance limitados (10 y 75 metros) y además utilizando unos ciclos de trabajos bastante bajos del orden de un 0,5%. Por ejemplo, en caso de que utilicemos una batería de una capacidad de 750 mAh en un rango de 10 metros, con un consumo de 10 mA de corriente en estado activo, nuestra batería podría alcanzar una duración de dos años si el ciclo de trabajo es inferior a 0,5%.

Por lo que respeta a la arquitectura de este estándar, la definición de sus niveles está basada en el modelo OSI (ver figura 1.1).

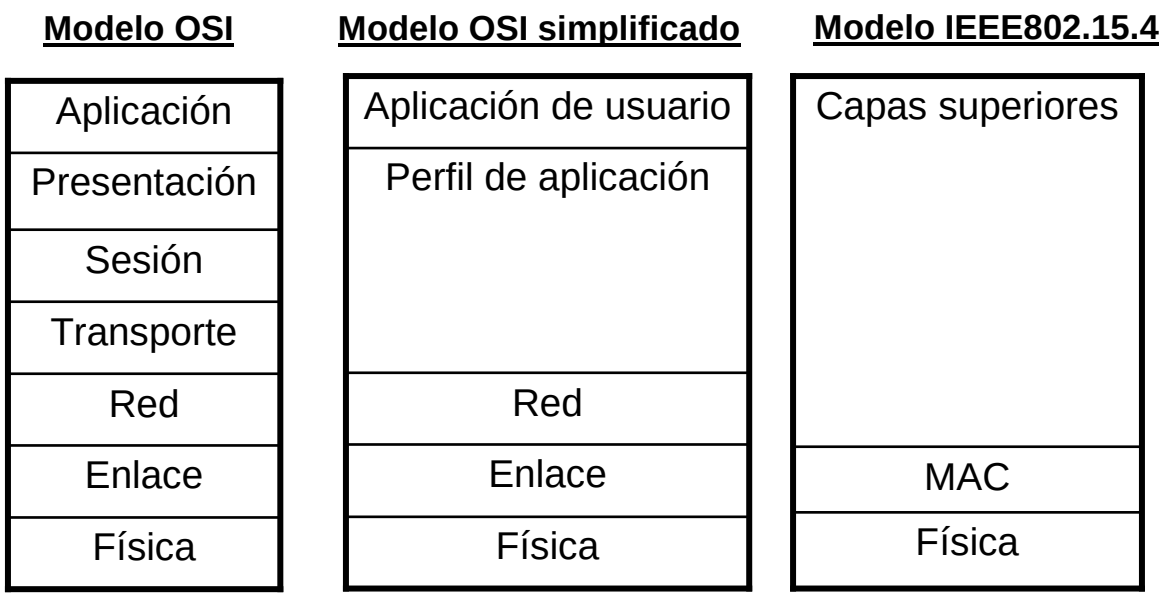

Fig. 1.1 Arquitectura de capas orientada a WSN.

Como podemos observar en la figura anterior, 802.15.4 se encarga únicamente definir el nivel físico y el control de acceso al medio (MAC).

Por lo que respeta al nivel físico, este se encarga de proporcionar el servicio de transmisión de datos sobre el medio y además controla el transceptor de radiofrecuencia, el consumo de energía de la señal y la selección de canales de acceso.

Por lo que respeta al control de acceso al medio (MAC), se utiliza la técnica CSMA/CA (Carrier Sense Multiplie Acces / Collision Avoidance) y de esta manera evitar colisión de tramas. En el caso de que un nodo desee transmitir una trama, en primer lugar el transceptor escuchará el medio. En caso que el medio este ocupado, la transmisión no podrá realizarse y tendrá que esperarse un cierto tiempo para volver a intentarlo y así sucesivamente hasta que encuentre el canal libre.

Este estándar trabaja en banda libre, ya que no requiere licencia para operación, y por eso, debe emplear una baja potencia de transmisión en el transceptor (0 dbm=1mW), CC2420 en nuestro caso. Estas bandas se llaman ISM (Industrial Scientific & Medical), empleadas para usos científicos, médicos e industriales a frecuencias de 868 Mhz en Europa, 915 Mhz en EEUU y 2,4 Mhz en todo el mundo que es la frecuencia que las empresas eligen para diseñar un dispositivo ya que es soportado en todo el mundo.

1.2. Redes inalámbricas de sensores (WSN)

Con el avance de la tecnología, se ha conseguido poder desarrollar dispositivos sensores distribuidos con tamaño reducido, bajo coste y consumo reducido, capaces de procesar información de manera local y pueden comunicarse de forma inalámbrica, y trabajar de forma cooperativa.

Una red inalámbrica de sensores consiste en una gran cantidad de dispositivos, llamados nodo sensores, capaces de recoger información de su entorno, tales como humedad, luz, temperatura, etc, mediante el uso de los sensores que llevan incorporados estos dispositivos y además de estos también existen los nodos repetidores que se encargaran de encaminar los datos hacia la estación base que se encuentra conectado a un ordenador que puede comunicarse hacia el exterior a través de Internet o una red de área local (LAN).

Los nodos sensores, se encuentran esparcidos por toda la red en diferentes tipos de topologías, que mas adelante explicaremos, dependiendo de la aplicación. Un ejemplo de red de sensores la vemos en la figura 1.2.

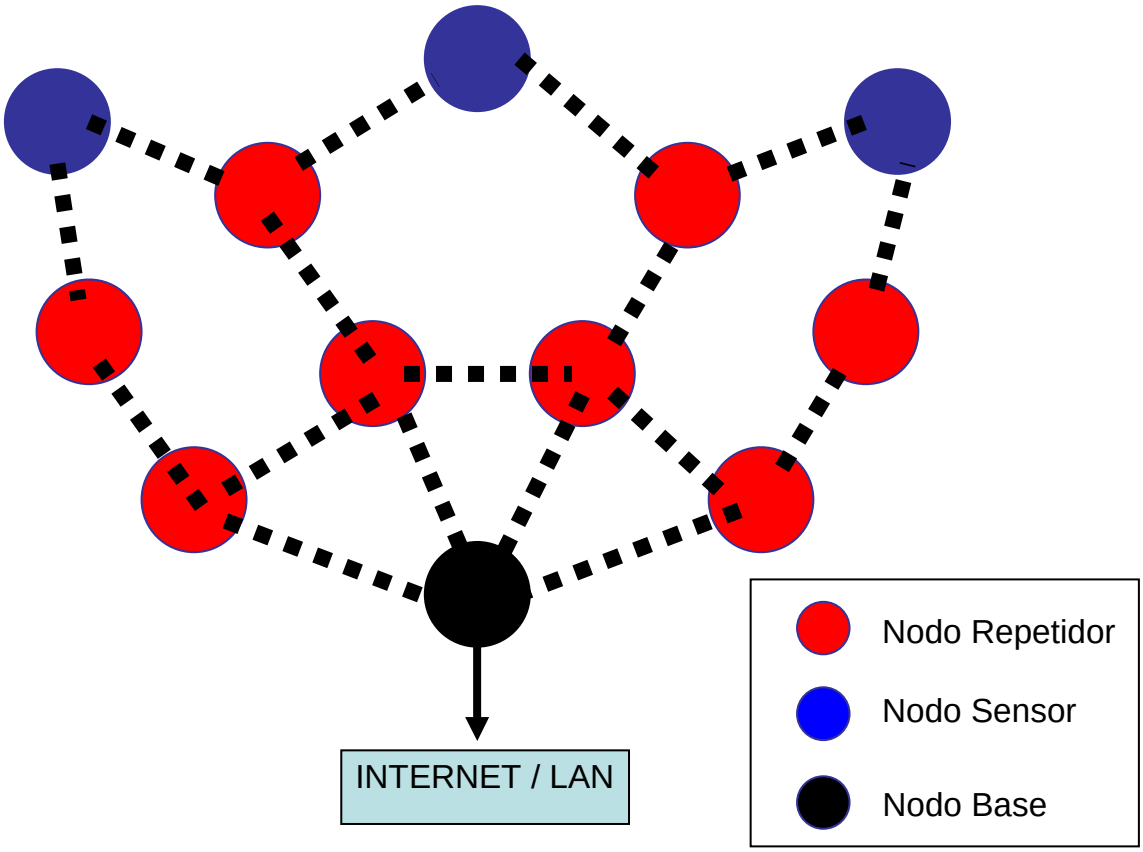

Fig. 1.2 Estructura de una red de sensores

1.2.1. Características.

A la hora de diseñar una red de sensores tenemos que tener en cuenta las siguientes características:

- Topología: es la manera en que los nodos de una red está distribuida. Existen varios tipos de topología tales como, estrella, malla, árbol entre otras. En nuestro, el algoritmo de localización lo hemos orientado a la topología de jerarquía de árbol en que distinguimos sensores, repetidores de primer y segundo nivel y nodo base.

- Medio de transmisión: en una red de sensores, los nodos están conectados de manera inalámbrica mediante radio, infrarrojo u óptico. En nuestro caso, utilizamos el transceptor CC2420 que opera en la banda de 2.4Ghz bajo estándar IEEE802.15.4

- Consumo energético: los nodos sensores que utilizaremos, al tratarse de un dispositivo de escaso tamaño, poseen una fuente energética bastante limitada, dependiendo de las baterías utilizadas. En nuestro caso, se trata de unas baterías de 1.2V y 2700 mAh de capacidad. Por lo que el tiempo de vida de un nodo viene condicionado por la batería y el ciclo

útil de trabajo de cada nodo, ya que en algunos casos la recarga de ellas depende de los módulos de alimentación adicionales, utilizando paneles solares para pilas recargables, aunque esta solución se encuentra fuera del alcance de este trabajo. Como anteriormente hemos comentado, los nodos sensores tienen la función de recolectar la información y posteriormente enrutarlas hacia el nodo base si es necesario, y en el caso de que un nodo se quedase sin batería, podría suponer un cambio significativo de la topología de la red y se requerirá un nuevo enrutamiento y una reorganización de la red.

1.2.2. Aplicaciones de las redes de sensores.

Las redes de sensores 802.15.4 pueden ser utilizadas en los siguientes ámbitos:

- Entornos de alta seguridad, tales como aeropuertos (medición de ruido acústico), centrales nucleares (sensores para la medición de radiaciones ionizantes), edificios de gubernamentales (sensores de movimiento).

- Entornos ambientales tales como bosques u océanos que se requiere tener un control de diversos parámetros tales como temperatura, humedad entre otras. De esta manera también se puede detectar y prevenir condiciones climáticas adversas.

- Entornos industriales para el monitoreo y diagnóstico del funcionamiento de de motores eléctricos y de combustión interna.

- Medicina: de esta manera, se podrá tener controlada las constantes vitales del paciente tales como pulsaciones, nivel de azúcar, presión arterial, etc.

- Domótica: de esta manera podremos tener controlado un hogar mediante sensores de intrusión e incendios entre otros. El uso de sensores de tamaño reducido, precio asequible y su velocidad de despliegue hacen bastante factible el uso de este tipo de redes para domotizar una vivienda.

- Trafico: mediante estos sensores, se puede determinar la presencia o ausencia de vehículos y de está manera se puede tener controlado las vías de circulación o modificar el estado de tráfico según la situación.

De hecho las redes de sensores inalámbricas (WSN) tienen el potencial de revolucionar los complejos sistemas de control u observación ya que se está tratando de una tecnología con un futuro bastante prometedor.

1.2.3. Dispositivos utilizados.

Los dispositivos utilizados para formar una red sensores inalámbrica, denominan motas. En este TFC hemos trabajado con motas Tmote Sky (figura 1.3), desarrolladas originalmente en la universidad de Berkeley procedentes del fabricante Moteiv [18], actualmente llamado Sentilla.

Fig. 1.3 Dispositivo Tmote Sky.

Las características principales de este dispositivo, son:

- Posee un transceptor IEEE 802.15.4 que utiliza el chip C2420 compatible con las bandas ISM (2.4 GHz hasta 2.4835 GHz) a velocidad de 250 kbps.

- Antena integrada en la placa base.

- Posee un microcontrolador MSP430F1611 de de 16 bits con 10 kB de RAM.

- Bajo consumo: teniendo en cuenta la potencia de transmisión máxima de 0 dbm (1 mw), para recepción se utiliza un máximo de 23 mA, mientras que en transmisión como máximo son 21 mA. Sin embargo en nuestro caso es menor porque manipulamos el registro de

control para la potencia de transmisión del transceptor CC2420, de manera que el nivel de potencia queda por debajo de los -25 dbm. Por otro lado, el microcontrolador utiliza una corriente de 1 mA cuando esta en modo activo, mientras que en modo “sleep” la corriente consumida es de 100 µA. como podemos ver, la corriente empleada por el microcontrolador es muy inferior frente a la utilizada por el transceptor CC2420.

- Dispone de 48 Kb de memoria de programa y una memoria flash externa de 1MB.

- Programación y recolección de datos a través de la interfaz USB.

- Alimentación mediante 2 pilas alcalinas AA (de 1,5 V) o mediante el puerto USB.

- Variedad de sensores: Luz, temperatura, humedad, tensión de la batería y además, dispone de un conector externo para sensar hasta tres canales de conversión A/D.

- Funciona bajo TinyOS 1.1 o superior.

Principalmente nos hemos decantado por utilizar motas por su compatibilidad con TinyOS (el sistema operativo empleado en este TFC), la interfaz USB ya que nos permite configurar la mota a través de ordenador con bastante facilidad y el hecho de que opere bajo el estándar 802.15.4.

Las motas están formadas por 4 dispositivos: sensores, microcontrolador, conversor y transceptor CC2420 (para más detalle ver Anexo F).

- Sensores: se encargan de tomar la medida de una magnitud específica tal como temperatura, humedad…

- Microcontrolador: se encarga de procesar la información que proviene de los sensores y de ejecutar el programa instalado previamente. En este dispositivo se incluye un conversor A/D 12 bits

- Conversor RS232 /USB: este dispositivo se encarga de convertir el puerto USB a un puerto virtual RS232.

- Transceptor CC2420: envía y recibe información procedente o hacia otras motas a través del canal de radio.

1.2.4. Sistemas operativos.

Para las redes WSN, existen varios sistemas operativos para la programación de motas, entre ellos:

- Nut [21]: esta plataforma trabaja con CPUs de 8 bits y se utiliza para aplicaciones en tiempo real.

- Ecos [22]: es un sistema operativo libre y gratuito desarrollado por la empresa Red Hat. Está diseñado para aplicaciones y sistemas embebidos que solo necesitan un proceso. Funciona sobre varias arquitecturas, entre ellas x86, PowerPC, MIPS o ARM.

- Eyeos [23]: se define como un entorno para escritorio basado en Web, permite monitorizar y acceder a un sistema remoto mediante un sencillo buscador.

- MagnetOS [17]: es un sistema operativo distribuido para redes de sensores o adhoc, cuyo objetivo es ejecutar aplicaciones de red que requieran bajo consumo de energía, adaptativas y fáciles de implementar.

- TinyOS: este sistema operativo es el que emplearemos en este proyecto y posteriormente pasaremos a explicar con más detalle. Fue desarrollado en la universidad de Berkeley.

- LiteOS [25]: Sistema operativo desarrollado en principio para calculadoras, pero que ha sido también utilizado para redes de sensores.

1.2.4.1. TinyOS.

El sistema operativo utilizado en este proyecto es TinyOS, desarrollado en la Universidad de Berkeley, en el que su diseño se basa en responder a las características y necesidades de las redes de sensores inalámbricos, tales como su reducido tamaño de memoria, bajo consumo de energía y operaciones de concurrencia intensiva en el que se ejecutan múltiples tareas interactivas de forma simultánea.

Además se encuentra optimizado en términos de uso de memoria y eficiencia de energía. El diseño del Kernel (núcleo) de TinyOS está basado en una estructura de dos niveles de planificación: eventos y tareas.

- Eventos: están pensados para realizar un proceso pequeño, en el que un componente de bajo nivel avisa a otro componente de alto nivel de

que algo ha sucedido de prioridad alta y de este modo interrumpe las tareas que se están ejecutando en ese momento. Ejemplo: expiración de un temporizador o recepción de una trama.

