TRABAJO FIN DE CARRERA

TÍTULO DEL TFC: Implementación y análisis de mecanismos de transmisión cooperativa en sistemas de comunicaciones inalámbricos

TITULACIÓN: Ingeniería Técnica de Telecomunicaciones, especialidad Telemática

AUTORS: Sonia Pardo Mas
Jordi Pérez Rueda

DIRECTOR: Luís Alonso Zárate

DATA: 9 de Octubre de 2009

Título: Implementación y análisis de mecanismos de transmisión cooperativa en sistemas de comunicaciones inalámbricos

Autor: Sonia Pardo Mas
Jordi Pérez Rueda

Director: Luís Alonso Zárate

Fecha: 9 de Octubre de 2009

Resumen

Los sistemas de comunicaciones inalámbricos disponen de alcance máximo o rango de cobertura. Alrededor de la zona límite los paquetes pueden llegar al destino con una alta probabilidad de error. La utilización de técnicas de transmisiones cooperativas en este tipo de escenarios es innovadora.

Este proyecto ha consistido en el estudio e implementación de protocolos de transmisión cooperativa en redes de comunicaciones inalámbricas, introduciendo el uso de nodos intermedios y el mecanismo de '*Majority Voting'*. Esta técnica se encarga de reconstruir un paquete erróneo a partir de la información reenviada por nodos cooperantes.

Por una parte, se han analizado las mejoras que ofrece el uso de esta técnica en escenarios donde la probabilidad que un paquete contenga errores en su contenido sea elevada.

Para estudiar el comportamiento del sistema se han desarrollado tres escenarios utilizando una red de motas CrossBow tipo TelosB, las cuales utilizan la tecnología Zigbee basada en el protocolo IEEE 802.14.5 y el sistema operativo TinyOS. Estos dispositivos utilizan el lenguaje de programación denominado NesC, un dialecto del lenguaje de programación C optimizado para las limitaciones de memoria de las redes de sensores.

Por otra parte, para poder recibir los datos y analizarlos en un ordenador se ha creado una interfaz gráfica programada en Java, puesto que TinyOS proporciona las herramientas Java necesarias para interpretar la información proveniente de las motas a través de un puerto USB.

Nuestro estudio se ha centrado en evaluar el sistema utilizando la técnica de '*Majority Voting'* en función del número de nodos cooperantes y medir la mejora en las comunicaciones que puede obtenerse. Analizando los resultados obtenidos se ha demostrado la importancia de utilizar las técnicas de cooperación en situaciones donde la probabilidad de error del contenido del paquete sea alta, ya sea por causa de posibles interferencias o atenuación de la señal.

Title: Implementation and analysis of cooperative transmission mechanisms in wireless communication systems

Author: Sonia Pardo Mas
Jordi Pérez Rueda

Director: Luís Alonso Zárate

Date: October, 9th 2009

Overview

Wireless communication systems offer maximum reach or coverage range. Packets reaching destinations situated at the furthest area can present a high error probability. Using cooperative transmission techniques in this sort of scenarios represents a significant innovation.

The purpose of the present project is to study and implement cooperative transmission protocols in wireless communication networks, as well as using intermediate nodes and the "Majority Voting" mechanism. This latter technique is used to rebuild packets with errors using the data forwarded by cooperative nodes.

On the one hand, we analysed the improvements this technique offers in scenarios where there is a high probability of packets having errors in its content.

In order to study system performance, we developed three scenarios using a dust network of CrossBow's TelosB motes, which use IEEE 802.14.5 protocol-based and TinyOS operating system-based Zigbee technology. These devices use a programming language called NesC, a dialect of C programming language optimized for the memory constraints of sensor networks.

On the other hand and taking into account that TinyOS offers the Java tools needed to interpret data sent by motes through a USB port, we created a Java-based graphical interface to be able to receive and analyse data in a computer.

Our study focused on evaluating the system using the "Majority Voting" technique according to the number of cooperative nodes, as well as on measuring the communication improvement gained. The analysis of the results demonstrates how important is to use cooperative techniques in scenarios where error probability in data packets is high, either because of interferences or because of signal fading.

ÍNDICE

INTRODUCCIÓN

Una de las grandes revoluciones tecnológicas del último siglo ha sido la comunicación inalámbrica, con la cual podemos comunicar ordenadores u otros dispositivos sin la necesidad de cables. Esto conlleva al desarrollo de diferentes estándares para transmitir y recibir información con tal de cubrir el mayor número de necesidades.

Actualmente, con esta finalidad, han aparecido unos dispositivos, pequeños, baratos, de bajo consumo, capaces de transmitir información de forma inalámbrica y procesarla localmente. A estos elementos se les conoce por el nombre de Motas. Estas motas utilizan la tecnología radio IEEE 802.15.4, la cual es la base sobre la que se define la especificación Zigbee y el sistema operativo tinyOS.