- Tareas: Las tareas están pensadas para hacer una cantidad mayor de procesado y no son críticas en tiempo. Las tareas son ejecutadas en su totalidad de forma asíncrona siempre que la CPU no tenga ningún evento pendiente de ejecutar. La solicitud de iniciar y finalizar una tarea son funciones separadas.

Con este diseño permitimos que los eventos, que se ejecutan rápidamente, puedan ser realizados inmediatamente, pudiendo interrumpir las tareas que tienen mayor carga computacional en comparación a los eventos, ya que éstos disponen de mayor prioridad frente a las tareas. De esta manera conseguimos un alto rendimiento en las aplicaciones de concurrencia intensiva en que la capacidad de la CPU del dispositivo es utilizada de manera eficiente consumiendo una energía mínima.

Las plataformas que apoyan este sistema operativo, principalmente son Linux (RedHat 9, Ubuntu), Windows 2000, Windows XP utilizando el simulador Linux, Cygwin. Inicialmente TinyOS empezó a funcionar bajo BSD (Unix). Para su funcionamiento bajo Windows (XP en nuestro caso) se necesita la instalación del simulador de plataformas Unix llamada Cygwin. Además la mayoría de las herramientas disponibles son compatibles con lenguajes de programación JAVA, Python y C.

Para la programación de los dispositivos (Motas), se realizará mediante el puerto USB, por lo tanto en el ordenador que utilizaremos para realizar este proyecto será imprescindible contar con al menos un puerto USB para la configuración de los dispositivos.

Se trata de un sistema operativo de distribución libre que fácilmente se puede encontrar en la página Web oficial de TinyOS [2] donde podremos disfrutar de sus últimas versiones y tutoriales online.

TinyOS y NesC, que es el lenguaje de programación utilizado para las motas en este proyecto, están completamente relacionados y más adelante pasaremos a explicar con profundidad dicho lenguaje de programación.

1.2.5. Lenguajes de programación.

La programación de sensores es relativamente compleja, debido a la limitada capacidad de cálculo y la escasa cantidad de recursos. Así como en los

sistemas informáticos tradicionales se encuentran entornos de programación prácticos y eficientes para generar y depurar código, incluso en tiempo de ejecución, en estos microcontroladores todavía no se encuentra disponible.

Podemos encontrar lenguajes como:

- NesC: lenguaje que utilizamos para nuestras motas, como se ha comentado anteriormente, está directamente relacionado con TinyOS y más adelante explicaremos con más profundidad.

- Protothreads [26]: específicamente diseñado para la programación concurrente, provee hilos de dos bytes como base de funcionamiento.

- GalsC [27]: es un lenguaje de programación orientado a tareas, fácil de depurar, y es compatible con los módulos Nesc de TinyOS.

1.2.5.1. Nesc

El lenguaje de programación que utilizaremos en este proyecto es NesC (Network Embedded Systems C), de sintaxis similar a C y está optimizado para las limitaciones de memoria en las redes inalámbricas de sensores.

NesC está orientado a componentes y especialmente está diseñado para programar en redes de sensores bajo el sistema operativo TinyOS.

Una aplicación NesC está formada por uno o más componentes enlazados entre ellos y de esta manera forman un ejecutable. Cada componente proporciona y utiliza interfaces bidireccionales ya que son el único punto de comunicación hacia otros componentes y en ella se puede realizar múltiples instancias hacia una misma, como por ejemplo en el caso que un componente necesite dos temporizadores, en este caso tendrá que hacer dos instancias a la interfaz que proporcione el temporizador. Gracias al uso de interfaces, conseguimos una unión de componentes de manera estática, mejor análisis del programa y un aumento de la eficiencia en lo que a tiempos de ejecución se refiere y en la robustez del diseño.

La implementación de los componentes en Nesc se realiza mediante módulos y configuraciones.

Los módulos proporcionan el código de la aplicación en que se implementan los eventos de una o varias interfaces formadas por una agrupación de comandos y eventos, mediante los cuales conseguimos dar respuesta a algún suceso determinado como puede ser al recibir cierto mensaje desde cierto componente.

Las configuraciones comúnmente conocidas por wiring, son utilizadas para ensamblar otros componentes de manera conjunta en que se conectan las

interfaces utilizadas por los componentes con las interfaces proporcionadas por otros, es decir, se encargarán de unir diferentes componentes en función de sus interfaces ya sea mediante comandos o eventos.

En la figura siguiente (figura 1.4) se mostrará un ejemplo de código en NesC:

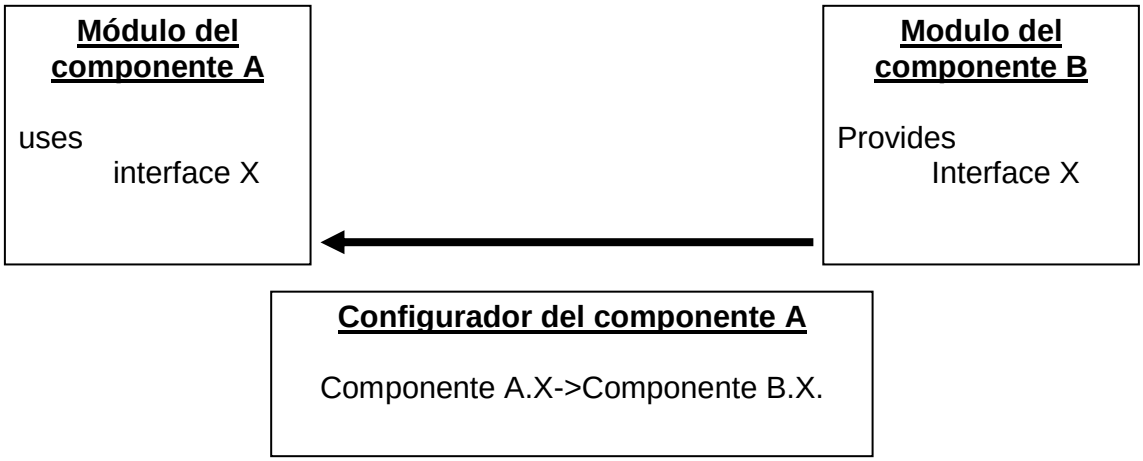

Fig. 1.4. Ejemplo Código NesC.

A partir de la figura anterior (figura 1.4), tal y como indican sus códigos, el componente B proporciona la interfaz X al componente A para que éste la utilice.

1.3. Técnicas de localización.

Una vez hemos formado una red inalámbrica de sensores, nuestro principal objetivo es poder localizar un nodo que este situado en esta red.

Para ello se utilizan diversas técnicas de localización que son divididas en dos grandes grupos:

- Las basadas en distancias: se necesita estimar las distancias entre los nodos o bien los ángulos entre ellos mediante el uso de algoritmos, o bien utilizar dispositivos externos como GPS o coordenadas respecto a un lugar de referencia.

- Las basadas en proximidad o libres de distancias: no necesitan estimar las distancias entre los nodos a diferencia del grupo anterior, se basan en el tiempo de vuelo de la señal de RF o el número de saltos de un mensaje activo por un nodo a la estación base.

1.3.1. Basadas en distancias.

Como se ha mencionado anteriormente, esta técnica se basa en el cálculo de la distancia entre nodos distintos para así realizar una estimación de la posición de un nodo. Antes de explicar las técnicas de localización de este apartado, se explicarán algunos algoritmos de estimación de distancias que tendrán que ser aplicados previamente a su localización:

- Tiempo de Llegada (ToA) [24]: este método, se encargará de medir el tiempo que tardará una señal en desplazarse de un nodo a otro sabiendo de antemano la velocidad de propagación en el medio utilizado. Por ejemplo, una señal que se propaga por el medio a una velocidad de 300 m/s y ha tardado en desplazarse de un nodo a otro 15 ms, podemos estimar que la distancia entre los nodos es de 4,5 m. Un inconveniente de utilizar este algoritmo, es que la velocidad de propagación en el medio, depende de factores externos tales como la temperatura y humedad, y a pesar de ello, no podemos obtener una distancia exacta en caso de la alteración de algunos de estos factores.

- Diferencia de tiempos de llegada (TDoA) [24]: en este algoritmo, se necesita una sincronización en tiempo de manera implícita entre el nodo emisor y receptor. Para ello, se necesita que el nodo emisor emita una señal ultrasónica junto con una señal radio y de esta manera, cuando el receptor reciba la primera señal, se iniciará un temporizador que será detenido cuando llegue la segunda señal. De esta manera, despreciaremos el tiempo de propagación de la primera señal, pero partiendo de la diferencia de tiempos de llegada de las dos señales y de las velocidades de propagación de cada una de las señales podemos obtener una estimación de la distancia. Este algoritmo seria fiable cuando se utilizan velocidades de propagación de órdenes de magnitud diferentes, ya que en caso contrario, si se emplean velocidades de propagación similares, la estimación de la distancia no seria fiable.

- Indicador de fuerza de la señal recibida (RSSI) [29]: en este algoritmo, partiendo de una serie de parámetros tales como la potencia de señal recibida (P_{rcvd}) y transmitida (P_{tx}), la atenuación de señal (c) y el coeficiente de pérdidas de señal (α) podemos hallar la distancia entre dos nodos por medio de la siguiente fórmula:

$$P_{rcvd} = c\frac{P_{tx}}{d^{\alpha}} \Leftrightarrow d = \sqrt[\alpha]{c\frac{P_{tx}}{P_{rcvd}}}$$

  La desventaja de utilizar este algoritmo es que aunque el nodo receptor y emisor se encuentren en una posición fija, los valores de RSSI no mantienen un valor constante debido a fenómenos externos como fast

fading y movimientos del entorno produciendo errores de medición de hasta un 50 %.

Una vez explicados los principales algoritmos de estimación de distancias, se explicará algunas de las técnicas empleadas para la localización de un nodo:

- GPS: funciona mediante una red formada por veinticuatro satélites en órbita con trayectorias sincronizadas para así cubrir la superficie de la tierra [4]. Cuando se desea determinar la posición, el aparato receptor debe captar la información de un mínimo de cuatro satélites pertenecientes a la red y de esta manera recibirá una señal por cada satélite en que se indicará la posición y el reloj de cada uno de ellos, y partiendo de esta información se puede hallar la distancia. En este caso el GPS se conecta a un puerto RS232 del nodo y se envía la información de la posición a la estación base.

- Cricket: esta técnica [5] está pensada para utilizarla en edificios con el fin de localizar tanto elementos móviles como estáticos. Sus ventajas son su privacidad y escalabilidad descentralizada. Sus inconvenientes son la carencia de gestión o monitorización centralizada y sobrecarga computacional debido a la temporización y procesado de señales tanto de radio como ultrasónicas que deben realizar los receptores.

- AHLoS: se trata de una técnica distribuida [6] en que se necesita que una parte de los nodos de la red conozca su posición exacta durante el despliegue mediante receptores GPS o bien mediante una programación de forma manual. Para poder descubrir la posición de un nodo, en primer lugar es necesario que cada nodo haga una estimación de distancia respecto a sus nodos vecinos. Posteriormente, los nodos desconocidos partiendo de la distancia estimada respecto a sus nodos vecinos y de las posiciones de los nodos base cercanos, pasarán a estimar sus posiciones de manera que será un nodo conocido en la red.

- N-hop Multilateration: en esta técnica [8] se realizan mediciones entre los nodos desconocidos y los nodos base conocidos mediante el método sum-dist [9]. Este método consiste en que los nodos base envían un mensaje con su identificador de nodo y un camino de longitud igual a cero. Entonces cada nodo que recibe este mensaje procedente del nodo base sumará la longitud al nodo base y el mensaje será reenviado con la nueva longitud, de manera que si un nodo recibe más de una vez información sobre un mismo nodo, se quedará con el mensaje que le indique menor distancia de manera que todos los nodos tendrán un solo camino y una longitud única a cada uno de los nodos de la red. Posteriormente se pasa a estimar una posición inicial para cada nodo en que se crea una caja limitadora denominada bounding box alrededor de cada nodo base de manera que el círculo de cobertura del nodo pasa a ser un cuadrado con longitud de lado igual al doble del radio de cobertura. De esta forma la posición de un nodo desconocido será estimado como el centro del rectángulo resultante de la intersección de los bounding box de los nodos bases cercanos.

- MDS-MAP: en esta técnica [11], inicialmente el sistema emplea un algoritmo del camino más corto para así poder estimar la distancia entre varios nodos como sea posible. A partir de estas distancias, se construirá una matriz que será utilizada mediante MDS (Escalado Multidimensional), de manera que se construirá un mapa relativo y posteriormente dicho mapa será transformado en un mapa absoluto basado en las posiciones absolutas de los nodos. Para esta técnica existe una alternativa de mejora llamada MDS-MAP(P) [11] de manera que se emplea la técnica MDS para nodos cercanos y finalmente se juntan todos los mapas construidos en un mapa global de la red de manera que en redes bastante extensas puede ser una herramienta útil.

1.3.2. Basadas en proximidad o libres de distancias.

Estas técnicas no necesitan una estimación de la distancia entre los nodos y además tampoco necesitan un hardware adicional al del nodo sensor para realizar la medida entre ellos y eso supone una gran ventaja.

En este grupo se destacan las siguientes técnicas:

- Intersección rectangular [12]: en primer lugar, partimos de que si hay conectividad entre dos nodos, se asume que uno está dentro de un cuadrado centrado en el otro nodo, con lado igual a dos veces el radio de cobertura. La ventaja principal de esta técnica es que el resultado de la intersección de dos o más cuadrados será un rectángulo, de manera que obtendremos una figura geométrica más sencilla que no el resultado de la intersección de varios círculos. Otra ventaja de esta técnica es su ejecución de forma distribuida, es decir que cada nodo adquiere las posiciones de los nodos vecinos de manera que al interseccionarse forman un rectángulo en que la posición estimada del nodo será el centro de dicho rectángulo (ver figura 1.6). El único inconveniente del uso de esta técnica es la necesidad de una conectividad elevada para así obtener una estimación de la posición bastante precisa.

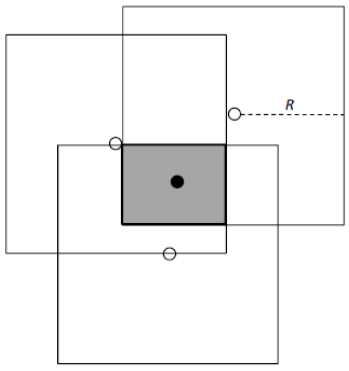

Fig. 1.5. Intersección rectangular.

- Intersección hexagonal: esta técnica [16] es similar a la ya mencionada anteriormente, la intersección rectangular, con la diferencia que el resultado de la intersección de varios hexágonos, da lugar a un polígono irregular donde el nodo estará situado en esa región formada.

- DV-Hop: este método [13] es el que utilizaremos durante el desarrollo de este TFC. Esta técnica esta basada en el mecanismo vector-distancia, en que un nodo difunde un mensaje que contiene las posiciones de los nodos vecinos con el campo hop count (contador de saltos) igual a uno. Cada nodo que reciba un mensaje de cada nodo, mantendrá el contador de cada una de ellas, de manera que los contadores que sean superiores a los que tiene almacenados, serán descartados. En la figura 1.6 podemos ver un ejemplo de su funcionamiento.

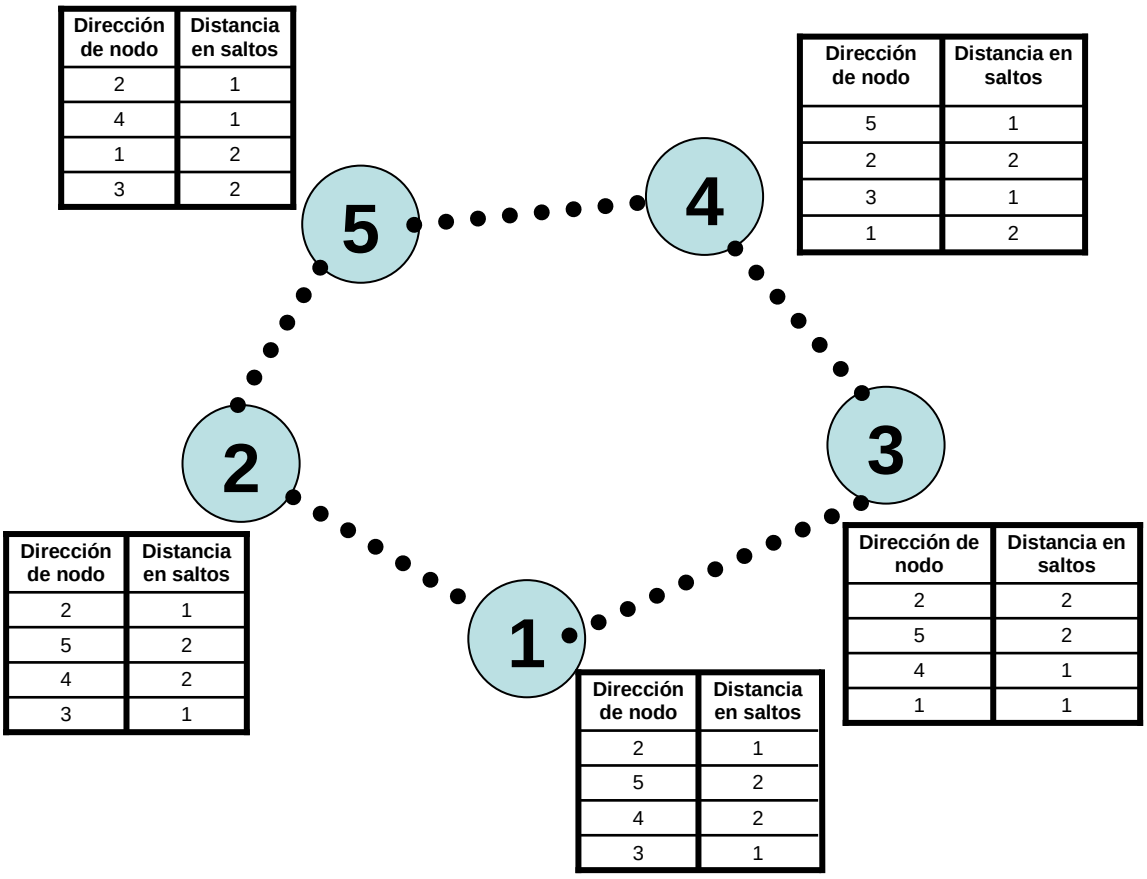

Fig. 1.6. Ejemplo Dv Hop.

En la figura anterior, una vez los nodos han tomado su posición, cada nodo definirá su tabla de encaminamiento teniendo en cuenta la distancia en saltos mínima recibida. En el caso que un nodo cambiase de posición, las tablas de encaminamiento deberían de modificar su contenido.

En nuestro caso hemos utilizado un temporizador que cada vez que expire las tablas de enrutamiento se inicializan y se configuran en función de la nueva información prodecente de los otros nodos.

En esta técnica, podemos destacar como ventajas principales, su facilidad de implementación mediante TinyOS y el hecho de no necesitar ningún hardware adicional al que utilizamos ni ningún algoritmo previo. Es por eso que nos hemos decantado por el uso de esta técnica.

Por contra, como inconveniente principal podemos destacar que mediante está técnica no podemos obtener una distancia exacta a partir del numero de saltos y esto hace que no sea del todo precisa. La distancia estimada queda definida por una cota inferior y superior (mencionado en el capítulo 3) que pueden ser calculadas a partir del radio de cobertura de los nodos y del número de saltos a la que se encuentre situado.

CAPÍTULO 2. DESARROLLO DE LA APLICACIÓN

Como hemos mencionado anteriormente, en este proyecto, hemos utilizado el lenguaje de programación NesC para la implementación de cada una de las aplicaciones en las que los códigos de cada una están adjuntados en el anexo.

Para alcanzar el objetivo propuesto en este proyecto, hemos implementado la técnica de localización DV-Hop (mencionada en el capítulo anterior) al ver que no necesita hardware adicional ni mucha dificultad de implementación.

Esta aplicación a su vez esta formada por 4 aplicaciones: nodo base, nodo sensor, nodo repetidor de primer nivel y nodo repetidor de segundo nivel.

A continuación explicaremos con más detalle el funcionamiento de cada una de estas aplicaciones desarrolladas y sus configuraciones utilizadas.

2.1. Aplicación nodo base

Esta aplicación es ejecutada en el nodo que está conectado al ordenador mediante el puerto USB. Dicho nodo se encargará de recibir las tramas enviadas por los repetidores o sensores que se encuentren dentro del radio de cobertura. Dichas tramas contienen información de nodos pertenecientes a la red, tal como las direcciones y las distancias hacia ellas medida en saltos.

Para efectos prácticos, hemos tenido que disminuir el radio de cobertura de los nodos (70 cm. aproximadamente) para poder realizar las pruebas en el laboratorio con mas comodidad. Para ello hemos tenido que modificar la librería del transceptor CC2420, que mas adelante explicaremos con más detalle.

De este modo, una vez se haya recibido la trama correctamente, podrá ser visualizada a través del ordenador mediante el hiperterminal Docklight [20] (véase en la figura 2.1).

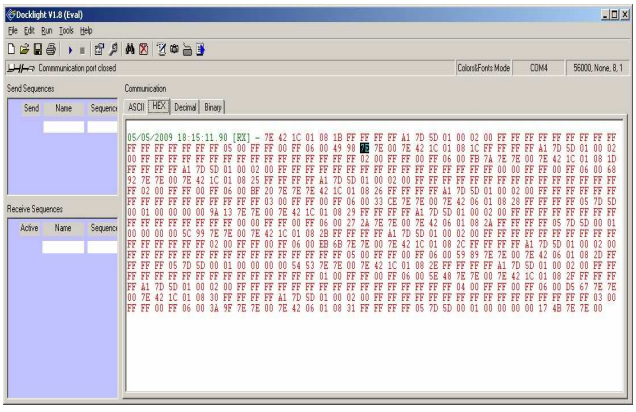

Fig. 2.1 Captura Docklight

2.1.1. Diagrama de flujo

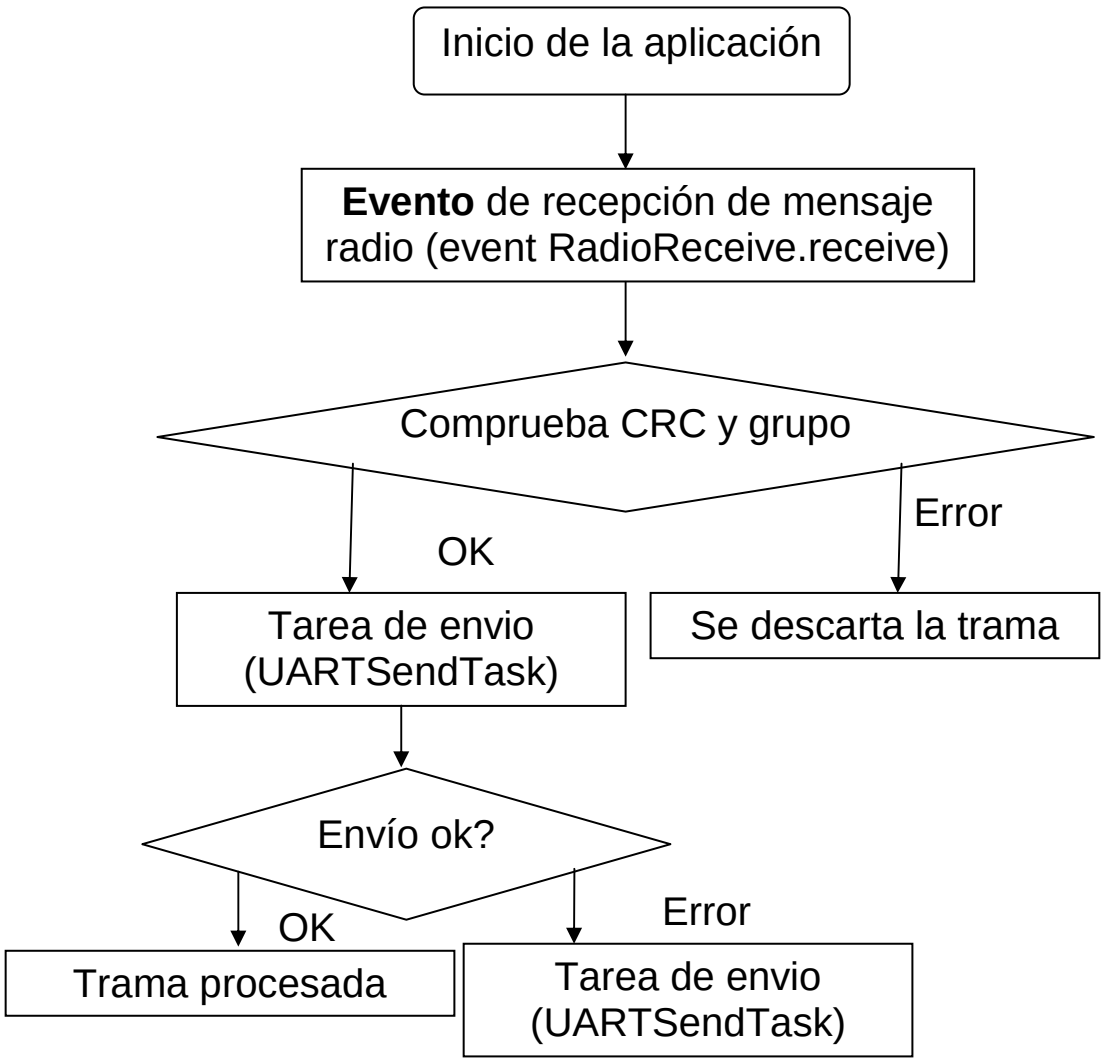

Fig. 2.2 Diagrama de flujo de la aplicación del nodo base

Como podemos observar en el diagrama anterior (figura 2.2), esta aplicación estará a la espera de la recepción de una trama, ya que de se disparará un evento en cuanto se reciba una trama vía radio (evento RadioReceive.receive) y en caso contrario la aplicación estará iniciada pero sin realizar ninguna operación.

Posteriormente una vez recibida la trama y antes de ser procesada deberá comprobar si el CRC (Código de Redundancia Cíclica) es correcto y si dicha trama recibida pertenece al grupo asignado por defecto (8D en nuestro caso). De no ser así la trama será descartada y en caso afirmativo, la trama será enviada al microcontrolador para que sea procesada para ser visualizada a través del ordenador.

Una vez procesada la trama con éxito, el nodo base estará a la espera de recibir la siguiente trama y se volverá a ejecutar el mismo proceso.

2.1.2. Configuración.

En este apartado explicaremos las interfaces principales que utilizamos en esta aplicación y que componentes nos las proporciona.

A continuación mostraremos un fragmento del código del configurador y los alias (figuras 2.3 y 2.4) de las interfaces utilizadas en nuestra aplicación y una explicación detallada de cada una de ellas.

```
components Main, TOSBaseM, RadioCRCPacket as Comm,
FramerM, LedsC;

Main.StdControl -> TOSBaseM;
TOSBaseM.UARTSend -> FramerM;
TOSBaseM.RadioReceive -> Comm;
TOSBaseM.Leds -> LedsC;
```

Fig. 2.3 Código configurador nodo base

```
interface BareSendMsg as UARTSend;
interface ReceiveMsg as RadioReceive;
interface Leds;
```

Fig. 2.4 Alias interfaces

- TOSBaseM.RadioReceive -> Comm.

El componente Comm es un alias del componente RadioCRCPacket y proporciona la interfaz ReceiveMsg (alias RadioReceive) que es utilizada para recibir tramas vía radio mediante el evento Radioreceive.receive.

- TOSBaseM.UARTSend -> FramerM.

El componente Framer proporciona la interfaz BareSendMsg (alias UARTSend) utilizada para el envío de la trama hacia el microcontrolador para ser procesada y poder ser visualizada a través del ordenador mediante el comando UARTSend.send.

- TOSBaseM.Leds -> LedsC

El componente LedsC proporciona la interfaz Leds para encender o apagar los leds de la mota durante el transcurso de nuestra aplicación. Este componente es utilizado en todas las aplicaciones de este de proyecto ya que bastante útil para comprobar el buen funcionamiento de la aplicación.

2.2. Aplicación nodo sensor

Esta aplicación consiste en el envío de tramas de manera broadcast cada cinco segundos por parte de los nodos sensor, de manera que cada uno de ellos enviará su dirección y la distancia hacia él mismo medida en saltos, es decir 0. Estas tramas serán recibidas únicamente por los repetidores o el nodo base mientras se encuentren dentro del radio de cobertura.

Este tipo de nodos, al igual que los repetidores no estarán conectados en ningún momento con el ordenador de manera que para su alimentación necesitaran un par de pilas alcalinas por cada nodo repetidor o sensor.

2.2.1. Diagrama de flujo

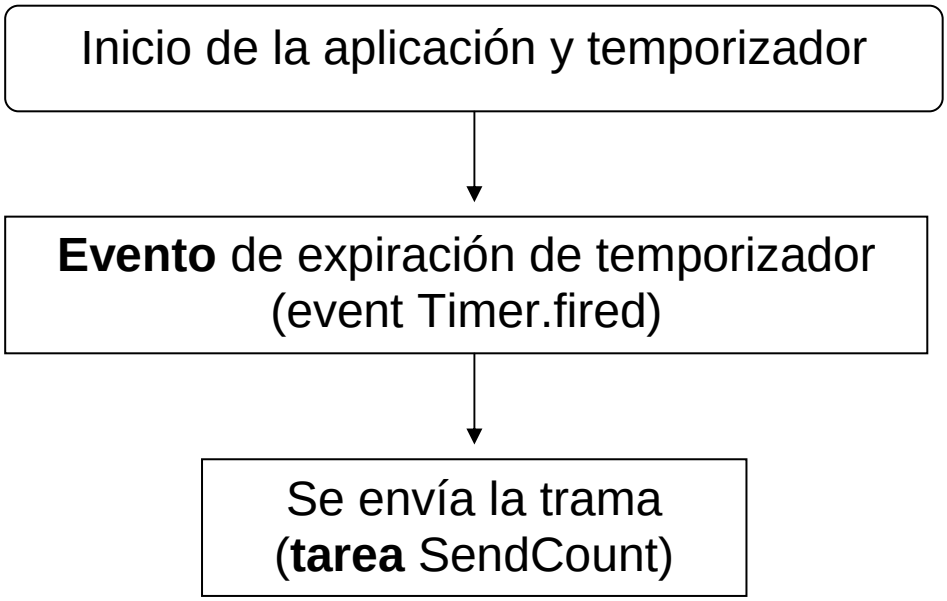

Fig. 2.5 Diagrama de flujo de la aplicación del nodo sensor

Como podemos observar en el diagrama anterior (figura 2.5), esta aplicación estará a la espera de la expiración del temporizador de cinco segundos (evento timer.fired) para el envió de tramas mediante la tarea SendCount.

Como se ha dicho anteriormente, el nodo sensor enviará su dirección y su distancia propia. Esta información ira adjuntada en el campo de datos de la trama que mas adelante explicaremos con más detalle su estructura utilizada.

2.2.2. Configuración.

En esta aplicación las interfaces principales que utilizaremos durante el transcurso de ella, son la Timer y SendMsg (véase en la figura 2.6) y posteriormente vamos a explicar con algo más de detalle.

```
Components Main, RadioDemoM, TimerC, LedsC,
GenericComm;

Main.StdControl -> RadioDemoM;
RadioDemoM.Timer -> TimerC.Timer[unique("Timer")];
RadioDemoM.SendMsg -> GenericComm
```

Fig. 2.6 Código del configurador nodo sensor.

- RadioDemoM.Timer -> TimerC.Timer[unique("Timer")];

Como podemos ver, en el código anterior, el componente TimerC, proporciona a nuestra aplicación la interfaz timer. De esta manera tenemos programado un temporizador iterativo de cinco segundos (TIMER_REPEAT, 5000) y en el momento de expiración se disparará el evento Timer.fired y posteará la tarea pertinente para el envió de la trama.

- RadioDemoM.SendMsg -> GenericComm

El componente GenericComm proporciona a la aplicación del sensor la interfaz SendMsg. Esta interfaz será utilizada para el envío de mensajes a través del comando SendMsg.send que es llamado por la tarea SendCount.

2.3. Aplicación nodo repetidor de primer nivel.

Esta aplicación, se encarga de almacenar en su tabla de enrutamiento las direcciones y distancias de los sensores a partir de los datos procedentes tanto

de los sensores como de los repetidores cercanos y posteriormente, dicha tabla será reenviada hacia el nodo base para poderla visualizar a través del ordenador. Además, se ha implementado un temporizador iterativo de treinta segundos y cada vez que expire dicho temporizador se borrarán todos los datos de la tabla de enrutamiento.

2.3.1. Diagrama de flujo

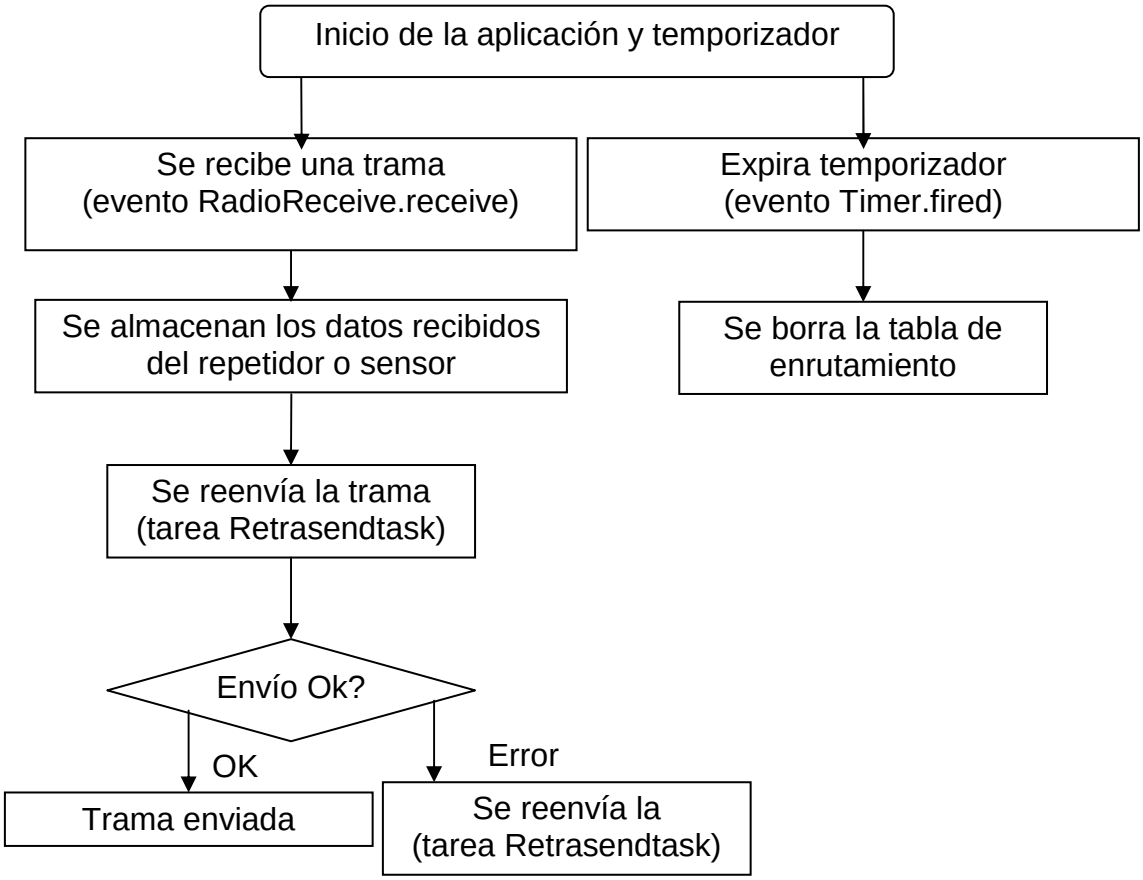

Fig. 2.7 Diagrama de flujo de la aplicación nodo repetidor de primer nivel

Como podemos observar en el diagrama anterior (figura 2.7), esta aplicación estará a la espera de la expiración del temporizador de treinta segundos (Timer.fired) o bien a la recepción de una trama por parte de un repetidor o sensor (RadioReceive.receive).

En caso de expiración de temporizador (evento Timer.fired), la tabla de enrutamiento será borrada, y posteriormente la tabla se ira actualizando en función de la información que vaya recibiendo durante el transcurso de la aplicación

En caso de recepción de una trama, deberemos de tratar la trama de manera diferente en caso de proceda de un sensor o bien un repetidor. Si se trata de

un sensor, únicamente deberemos de extraer su dirección y su distancia y almacenarla en la tabla de enrutamiento. En caso contrario, si se trata de un nodo repetidor, además de su dirección y su distancia propia, también deberemos extraer los datos referentes a los nodos vecinos y añadirlos en la tabla de manera pertinente.

Una vez añadidos los datos recibidos bien sean procedentes de un nodo sensor o repetidor, la tabla de enrutamiento será enviada hacia el nodo base o al resto de repetidores (tarea RetrasendTask) y en caso de envío fallido, se volverá a reenviar la trama hasta que tenga éxito (evento Retrasend.done).

Una vez enviada la trama con éxito, la aplicación estará a la espera de recibir otra trama o a la expiración del temporizador para que se dispare el evento pertinente.

2.3.2. Configuración.

En las siguientes figuras (figuras 2.8 y 2.9), podemos ver los alias de las interfaces principales y que componente nos las proporciona para poder ser utilizadas durante el transcurso de la aplicación del nodo repetidor de primer nivel.

```
interface BareSendMsg as RetraSend;
interface ReceiveMsg as RadioReceive;
interface Timer;
```

Fig. 2.8 Alias utilizados

```
components Main, RepetidorM, RadioCRCPacket as Comm,
FramerM, UART, LedsC, TimerC;

Main.StdControl -> RepetidorM;
RepetidorM.Timer -> TimerC.Timer[unique("Timer")]
RepetidorM.RetraSend -> FramerM;
RepetidorM.RetraSend -> Comm;
RepetidorM.RadioReceive -> Comm;
```

Fig. 2.9 Código del configurador repetidor de primer nivel.

- RepetidorM.Timer -> TimerC.Timer[unique("Timer")];

El componente TimerC proporciona la interfaz timer. De este modo conseguimos programar un temporizador iterativo de treinta segundos (TIMER_REPEAT, 30000) y se disparara el evento de expiración (timer.fired) para borrar los datos de la tabla de enrutamiento.

- RepetidorM.RetraSend -> FramerM;

El componente Framer proporciona la interfaz BareSendMsg (alias RetraSend) utilizada para el envío interno de la trama hacia el microcontrolador del nodo repetidor y así poder ser procesada.

- RepetidorM.RetraSend -> Comm;

El componente RadioCRCPacket (alias Comm) proporciona la interfaz Retrasend para que el repetidor pueda enviar la trama vía radio mediante el comando Retrasend.send llamado desde la tarea de reenvío de trama, RetrasendTask.

- RepetidorM.RadioReceive -> Comm;

El componente Comm proporciona la interfaz Receive para que el repetidor pueda recibir mensajes vía radio mediante el disparo del evento propio de esta interfaz llamado RadioReceive.receive. De esta manera conseguimos almacenar los datos de los sensores o repetidores en nuestra tabla de enrutamiento.

2.4. Aplicación nodo repetidor de segundo nivel.

Esta aplicación, se encarga de almacenar en su tabla de enrutamiento las direcciones y distancias de los sensores a partir de los datos procedentes únicamente de los sensores cercanos que se encuentren dentro del radio de cobertura. Posteriormente, dicha tabla será reenviada hacia el nodo base o hacia los repetidores de primer nivel cercanos. Además, también hemos implementado un temporizador iterativo de treinta segundos y cada vez que expire dicho temporizador se borrarán todos los datos de la tabla de enrutamiento.

2.4.1. Diagrama de flujo

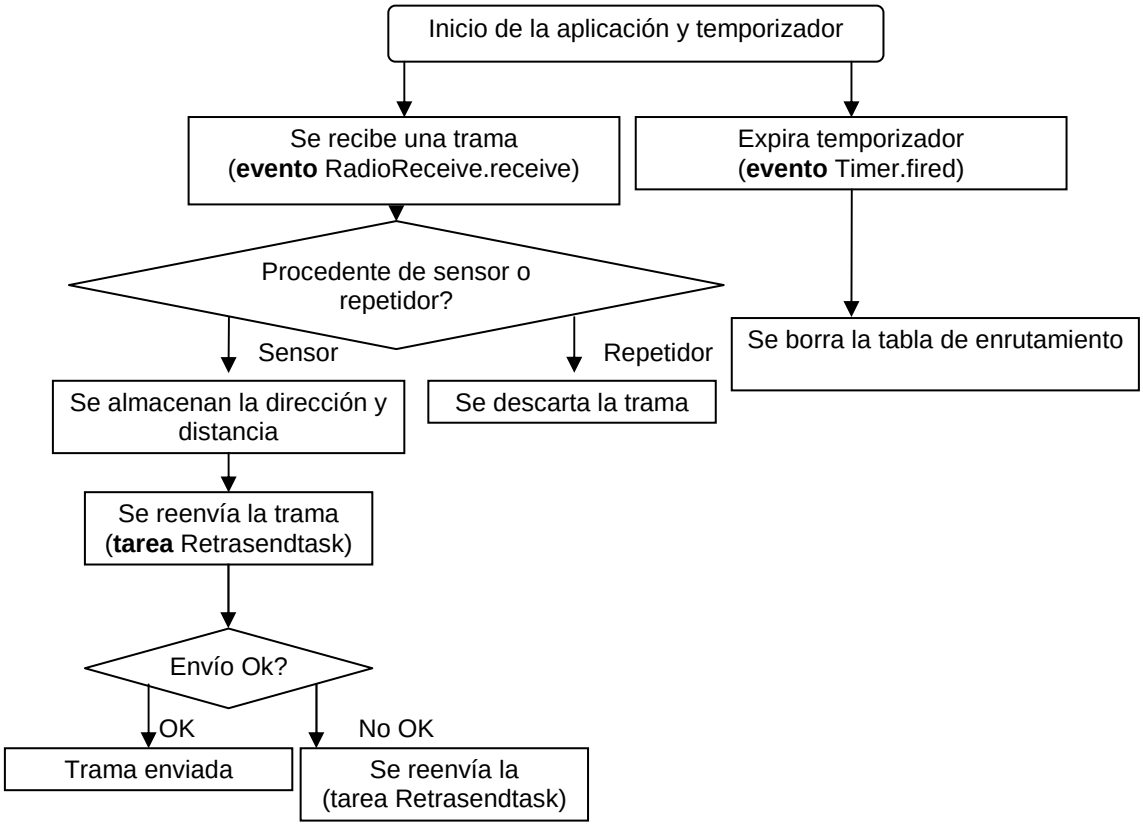

Fig. 2.10 Diagrama de flujo de la aplicación del nodo repetidor de segundo nivel

Como podemos ver en el diagrama anterior (figura 2.10), igual que en el repetidor de primer nivel, estará a la espera de la expiración del temporizador de treinta segundos (Timer.fired) o bien a la recepción de una trama con la diferencia que únicamente procesará las tramas procedentes de los sensores ya que en caso de ser procedentes de un repetidor, éstas serán descartadas.

En caso de expiración de temporizador (evento Timer.fired), la tabla de enrutamiento será borrada, y posteriormente la tabla se ira actualizando en función de la información que vaya recibiendo durante el transcurso de la aplicación

En caso de recepción de una trama, deberemos comprobar si la trama es procedente de un nodo sensor o repetidor. En caso de ser repetidor, la trama será descartada. En caso contrario, al tratarse de un sensor, únicamente deberemos de extraer su dirección y su distancia y almacenarla en la tabla de enrutamiento de manera pertinente.

Una vez añadidos los datos recibidos procedentes del nodo sensor, la tabla de enrutamiento será enviada hacia el nodo base o a los repetidores de primer nivel mediante la tarea (tarea RetrasendTask) y en caso de envío fallido, se volverá a reenviar la trama hasta que tenga éxito (evento Retrasend.done).
Una vez enviada la trama con éxito, la aplicación estará a la espera de recibir otra trama o a la expiración del temporizador para que se dispare el evento pertinente.

2.4.2. Configuración.

En las siguientes figuras (2.11 y 2.12), podemos ver los alias de las interfaces principales y que componente nos las proporciona para poder ser utilizadas durante el transcurso de la aplicación del nodo repetidor de primer nivel.

```
interface BareSendMsg as RetraSend;
interface ReceiveMsg as RadioReceive;
interface Timer;
```

Fig. 2.11 Alias

```
components Main, RepetidorM, RadioCRCPacket as Comm,
FramerM, LedsC, TimerC;

Main.StdControl -> RepetidorM;

RepetidorM.RetraSend -> FramerM;
RepetidorM.RetraSend -> Comm;
RepetidorM.RadioReceive -> Comm;
```

Fig. 2.12 Código del configurador repetidor de segundo nivel.

- RepetidorM.Timer -> TimerC.Timer[unique("Timer")];

El componente TimerC proporciona la interfaz timer. De este modo conseguimos programar un temporizador iterativo de treinta segundos (TIMER_REPEAT, 30000) y se disparara el evento de expiración (timer.fired) para borrar los datos de la tabla de enrutamiento.

- RepetidorM.RetraSend -> FramerM;

El componente Framer proporciona la interfaz BareSendMsg (alias RetraSend) utilizada para el envío interno de la trama hacia el microcontrolador del nodo repetidor y así poder ser procesada.

- RepetidorM.RetraSend -> Comm;

El componente RadioCRCPacket (alias Comm) proporciona la interfaz Retrasend para que el repetidor pueda enviar la trama vía radio mediante el comando Retrasend.send llamado desde la tarea de reenvío de trama, RetrasendTask.

- RepetidorM.RadioReceive -> Comm;

El componente Comm proporciona la interfaz Receive para que el repetidor pueda recibir mensajes vía radio mediante el disparo del evento propio de esta interfaz llamado RadioReceive.receive. De esta manera conseguimos almacenar los datos procedentes de los sensores en nuestra tabla de enrutamiento.

2.5. Estructura de la trama.

Durante el desarrollo de la aplicación de este proyecto, para el envío de información, se utilizan tramas de estructura similar a las del estándar 802.15.4 (ver figura 2.13) donde serán almacenados los datos tales como las tablas de enrutamiento o información de un nodo sensor.

SYNC	Longitud de Trama	FCF	Num. sec	Dirección de destino	Longitud payload	Grupo	Payload	CRC	Fin de trama

Fig. 2.13 Estructura de la trama

- Sync (2 byte): campo de sincronismo e inicio de trama, sus valores suelen ser 7E 42.

- Longitud de trama (1 byte): indica la longitud de la trama recibida.

- FCF (2 bytes): este campo se encarga del control de trama. Se mantiene constante durante una conexión mientras sea enviado por el mismo repetidor o sensor.

- Num. sec (1 byte): es el número de secuencia de la trama, durante una conexión los paquetes recibidos de un mismo nodo, este campo será incrementado en una unidad. De esta manera podemos observar si se pierden paquetes, ya que al fijarnos en dicho campo, si vemos que los

números de secuencia no son consecutivos podemos deducir que se ha producido alguna perdida causada por una colisión de tramas. En nuestra aplicación la perdida de tramas no supone ningún problema en el desarrollo de nuestro algoritmo ya que se realiza un reenvió constante de tramas y las tablas de enrutamiento se actualizan constantemente.

- Dirección de destino (4 bytes): es la dirección hacia donde va dirigida la trama que ha sido enviada. En nuestro caso, como se trata de envíos de manera broadcast, utilizamos la dirección de destino FF FF FF FF.

- Long. Payload (1 Byte): indica la longitud del campo de datos de la trama. En nuestro caso el payload es de 8 bytes por lo tanto tendrá el valor 08.

- Payload: se trata de la información de datos útiles de la trama. Como ya he dicho antes, en nuestro caso es de 8 bytes (ver figura 2.14) ya que debido a la escasez de material, veíamos un tamaño suficiente para almacenar los datos de los sensores (máximo de tres nodos vecinos). En caso de ampliar nuestra red de sensores, deberíamos ampliar el tamaño del payload modificando el parámetro longitud en la librería pertinente teniendo en cuenta que por cada nodo repetidor adicional (1 salto), se requiere ampliar el campo del payload en 2 bytes,

PAYLOAD							
Nodo emisor	**Distancia al nodo emisor**	**Nodo vecino 1**	**Distancia al nodo vecino 1**	**Nodo vecino 2**	**Distancia al nodo vecino 2**	**Nodo vecino 3**	**Distancia al nodo vecino 3**
1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte

Fig. 2.14. Campo Payload

En este proyecto, el campo payload lo utilizaremos para almacenar las tablas de enrutamiento con la dirección y distancia de cada nodo para luego ser reenviadas por la red hasta alcanzar el nodo base que se encargara hacer una estimación de la distancia a la cual se encuentra el nodo que queremos localizar.

- CRC (2 Bytes): según sus siglas significa comprobación de redundancia cíclica que consiste en un control de errores de trama. De esta manera mediante este campo, podemos detectar si se han producido errores durante la transmisión o procesado de trama.

- Fin de trama (2 -3 bytes): indica el final de la trama para así dar paso a la siguiente trama. Normalmente este campo suele tener los valores 7E 7E o 7E 7E 00.

2.6. Memoria utilizada por las aplicaciones

Existen dos tipos de memoria:

- RAM (memoria de acceso aleatorio): se utiliza para almacenar los datos de la aplicación tales como las variables utilizadas en las que se necesita tener acceder rápidamente. Nuestro dispositivo dispone de una memoria RAM de 10 Kb.

- Memoria de programa: conocida por memoria EEPROM (Electrically-Erasable Programmable Read-Only Memory) que es utilizada para almacenar el programa instalado en la mota. El dispositivo que nosotros utilizamos dispone de una memoria flash de 48 kB, y además también dispone de una memoria flash externa de 1 MB pero en este proyecto no ha sido utilizada.

A continuación se muestra una tabla comparativa donde detallaremos la memoria RAM y memoria de programa utilizada por cada una de nuestras aplicaciones.

Aplicaciones	Memoria RAM	Memoria de programa
Nodo Base	3893 bytes	14148 bytes
Nodo Sensor	1120 bytes	26124 bytes
Nodo repetidor 1 nivel	1769 bytes	13996 bytes
Nodo repetidor 2 nivel	1769 bytes	13860 bytes

Taba 2.1 Tabla comparativa de la memoria utilizada por cada aplicaciones

2.7. Preparación de las aplicaciones.

Primeramente, antes de compilar e instalar las aplicaciones en los nodos, para hacer una red más manejable, hemos optado por reducir la potencia de señal de transmisión del transceptor CC2420 [7], y así reducimos el radio de cobertura de las motas para poder trabajar en el laboratorio con más comodidad. Para ello tenemos que abrir la librería CC2424Radio (CC2420Const.h ubicado en la carpeta C:\cygwin\opt\moteiv\tinyos-1.x\tos\lib\CC2420Radio) y modificamos el valor de CC2420_DEF_RFPOWER a 0x01 (ver figura 3.16). Esta potencia, será utilizada durante el transcurso de la aplicación y no será modificada durante el transcurso de ella.

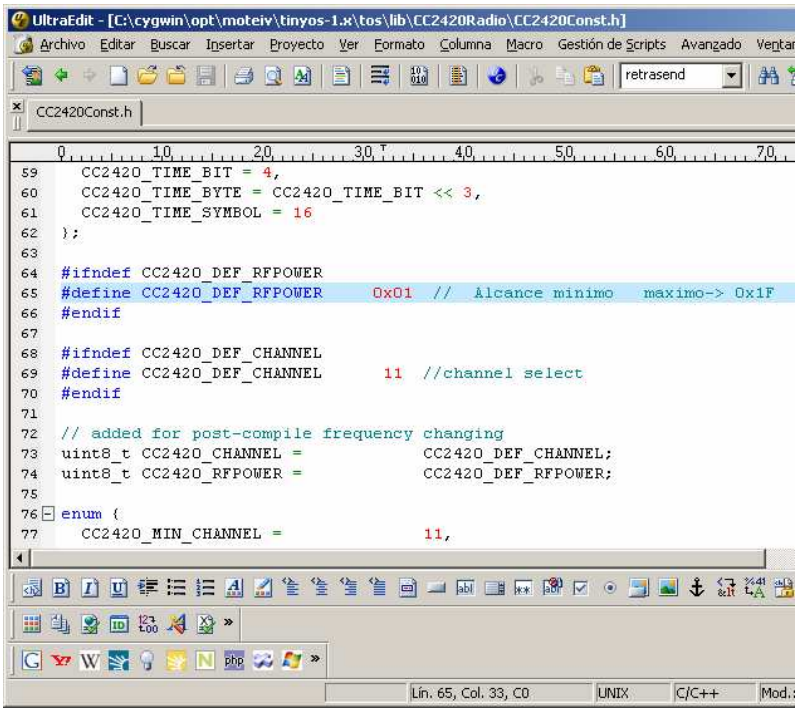

Fig. 2.15 Librería CC2420Const.h

Por defecto el valor del CC2420_DEF_RFPOWER era 0x1F que en decimal equivale a 31 que corresponde al nivel máximo de potencia de salida igual a 0 dbm.

Con el valor actual que hemos asignado la potencia de salida esta por debajo de los -25 dbm y empíricamente supone un radio de cobertura de unos 70 centímetros aproximadamente.

Una vez modificado el fichero de la librería CC2420, tenemos que definir el grupo por defecto en el fichero Makedefaults (ubicado en la carpeta C:\cygwin\opt\tinyos-1.x\tools\make) tal y como indica la figura 3.17. En nuestro caso será el 8d y de este modo, nuestros nodos utilizados, únicamente recibirán las tramas procedentes de los nodos de nuestra red.

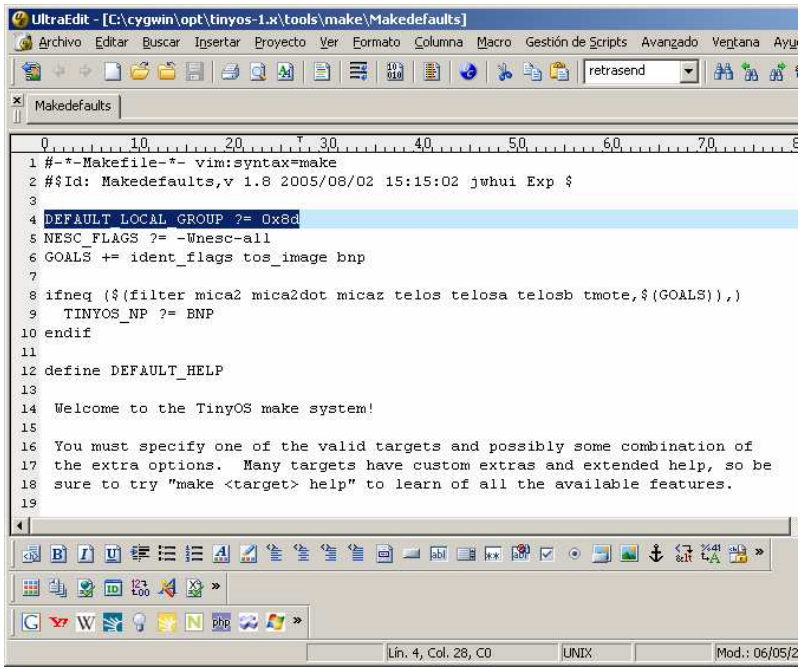

Fig. 2.16 Fichero Makedefault

De esta manera, tanto el nodo base como los nodos repetidores, únicamente recibirán las tramas que tengan asignadas el grupo 8d y en caso contrario las tramas de diferentes grupos no serán procesadas.

Una vez asignados los parámetros de grupo y de potencia de señal, para hacer funcionar las aplicaciones en los nodos, una vez estamos dentro de la carpeta de la aplicación, debemos compilar e instalar las aplicaciones en las motas mediante el uso de comandos introducidos en el simulador de Linux llamado Cygwin.

Para su compilación, empleamos el comando make tmote y de esta manera comprobaremos si existen errores o avisos de compilación y además podemos ver la memoria RAM y ROM que ocupa la aplicaciones que estamos compilando.

Una vez compilada la aplicación con éxito, ejecutaremos el comando make tmote reinstall seguido de la dirección que le queremos asignar a ese nodo tal y como muestra la figura 2.17.

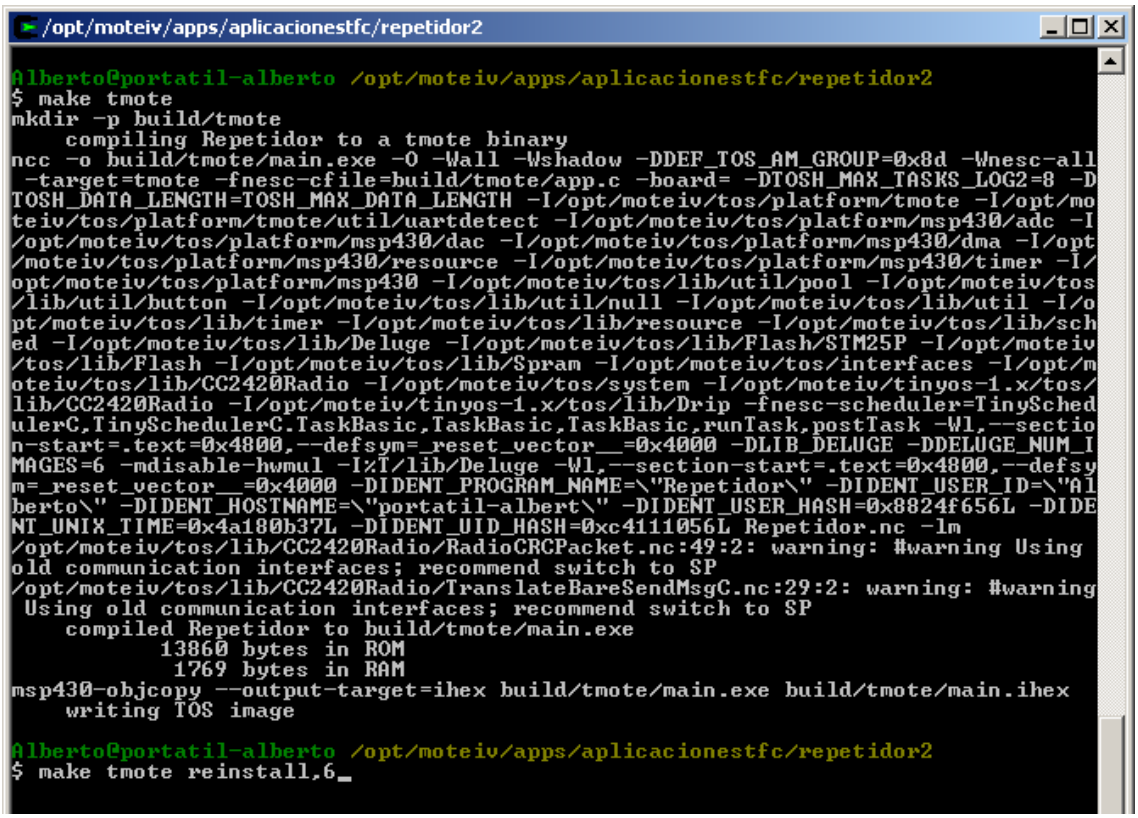

Fig. 2.17 Compilación e instalación de aplicación mediante Cygwin

Para comprobar el funcionamiento de la aplicación hemos utilizado una aplicación llamada Docklight (mencionada en el apartado 2.1), que sirve para capturar el tráfico recibido y así poder visualizar las tramas recibidas por el nodo base.

CAPÍTULO 3. PRUEBAS Y RESULTADOS EXPERIMENTALES

Una vez implementadas las aplicaciones necesarias, para poder comprobar el correcto funcionamiento del algoritmo de localización DV-hop, hemos propuesto dos escenarios diferentes teniendo en cuenta que el radio de cobertura es de aproximadamente de unos 70 cm.:

- Escenario 1: dos nodos sensores, un nodo repetidor.
- Escenario 2: un nodo sensor, dos nodos repetidores.

Partiendo del radio de cobertura (R) y el número de saltos (n) podemos obtener una distancia aproximada (d) comprendida entre los siguientes valores:

$$n*R<d<(n+1)*R$$

Por ejemplo, si en la tabla de encaminamiento tenemos un sensor X situado a 1 salto, su distancia aproximada desde el nodo base, estará comprendida entre 70 y 140 cm.

A continuación mostraremos los escenarios utilizados y las variaciones de las tablas de encaminamientos durante el transcurso de las aplicaciones.

3.1. Escenario 1: Dos nodos sensores y un nodo repetidor.

En este primer escenario disponemos de dos sensores (sensor 3 y sensor 7) conectados vía radio al repetidor 5 y éste conectado al nodo base tal y como muestra la figura 3.1.

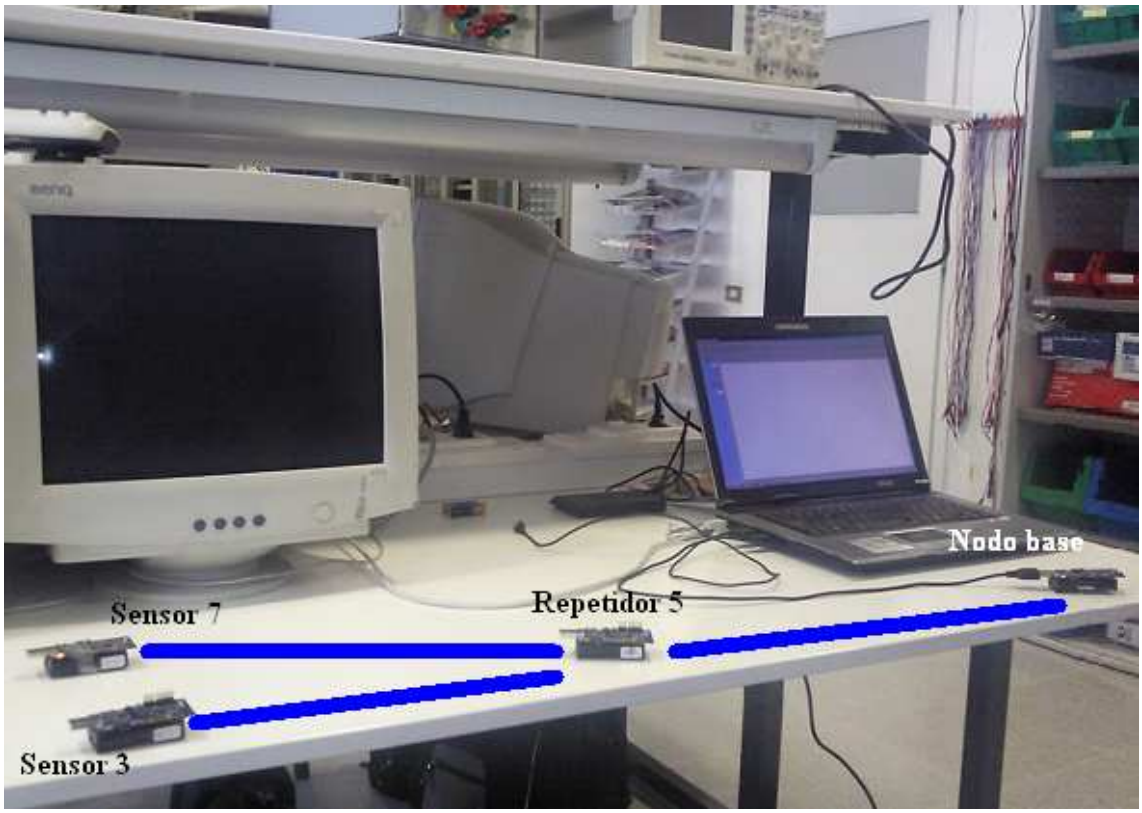

Fig. 3.1 Escenario 1a: 2 sensores y un repetidor.

Una vez situados los nodos de la manera deseada, arrancaremos la aplicación Docklight para capturar el tráfico recibido y observamos que recibimos las siguientes tramas (figura 3.2) procedentes del repetidor 5:

7E 42 08 01 08 14 FF FF FF FF 08 8D 05 00 03 01 07 01 00 00 10 84 7E 7E 00

7E 42 08 01 08 15 FF FF FF FF 08 8D 05 00 03 01 07 01 00 00 EC 2A 7E 7E 00

7E 42 08 01 08 17 FF FF FF FF 08 8D 05 00 03 01 07 01 00 00 35 67 7E 7E 00

Fig. 3.2. Captura 1

A partir de las tramas recibidas por el repetidor 5 y de los datos contenidos en el payload (subrayado en amarillo), las tablas de encaminamientos quedarían confeccionadas de la siguiente manera (tabla 3.1):

Dirección de nodos	Distancia en saltos	Distancia en cm.
5	0	0 - 70
3	1	70 – 140
7	1	70 – 140

Tabla 3.1 Tabla de encaminamiento del repetidor 5.

Posteriormente, apartamos el sensor 7, de manera que el escenario queda reducido a un sensor (sensor 3) conectado a un repetidor (repetidor 5) y éste conectado al nodo base tal y como muestra la figura 3.3.

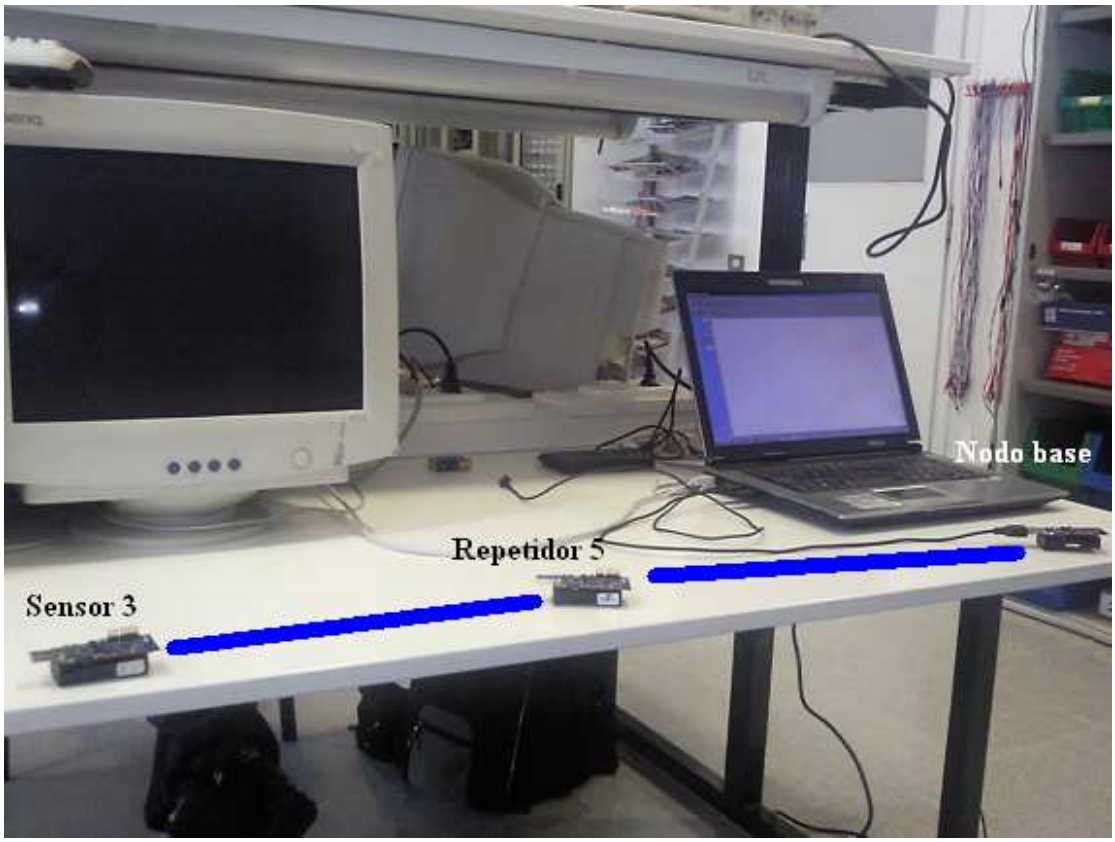

Fig. 3.3 Escenario 1b: 1 sensor y un repetidor

Una vez alejado el sensor 7 de manera que el repetidor 5 se encuentra fuera del radio de alcance del nodo 7, al expirar el temporizador de 30 segundos, la tabla de encaminamiento del repetidor quedará vacía y esperara a recibir una trama procedente del sensor para empezar a rellenarla con su dirección y

distancia correspondiente, de manera que a través del Docklight vemos que recibimos las siguientes tramas (figura 3.4) procedentes del repetidor 5.

7E 42 08 01 08 1D FF FF FF FF 08 8D 05 03 01 00 00 00 00 00 CF 94 7E 7E 00

7E 42 08 01 08 1F FF FF FF FF 08 8D 05 03 01 00 00 00 00 00 33 3A 7E 7E 00

7E 42 08 01 08 21 FF FF FF FF 08 8D 05 00 03 01 00 00 00 00 D2 A2 7E 7E 00

Fig. 3.4 Captura 2.

A partir de las tramas recibidas por el repetidor 5 y de los datos contenidos en el payload (subrayado en amarillo), la tabla de encaminamiento quedaría modificada de la siguiente manera (tabla 3.2):

Dirección de nodos	Distancia en saltos	Distancia en cm.
5	0	0 - 70
3	1	70 – 140
0	0	0

Tabla 3.2 Tabla de encaminamiento del repetidor 5 en saltos.

En las tablas de encaminamiento anteriores (tabla 3.2), vemos que el sensor 7 ha desaparecido de la tabla, por el hecho de que el repetidor 5 este fuera del radio de cobertura como ya se ha comentado anteriormente.

3.2. Escenario 2: Dos nodos repetidores y un nodo sensor.

En este segundo escenario, disponemos de un sensor (sensor 3) conectado al repetidor 6, y éste conectado al repetidor 5 y por ultimo con el nodo base tal y como se indica en la figura 3.5.

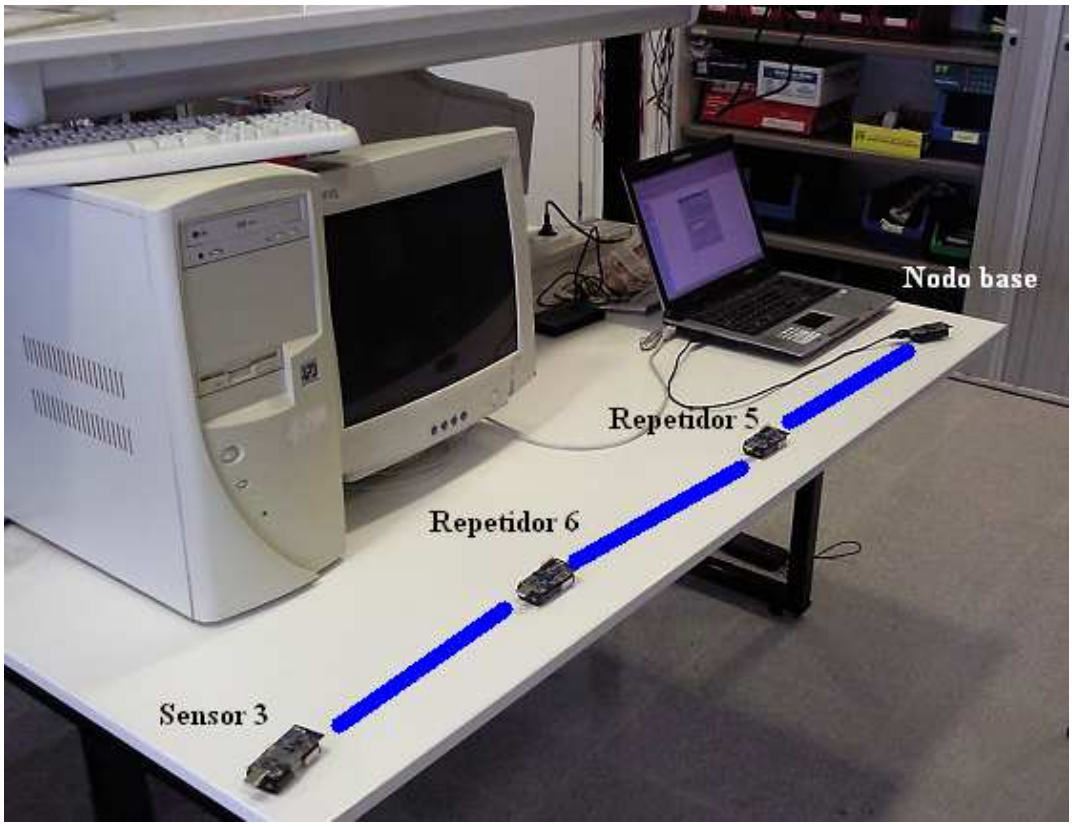

Fig. 3.5 Escenario 2a: un nodo sensor y dos nodos repetidores.

Una vez situados los nodos de la manera deseada, en este segundo escenario, arrancaremos la aplicación Docklight para capturar el tráfico recibido y observamos que recibimos las siguientes tramas (figura 3.6) procedentes del repetidor 5:

7E 42 08 01 08 04 FF FF FF FF 08 8D 05 00 06 01 03 02 00 00 3A 1A 7E 7E 00

7E 42 08 01 08 05 FF FF FF FF 08 8D 05 00 06 01 03 02 00 00 C6 B4 7E 7E 00

7E 42 08 01 08 06 FF FF FF FF 08 8D 05 00 06 01 03 02 00 00 E3 57 7E 7E 00

Fig. 3.6 Captura 3

A partir de las tramas recibidas por el repetidor 5 y de los datos contenidos en el payload (subrayado en amarillo), la tabla de encaminamiento del repetidor 5 quedará de la siguiente manera (tabla 3.3):

Dirección de nodos	Distancia en saltos	Distancia en cm.
5	0	0 - 70
6	1	70 – 140
3	2	140 – 210

Tabla 3.3 Tabla de encaminamiento del repetidor 5 en saltos.

Posteriormente, cogemos el sensor 3 y lo situamos entre el repetidor 5 y 6 de manera que escenario queda modificado como indica la figura 3.7 donde el sensor 3 tendrá conectividad tanto con el repetidor 5 y el repetidor 6

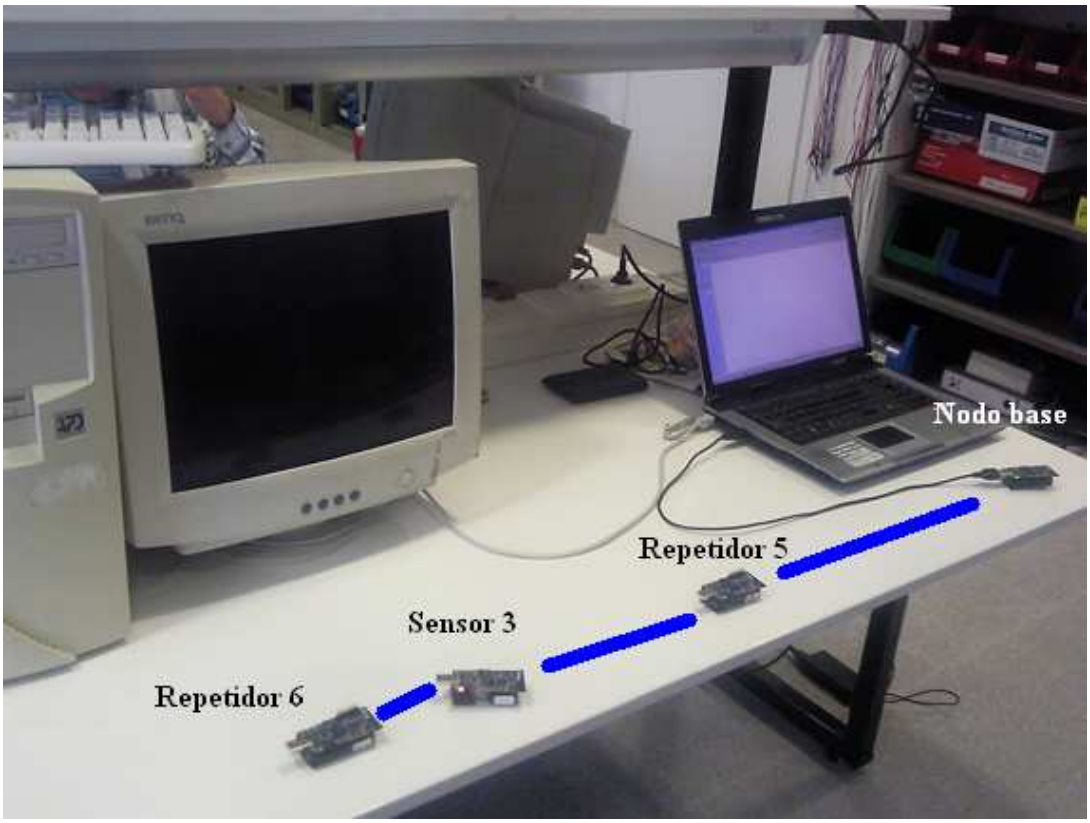

Fig. 3.7 Escenario 2b: un nodo sensor y dos nodos repetidores

Una vez movido el sensor 3, de manera que dentro de su radio de cobertura se encuentran tanto el repetidor 5 y repetidor 6, al expirar el temporizador de 30 segundos, la tabla de encaminamientos del repetidor 5 quedara vacía y esperara a recibir tramas para empezara a rellenarla con la direcciones y

distancias de los nodos según corresponda (ver figura 3.8), y de esta manera, a través del Docklight vemos que recibimos las siguientes tramas procedentes del repetidor 5.

7E 42 08 01 08 18 FF FF FF FF A1 8D 05 00 03 01 06 01 00 00 BD 6C 7E 7E 00

7E 42 08 01 08 19 FF FF FF FF 08 8D 05 00 03 01 06 01 00 00 5E 2A 7E 7E 00

7E 42 08 01 08 1F FF FF FF FF A1 8D 05 00 03 01 06 01 00 00 B2 66 7E 7E

Fig. 3.8. Captura 4.

A partir de las tramas recibidas por el repetidor 5 y de los datos contenidos en el payload (subrayado en amarillo), la tabla encaminamiento quedará de la siguiente manera (ver tabla 3.4):

Dirección de nodos	Distancia en saltos	Distancia en cm.
5	0	0 - 70
3	1	70 – 140
6	1	70 – 140

Tabla 3.4 Tabla de encaminamiento del repetidor 5 en saltos.

Una vez obtenida la tabla de encaminamiento tras haber cambiado el sensor 3 su posición inicial, vemos que dicho sensor se encuentra a un salto del repetidor 5 cuando inicialmente se encontraba a dos.

Conclusiones.

Al finalizar este proyecto, hemos cumplido el objetivo deseado, el cual consistía en la implementación de una técnica de localización de nodos en una red WSN, concretamente nos hemos centrado en la técnica de localización DV-hop.

Se ha profundizado en la programación de redes de sensores con el lenguaje NesC y el uso del sistema operativo TinyOS que permite desarrollar rutinas de alto nivel para controlar nodos sensores, en este caso para localizar nodos sensores en una red WSN. Tanto el sistema operativo como el lenguaje de programación utilizados eran desconocidos para mi, pero finalmente nos decantamos por estos ya que están preparados para la implementación de WSN y por tratarse de un software libre.

Nuestras aplicaciones desarrolladas, inicialmente están preparadas para trabajar bajo una red bastante simple y limitada, debido a la escasez de nodos en el laboratorio. Actualmente utilizamos un payload de 8 bytes, que únicamente nos permite guardar la información de tres nodos vecinos más el nodo transmisor. Pero en el caso de querer ampliar nuestra red, es decir añadir más nodos, deberíamos hacer una simple modificación del parámetro de longitud en la librería pertinente.

Una alternativa de mejora para complementar el algoritmo de localización de nodos DV-HOP utilizado en este proyecto, seria utilizar esta técnica junto con la lectura del indicador de la potencia de la señal recibida (RSSI) por el transceptor para obtener un posicionamiento mas preciso de los nodos. De esta manera, cuando tengamos varios nodos situados a la misma distancia en saltos, en función del parámetro RSSI podremos averiguar que nodo está más cerca (potencia menor) y que nodo se encontrará más alejado (potencia mayor) de la estación base. Para que ésta mejora tenga éxito, se debería utilizar desde un principio TinyOS 2.0 en vez de TinyOS 1.1, que es la versión que hemos utilizado durante el transcurso de este proyecto.

Bibliografía.

[1] 802.15.4
http://ieee802.org/15/pub/TG4.html
http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

[2] Página web oficial de TinyOS
http://www.tinyos.net

[3] Kazem Sohraby, Daniel Minoli &Taiem Znati, Wireless Sensor Networks: Technology, Protocols, and Applications (Pag 27). Editorial Willey Interscience. Canada (2.007).

[4] Hofmann-Wellenhoff, B., Lichtenegger, H., & Collins, J.: Global Positions System: Theory and Practice. Fourth Edition. Springer Verlag (1997).

[5] Priyantha, N. B., Chakraborty, A., & Balakrishnan, H.: The cricket location-support system. (págs. 32-43). New York, USA: Proceedings of the 6th annual international conference on Mobile computing and networking (MobiCom '00). ACM Press (2000).

[6] Savvides, A., Han, C.-C., & Strivastava, M. B.: Dynamic fine-grained localization in ad-hoc networks of sensors. (págs. 166-179). New York, USA: Proceedings of the 7th annual international conference on Mobile computing and networking (MobiCom '01). ACM Press (2001).

[7] 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver

[8] Savvides, A., Park, H., & Srivastava, M. B.: The bits and the flops of the N-hop multilateration primitive for node localization problems. ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02) (2002).

[9] Langendoen, K., & Reijers, N.: Distributed localization in wireless sensor networks: a quantitative comparison. Computer Networks (Elsevier), special issue on Wireless Sensor Networks (November, 2003).

[10] Shang, Y., Ruml, W., Zhang, Y., & Fromherz, M. P.: Localization from Mere Connectivity. MobiHoc (2003).

[11] Shang, Y., & Ruml, W.: Improved MDS-based localization. INFOCOM 2004 (2004).

[12] Simic, S. N., & Sastry, S.: A distributed algorithm for localization in random wireless networks. Discrete Applied Mathematics (2002).

[13] Qiqian Huang and S. Selvakennedy "A Range-Free Localization Algorithm for Wireless Sensor Networks" University of Sydney 2006, Australia.

[14] 802.15.1
http://www.ieee802.org/15/pub/TG1.html]

[15] 802.16 IEEE Standard for Local and Metropolitan area networks, 3 Park Avenue, New York, NY 10016-5997, USA (2.004).

[16] García, E. M., Bermúdez, A., Casado, R., & Quiles, F. J.: Wireless Sensor Network Localization using Hexagonal Intersection. In proceedings of the 1st IFIP International Conference on Wireless Sensor and Actor Networks (WSAN'07) (October, 2007).

[17] R. Barr et al., ‘‘On the Need for System-Level Support for Ad Hoc and Sensor Networks,’’ ACM Operating Systems Review, Vol. 36, No. 2, Apr. 2002, pp. 1–5.

[18] Página oficial de Moteiv
http://www.moteiv.com/

[19] Sentilla. Tmote tools CD v2.0.5 boomerang. TinyOS, drivers, documentation. http://sentilla.com

[20] Página Web oficial de Docklight
http://www.docklight.de

[21] Nut
http://www.proconx.com/xnut/nutos/

[22] Ecos
http://ecos.sourceware.org/

[23] Eyeos
http://eyeos.org/es

[24] Holger Karl and Andreas Willig, Protocols and Architectures for Wireless Sensor Networks (pág. 236). John Wiley & Sons Ltd. England (2.005).

[25] LiteOS
http://www.liteos.net/

[26] Protothread
http://protothread.wiki.sourceforge.net/

[27] GalsC
http://galsc.sourceforge.net/

[28] 802.11
http://www.ieee802.org/11/

[29] Holger Karl and Andreas Willig, Protocols and Architectures for Wireless Sensor Networks (pág. 235). John Wiley & Sons Ltd. England (2.005).

ANEXOS.

Anexo A – Código de la aplicación del nodo base.

- **Archivo de configuración:**

```
configuration TOSBase {
}
implementation {
  components Main, TOSBaseM, RadioCRCPacket as Comm,
FramerM, UART, LedsC;

  Main.StdControl -> TOSBaseM;

  TOSBaseM.UARTControl -> FramerM;
  TOSBaseM.UARTSend -> FramerM;
  TOSBaseM.UARTReceive -> FramerM;
  TOSBaseM.UARTTokenReceive -> FramerM;

  TOSBaseM.RadioControl -> Comm;
  TOSBaseM.RadioSend -> Comm;
  TOSBaseM.RadioReceive -> Comm;

  TOSBaseM.Leds -> LedsC;

  FramerM.ByteControl -> UART;
  FramerM.ByteComm -> UART;
}
```

- **Archivo módulo:**

```
#ifndef TOSBASE_BLINK_ON_DROP
#define TOSBASE_BLINK_ON_DROP
#endif

module TOSBaseM {
  provides interface StdControl;
  uses {
    interface StdControl as UARTControl;
    interface BareSendMsg as UARTSend;
    interface ReceiveMsg as UARTReceive;
    interface TokenReceiveMsg as UARTTokenReceive;

    interface StdControl as RadioControl;
    interface BareSendMsg as RadioSend;
    interface ReceiveMsg as RadioReceive;

    interface Leds;
  }
}
```

```
implementation
{
  enum {
    UART_QUEUE_LEN = 12,
    RADIO_QUEUE_LEN = 12,
  };

  TOS_Msg    uartQueueBufs[UART_QUEUE_LEN];
  uint8_t    uartIn, uartOut, uartCount;
  bool       uartBusy;

  TOS_Msg    radioQueueBufs[RADIO_QUEUE_LEN];
  uint8_t    radioIn, radioOut, radioCount;
  bool       radioBusy;

  task void UARTSendTask();
  task void RadioSendTask();

  void failBlink();
  void dropBlink();
  void  processUartPacket(TOS_MsgPtr  Msg,  bool  wantsAck,
uint8_t Token);

  command result_t StdControl.init() {
    result_t ok1, ok2, ok3;

    uartIn = uartOut = uartCount = 0;
    uartBusy = FALSE;

    radioIn = radioOut = radioCount = 0;
    radioBusy = FALSE;

    ok1 = call UARTControl.init();
    ok2 = call RadioControl.init();
    ok3 = call Leds.init();

    dbg(DBG_BOOT, "TOSBase initialized\n");

    return rcombine3(ok1, ok2, ok3);
  }

  command result_t StdControl.start() {
    result_t ok1, ok2;

    ok1 = call UARTControl.start();
    ok2 = call RadioControl.start();

    return rcombine(ok1, ok2);
  }
```

```
  command result_t StdControl.stop() {
    result_t ok1, ok2;

    ok1 = call UARTControl.stop();
    ok2 = call RadioControl.stop();

    return rcombine(ok1, ok2);
  }

  event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr Msg) {

    dbg(DBG_USR1, "TOSBase received radio packet.\n");

    if ((!Msg->crc) || (Msg->group != 0x8d)) //TOS_AM_GROUP
      return Msg;

    if (uartCount < UART_QUEUE_LEN) {

      memcpy(&uartQueueBufs[uartIn], Msg, sizeof(TOS_Msg));
      uartCount++;

      if( ++uartIn >= UART_QUEUE_LEN ) uartIn = 0;

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

    return Msg;
  }

  task void UARTSendTask() {
    dbg  (DBG_USR1,  "TOSBase  forwarding  Radio  packet  to
UART\n");

    if (uartCount == 0) {

      uartBusy = FALSE;

    } else {

      if  (call  UARTSend.send(&uartQueueBufs[uartOut])  ==
SUCCESS) {
      call Leds.greenToggle();


      } else {
```

```
          failBlink();
          post UARTSendTask();
        }
      }
    }

    event result_t UARTSend.sendDone(TOS_MsgPtr msg, result_t
success) {

      if (!success) {
        failBlink();
      } else {
          uartCount--;
        if( ++uartOut >= UART_QUEUE_LEN ) uartOut = 0;
      }

      post UARTSendTask();

      return SUCCESS;
    }

    event TOS_MsgPtr UARTReceive.receive(TOS_MsgPtr Msg) {
      processUartPacket(Msg, FALSE, 0);
      return Msg;
    }

    event TOS_MsgPtr UARTTokenReceive.receive(TOS_MsgPtr Msg,
uint8_t Token) {
      processUartPacket(Msg, TRUE, Token);
      return Msg;
    }

    void  processUartPacket(TOS_MsgPtr  Msg,  bool  wantsAck,
uint8_t Token) {
      bool reflectToken = FALSE;

      dbg(DBG_USR1, "TOSBase received UART token packet.\n");

      if (radioCount < RADIO_QUEUE_LEN) {
        reflectToken = TRUE;

        memcpy(&radioQueueBufs[radioIn],Msg,sizeof(TOS_Msg));

        radioCount++;

        if( ++radioIn >= RADIO_QUEUE_LEN ) radioIn = 0;

        if (!radioBusy) {
            if (post RadioSendTask()) {
              radioBusy = TRUE;
            }
```

```
      }
    } else {
      dropBlink();
    }

    if (wantsAck && reflectToken) {
      call UARTTokenReceive.ReflectToken(Token);
    }
  }

  task void RadioSendTask() {

    dbg(DBG_USR1, "TOSBase forwarding UART packet to
Radio\n");

    if (radioCount == 0) {

      radioBusy = FALSE;

    } else {

      radioQueueBufs[radioOut].group = TOS_AM_GROUP;

      if (call RadioSend.send(&radioQueueBufs[radioOut]) ==
SUCCESS) {
                call Leds.redToggle();
      } else {
                failBlink();
                post RadioSendTask();
      }
    }
  }

  event result_t RadioSend.sendDone(TOS_MsgPtr msg,
result_t success) {

    if (!success) {
      failBlink();
    } else {
          radioCount--;
      if( ++radioOut >= RADIO_QUEUE_LEN ) radioOut = 0;
    }

    post RadioSendTask();
    return SUCCESS;
  }

  void dropBlink() {
#ifdef TOSBASE_BLINK_ON_DROP
    call Leds.yellowToggle();
#endif
```

```
  }

  void failBlink() {
#ifdef TOSBASE_BLINK_ON_FAIL
    call Leds.yellowToggle();
#endif
  }
}
```

Anexo B – Código de la aplicación del nodo sensor.

- **Archivo de configuración:**

```
#include "Mensaje.h"

configuration RadioDemoC
{
}
implementation
{
  components Main
           , RadioDemoM
           , TimerC
           , LedsC
           , GenericComm
           , DelugeC;

  Main.StdControl -> DelugeC;
  Main.StdControl -> TimerC;
  Main.StdControl -> GenericComm;
  Main.StdControl -> RadioDemoM;

  RadioDemoM.Timer -> TimerC.Timer[unique("Timer")];
  RadioDemoM.Leds -> LedsC;
  RadioDemoM.SendMsg -> GenericComm.SendMsg[AM_Mensaje];
  RadioDemoM.ReceiveMsg                                    ->
GenericComm.ReceiveMsg[AM_Mensaje];
}
```

- **Archivo módulo:**

```
#include "Mensaje.h"

module RadioDemoM
{
  provides interface StdControl;
  uses interface Timer;
  uses interface Leds;
```

```
  uses interface SendMsg;
  uses interface ReceiveMsg;
}
implementation
{

  uint8_t distancia0;

TOS_Msg mrecibido;
bool m_is_sending;

  command result_t StdControl.init()
  {
     distancia0 = 0;
     call Leds.init();
     m_is_sending = FALSE;
     return SUCCESS;
  }

  command result_t StdControl.start()
  {
     call Timer.start(TIMER_REPEAT, 5000);
     call Leds.set(1);
     return SUCCESS;
  }

  command result_t StdControl.stop()
  {
    call Timer.stop();
    return SUCCESS;
  }

  task void sendCount(  )
  {
    if( m_is_sending == FALSE )
    {
      Mensaje_t* body = (Mensaje_t*)mrecibido.data;
      body->distancia = distancia0;
      body->direccion = TOS_LOCAL_ADDRESS;
      body->relleno= 0;
      body->relleno1= 0;
      if(call
SendMsg.send(TOS_BCAST_ADDR,sizeof(Mensaje_t),&mrecibido)
== SUCCESS )
    {
          m_is_sending = TRUE;
      }
    }
  }

  event result_t Timer.fired()
```

```
  {
   post sendCount();
   call Leds.set(2);
   return SUCCESS;
  }

  event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t
result)
  {
    m_is_sending = FALSE;
    call Leds.set(3);
    return SUCCESS;
  }

  event TOS_MsgPtr ReceiveMsg.receive( TOS_MsgPtr Msg )
  {
     Mensaje_t* body = (Mensaje_t*)Msg->data;
     body-> direccion = TOS_LOCAL_ADDRESS;
     body-> distancia = distancia0;

    call Leds.set(4);
    return Msg;
  }
}
```

- **Librería Mensaje.h**

```
#ifndef MENSAJE_H
#define MENSAJE_H

enum
{
  AM_Mensaje = 8,
};

typedef struct Mensaje
{
  nx_uint8_t direccion;
  nx_uint8_t distancia;
  nx_uint32_t relleno;
  nx_uint16_t relleno1;
} Mensaje_t;

#endif
```

Anexo C – Código de la aplicación del nodo repetidor del primer nivel.

- **Archivo de configuración:**

```
configuration Repetidor {
}
implementation {
  components Main, RepetidorM, RadioCRCPacket as Comm,
FramerM, UART, LedsC, TimerC;

  Main.StdControl -> RepetidorM;
  Main.StdControl -> TimerC;
  RepetidorM.Timer -> TimerC.Timer[unique("Timer")];
  RepetidorM.RetraControl -> FramerM;
  RepetidorM.RetraSend -> FramerM;

  RepetidorM.RadioControl -> Comm;
  RepetidorM.RetraSend -> Comm;
  RepetidorM.RadioReceive -> Comm;
  RepetidorM.Leds -> LedsC;

  FramerM.ByteControl -> UART;
  FramerM.ByteComm -> UART;
}
```

- **Archivo módulo:**

```
module RepetidorM {
  provides interface StdControl;
  uses {
    interface StdControl as RetraControl;
    interface BareSendMsg as RetraSend;
    interface StdControl as RadioControl;
    interface ReceiveMsg as RadioReceive;
    interface Leds;
    interface Timer;
  }
}
implementation
{

  enum {

    RADIO_QUEUE_LEN = 2,
    RETRA_QUEUE_LEN = 8,
  };
```

```
  nx_uint8_t direccion;
  nx_uint8_t distancia0;
  nx_uint8_t direccionN1;
  nx_uint8_t distanciaN1;
  nx_uint8_t direccionN2;
  nx_uint8_t distanciaN2;
  nx_uint8_t direccionN3;
  nx_uint8_t distanciaN3;
  nx_uint8_t direccionN4;
  nx_uint8_t distanciaN4;
  nx_uint8_t auxdireccionN1;
  nx_uint8_t auxdistanciaN1;
  nx_uint8_t auxdireccionN2;
  nx_uint8_t auxdistanciaN2;
  nx_uint8_t auxdireccionN3;
  nx_uint8_t auxdistanciaN3;
  nx_uint8_t auxdireccionN4;
  nx_uint8_t auxdistanciaN4;

  uint8_t    radioIn, radioOut, radioCount;
  bool       radioBusy;

  TOS_Msg    retraQueueBufs[RETRA_QUEUE_LEN];
  uint8_t    retraIn, retraOut, retraCount;
  bool       retraBusy;
task void RetraSendTask();
void failBlink();
void dropBlink();
command result_t StdControl.init() {
    result_t ok1, ok2, ok3;

     auxdireccionN1 = auxdistanciaN1=0;
     auxdireccionN2 = auxdistanciaN2=0;
     auxdireccionN3 = auxdistanciaN3=0;
     auxdireccionN4 = auxdistanciaN4=0;

     direccion= TOS_LOCAL_ADDRESS;
     distancia0=0;
     retraIn = retraOut = retraCount = 0;
     retraBusy = FALSE;
     radioIn = radioOut = radioCount = 0;
     radioBusy = FALSE;

     ok1 = call RetraControl.init();
     ok2 = call RadioControl.init();
     ok3 = call Leds.init();
     return rcombine3(ok1, ok2, ok3);
 }
  command result_t StdControl.start() {

    call Timer.start(TIMER_REPEAT, 30000);
```

```
    call RadioControl.start();
    return SUCCESS;
  }
  command result_t StdControl.stop() {
    result_t ok1, ok2;
    ok2 = call RadioControl.stop();
    call Timer.stop();
    return rcombine(ok1, ok2);
  }
event result_t Timer.fired()
{
   auxdireccionN1 = auxdistanciaN1=0;
   auxdireccionN2 = auxdistanciaN2=0;
   auxdireccionN3 = auxdistanciaN3=0;
   return SUCCESS;
}
event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr Msg) {

if (retraCount < RETRA_QUEUE_LEN)
{
   memcpy(&retraQueueBufs[retraIn], Msg, sizeof(TOS_Msg));
   direccionN1=retraQueueBufs[retraIn].data[0];
   distanciaN1=retraQueueBufs[retraIn].data[1]+1;
   direccionN2=0;
   distanciaN2=0;
   direccionN3=0;
   distanciaN3=0;


  if (retraQueueBufs[retraIn].data[2]==0)
  {
   direccionN2=0;
   distanciaN2=0;
   direccionN3=0;
   distanciaN3=0;
  }

  else //caso repetidor
  {
     direccionN2=retraQueueBufs[retraIn].data[2];
     distanciaN2=retraQueueBufs[retraIn].data[3];
     direccionN3=retraQueueBufs[retraIn].data[4];
     distanciaN3=retraQueueBufs[retraIn].data[5];
  }

 retraCount++;
 if( ++retraIn >= RETRA_QUEUE_LEN ) retraIn = 0;
 if (!retraBusy)
 {
    if (post RetraSendTask())
    {
```

```
      retraBusy = TRUE;
    }
  }
}
 return Msg;
}
task void RetraSendTask() {

    if (retraCount == 0) {

      retraBusy = FALSE;

    } else
    {
      if (direccionN2==0)      {
        if (auxdireccionN1 ==0)
        {
           auxdireccionN1=direccionN1;
           auxdistanciaN1=distanciaN1;
        }
        if (auxdireccionN1 ==direccionN1)
        {
           if(auxdistanciaN1>distanciaN1)
               auxdistanciaN1=distanciaN1;
        }
        else
        {
           if (auxdireccionN2 ==0)
           {
              auxdireccionN2=direccionN1;
              auxdistanciaN2=distanciaN1;
           }
           if (auxdireccionN2 ==direccionN1)
           {
              if(auxdistanciaN2>distanciaN1)
                 auxdistanciaN2=distanciaN1;
           }
           else
           {
               if (auxdireccionN3 ==0)
               {
                 auxdireccionN3=direccionN1;
                 auxdistanciaN3=distanciaN1;
               }
               if (auxdireccionN3 ==direccionN1)
               {
                  if(auxdistanciaN3>distanciaN1)
                     auxdistanciaN3=distanciaN1;
               }

           }
```

```
          }
        }
      }

      else{
         auxdireccionN1=direccionN1;
         auxdistanciaN1=distanciaN1;
         auxdireccionN2=direccionN2;
         auxdistanciaN2=distanciaN2;
         auxdireccionN3=direccionN3;
         auxdistanciaN3=distanciaN3;
      }

  retraQueueBufs[retraOut].data[0] = direccion;
  retraQueueBufs[retraOut].data[1] = distancia0;
  retraQueueBufs[retraOut].data[2] = auxdireccionN1;
  retraQueueBufs[retraOut].data[3] = auxdistanciaN1;
  retraQueueBufs[retraOut].data[4] = auxdireccionN2;
  retraQueueBufs[retraOut].data[5] = auxdistanciaN2;
  retraQueueBufs[retraOut].data[6] = auxdireccionN3;
  retraQueueBufs[retraOut].data[7] = auxdistanciaN3;

    if (call RetraSend.send(&retraQueueBufs[retraOut]) ==
SUCCESS) {   }
    else {
        post RetraSendTask();
    }
    }
}

event result_t RetraSend.sendDone(TOS_MsgPtr msg, result_t
success) {
    if (!success)
    {
    }
    else
    {
       retraCount--;
       if( ++retraOut >= RETRA_QUEUE_LEN ) retraOut = 0;
         post RetraSendTask();
    }
        return SUCCESS;
  }
}
```

Anexo D – Código de la aplicación del nodo repetidor de segundo nivel.

- **Archivo de configuración:**

```
configuration Repetidor {
}
implementation {
    components Main, RepetidorM, RadioCRCPacket as Comm,
    FramerM, UART, LedsC, TimerC;

  Main.StdControl -> RepetidorM;
  Main.StdControl -> TimerC;
  RepetidorM.Timer -> TimerC.Timer[unique("Timer")];

  RepetidorM.RetraControl -> FramerM;
  RepetidorM.RetraSend -> FramerM;

  RepetidorM.RadioControl -> Comm;
  RepetidorM.RetraSend -> Comm;
  RepetidorM.RadioReceive -> Comm;

  RepetidorM.Leds -> LedsC;

  FramerM.ByteControl -> UART;
  FramerM.ByteComm -> UART;
}
```

- **Archivo módulo:**

```
module RepetidorM {
  provides interface StdControl;
  uses {
    interface StdControl as RetraControl;
    interface BareSendMsg as RetraSend;
    interface StdControl as RadioControl;
    interface ReceiveMsg as RadioReceive;
    interface Timer;
    interface Leds;
  }
}
implementation
{

enum {
    RADIO_QUEUE_LEN = 2,
    RETRA_QUEUE_LEN = 8,
};
  nx_uint8_t direccion;
```

```
  nx_uint8_t distancia0;
  nx_uint8_t direccionN1;
  nx_uint8_t distanciaN1;
  nx_uint8_t direccionN2;
  nx_uint8_t distanciaN2;
  nx_uint8_t direccionN3;
  nx_uint8_t distanciaN3;
  nx_uint8_t direccionN4;
  nx_uint8_t distanciaN4;
  nx_uint8_t auxdireccionN1;
  nx_uint8_t auxdistanciaN1;
  nx_uint8_t auxdireccionN2;
  nx_uint8_t auxdistanciaN2;
  nx_uint8_t auxdireccionN3;
  nx_uint8_t auxdistanciaN3;
  nx_uint8_t auxdireccionN4;
  nx_uint8_t auxdistanciaN4;

  uint8_t    radioIn, radioOut, radioCount;
  bool       radioBusy;

  TOS_Msg    retraQueueBufs[RETRA_QUEUE_LEN];
  uint8_t    retraIn, retraOut, retraCount;
  bool       retraBusy;
task void RetraSendTask();
command result_t StdControl.init() {
    result_t ok1, ok2, ok3;

    auxdireccionN1 = auxdistanciaN1=0;
    auxdireccionN2 = auxdistanciaN2=0;
    auxdireccionN3 = auxdistanciaN3=0;
    direccion= TOS_LOCAL_ADDRESS;
    distancia0=0;

    retraIn = retraOut = retraCount = 0;
    retraBusy = FALSE;

    radioIn = radioOut = radioCount = 0;
    radioBusy = FALSE;

    ok1 = call RetraControl.init();
    ok2 = call RadioControl.init();
    ok3 = call Leds.init();
    return rcombine3(ok1, ok2, ok3);
  }
  command result_t StdControl.start() {

     call Timer.start(TIMER_REPEAT, 30000);
     call RadioControl.start();
     return SUCCESS;
  }  command result_t StdControl.stop() {
```

```
    result_t ok1, ok2;
    ok2 = call RadioControl.stop();
    call Timer.stop();
    return rcombine(ok1, ok2);
  }
event result_t Timer.fired()
{

   auxdireccionN1 = auxdistanciaN1=0;
   auxdireccionN2 = auxdistanciaN2=0;
   auxdireccionN3 = auxdistanciaN3=0;
  return SUCCESS;
}
event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr Msg) {

  if (retraCount < RETRA_QUEUE_LEN)
  {
    memcpy(&retraQueueBufs[retraIn], Msg, sizeof(TOS_Msg));

    if (retraQueueBufs[retraIn].data[2]==0){
       direccionN1=retraQueueBufs[retraIn].data[0];
       distanciaN1=retraQueueBufs[retraIn].data[1]+1;
       direccionN2=0;
       distanciaN2=0;
       direccionN3=0;
       distanciaN3=0;

        retraCount++;
        if( ++retraIn >= RETRA_QUEUE_LEN ) retraIn = 0;
          if (!retraBusy)
          {
              if (post RetraSendTask())
              {
                 retraBusy = TRUE;
              }
          }
     }
    }
return Msg;
}

task void RetraSendTask() {

    if (retraCount == 0) {
      retraBusy = FALSE;
    }
    else
    {
        if (auxdireccionN1 ==0)
        {
           auxdireccionN1=direccionN1;
```

```
            auxdistanciaN1=distanciaN1;
         }
         if (auxdireccionN1 ==direccionN1)
         {
            if(auxdistanciaN1>distanciaN1)
                auxdistanciaN1=distanciaN1;
         }
         else
         {
            if (auxdireccionN2 ==0)
            {
               auxdireccionN2=direccionN1;
               auxdistanciaN2=distanciaN1;
            }
            if (auxdireccionN2 ==direccionN1)
            {
               if(auxdistanciaN2>distanciaN1)
                  auxdistanciaN2=distanciaN1;
            }
            else
            {
                if (auxdireccionN3 ==0)
                {
                  auxdireccionN3=direccionN1;
                  auxdistanciaN3=distanciaN1;
                }
                if (auxdireccionN3 ==direccionN1)
                {
                   if(auxdistanciaN3>distanciaN1)
                      auxdistanciaN3=distanciaN1;
                }

            }

          }


retraQueueBufs[retraOut].data[0] = direccion;
retraQueueBufs[retraOut].data[1] = distancia0;
retraQueueBufs[retraOut].data[2] = auxdireccionN1;
retraQueueBufs[retraOut].data[3] = auxdistanciaN1;
retraQueueBufs[retraOut].data[4] = auxdireccionN2;
retraQueueBufs[retraOut].data[5] = auxdistanciaN2;
retraQueueBufs[retraOut].data[6] = auxdireccionN3;
retraQueueBufs[retraOut].data[7] = auxdistanciaN3;

if   (call   RetraSend.send(&retraQueueBufs[retraOut])   ==
SUCCESS){ }
else{
   post RetraSendTask();
}
```

```
}
}

event result_t RetraSend.sendDone(TOS_MsgPtr msg, result_t
success)
{

   if (!success) { }
   else
   {
       retraCount--;
       if( ++retraOut >= RETRA_QUEUE_LEN ) retraOut = 0;
   }

  return SUCCESS;
}
```

Anexo E – Instalación TinyOS.

Para la instalación de TinyOS en Windows, hemos utilizado el CD Tmote Tools v.2.0.5.

En la instalación de este sistema operativo, se instalará la aplicación Cygwin y la plataforma Java necesaria para su funcionamiento.

Su instalación es bastante sencilla e intuitiva y están todos sus pasos guiados en el manual de instalación [19].

Anexo F – Descripción detallada de la mota Tmote Sky

A continuación se muestra una imagen detallada donde podemos ver los diferentes componentes que forman este dispositivo y su diagrama de bloques que lo forman. Esta imagen ha sido extraída de la hoja de especificaciones que podemos obtener a través de la página web oficial de Moteiv [18].

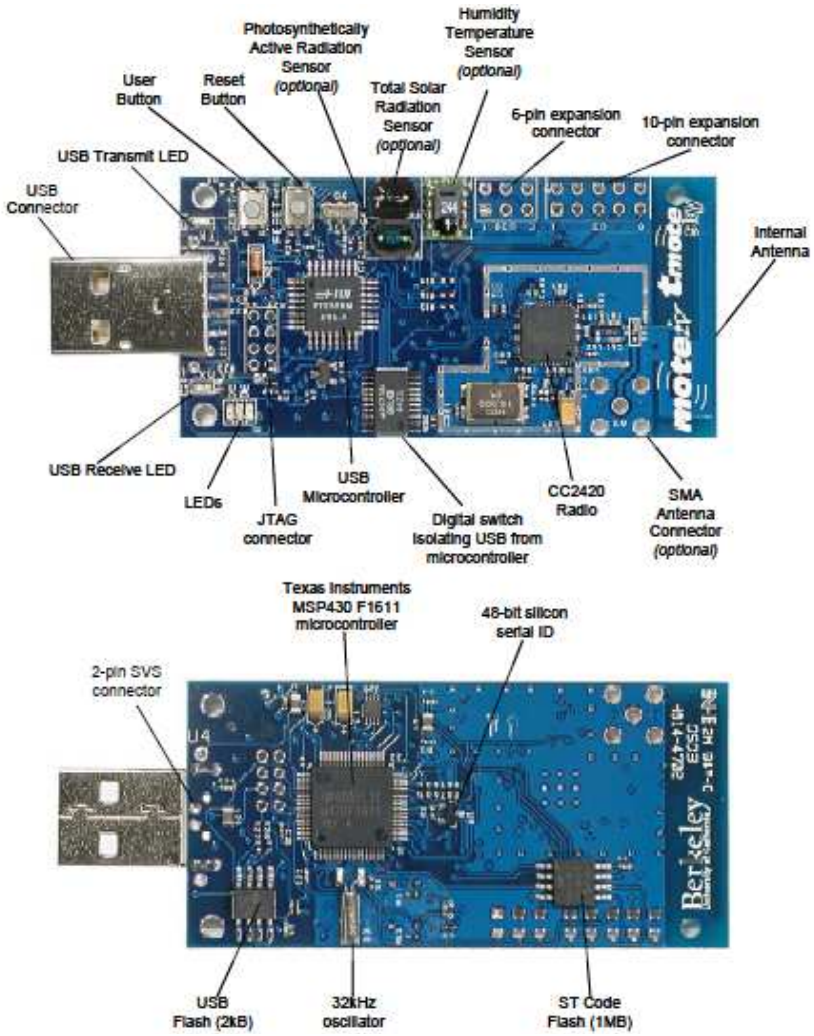

Fig. F1 Parte delantera y trasera del dispositivo Tmote Sky

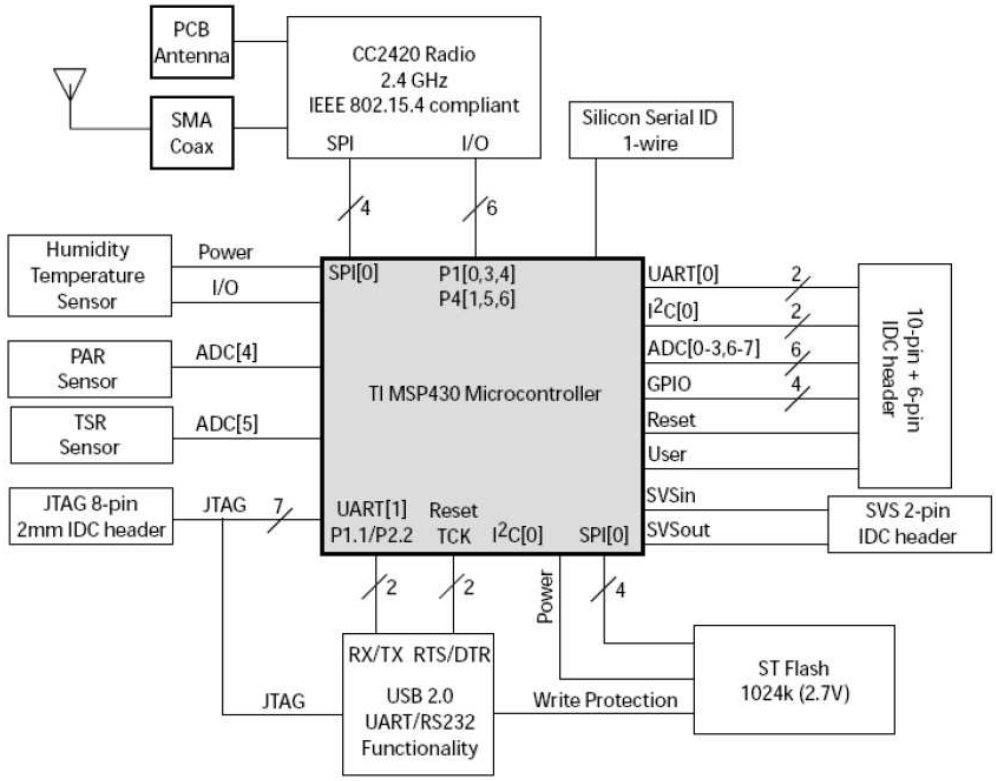

Fig. F.2 Diagrama de bloques de la mota Tmote Sky