En todo medio radio existe una zona de cobertura, donde la mayoría de los paquetes enviados se reciben correctamente. Una vez fuera de esta zona de cobertura los datos no llegan a su destino. Justo entre estas dos zonas se encuentra un umbral en el cual los paquetes que se reciben en destino pueden contener errores. Estudiar el comportamiento de un sistema trabajando en este umbral no es sencillo, ya que puede variar mucho, y es fácil pasar a la zona donde ya no hay cobertura.

Nuestro estudio se basa en mejorar el número de transmisiones correctas dentro de éste umbral mediante el sistema de cooperatividad entre sensores Motas (transmisiones cooperativas). Cada mota cooperante tiene una función. Una de ellas será la mota emisora, que será la encargada de enviar tantos paquetes como configuremos durante un tiempo determinado. Otra mota será la receptora encargada de recibir los paquetes y enviarlos a un PC de control. Y por último tenemos los nodos cooperantes, que retransmitirán el paquete a la estación base (nodo receptor) cuando sea necesario.

En el sistema cooperativo se ha aplicado el método conocido como *Majority Voting*, el cual se basa en la reconstrucción de paquetes con la ayuda de nodos cooperantes. Es decir, no sólo tenemos un emisor y un receptor, sino que también tenemos unos nodos intermedios, más cercanos al receptor, que nos ayudaran y nos retransmitirán el paquete en caso de fallo. La mota receptora será la encargada de reconstruir el paquete mediante los paquetes reenviados por los nodos intermedios.

Para la realización de dicho proyecto se ha partido de un trabajo anterior, el cual utilizaba dos tipos de aplicaciones. Una primera en lenguaje JAVA dónde se configura la potencia y el número de paquetes que debe enviar el emisor, y nos informa en el PC de aquellos que han sido recibidos; y otra en lenguaje NesC que se ejecuta en cada una de las Motas.

Por tanto, nuestro trabajo está estructurado en las siguientes fases:

- Estudio de la plataforma tinyOs y de su lenguaje de programación, NesC.
- Estudio de la aplicación desarrollada en el proyecto anterior.
- Estudio de las bases teóricas necesarias para poder realizar dicho proyecto.
- Planteamiento y programación de las técnicas de transmisión cooperativa en las motas telosb.
- Realización de las pruebas en los distintos escenarios con tal de poder estudiar el comportamiento de los mismos.
- Análisis de los resultados obtenidos en las diversas pruebas.
- Escrito de la memoria, donde se pueden diferenciar los siguientes bloques:
    - Introducción a las bases teóricas
    - Explicación de forma detallada de como se ha ampliado la aplicación JAVA y el código NesC, así como la creación de nuevas aplicaciones en las motas para poder realizar un estudio del sistema.
    - Exposición de los resultados obtenidos en el estudio desarrollado en el capítulo anterior.
    - Conclusiones finales.

CAPÍTULO 1. PRINCIPIOS TEÓRICOS

1.1. Comunicaciones Inalámbricas

La comunicación inalámbrica es un tipo de comunicación que utiliza la modulación de ondas electromagnéticas, las cuales se propagan a través del espacio sin la necesidad de un medio físico que comunique los dos extremos de la transmisión.

El inicio de esta tecnología se produjo en el año 1867, cuando James Clerk Maxwell realizo unos estudios en los que desarrolló la teoría electromagnética dando un gran paso ante la posibilidad de crear este tipo de ondas y propagarlas por el espacio. Más adelante, en el año 1887, el alemán Hertz realizó los experimentos necesarios para confirmar esta teoría donde obtuvo la primera transmisión sin hilos, pasándose a denominarse ondas hertzianas.

Actualmente este tipo de tecnologías está en alza y han avanzado mucho desde ese entonces, creando una multitud de estándares, como pueden ser Wifi, Bluetooth o Zigbee.

1.1.1. Ventajas e inconvenientes

Este tipo de tecnología presenta grandes ventajas, pero obviamente también presenta algunos inconvenientes.

Sus principales ventajas son:

- Movilidad de los usuarios, ya que al no disponer de un cable físico, ésta es mucho más posible.
- Gran flexibilidad para instalaciones con un cableado y mantenimiento difícil.
- Rápida instalación de la red
- El costo de mantenimiento comparado con una red convencional es menor.

Como principales inconvenientes cabe destacar:

- Las interferencias, ya que tienen una tasa de error y pérdida de paquete elevada.
- La seguridad, ya que pueden ser interceptadas y saboteadas.
- Dispone de un alcance limitado a un área determinada.

1.1.2. Clasificación

Las comunicaciones inalámbricas se clasifican en diversas categorías, en función del área geográfica que pueden alcanzar.

- **WAN** (Wide Area Network): Es un tipo de red capaz de cubrir desde decenas hasta miles de kilómetros, dando servicio a un país o continente. Unas de sus principales tecnologías son UMTS, GPRS y GSM.
- **MAN** (Metropilitan Area Network): Este tipo de tecnología esta pensada para abarcar un área geográfica extensa, como puede ser un área metropolitana de una ciudad.
- **LAN** (Local Area Network): Su extensión, en este caso, esta limitada físicamente a un edificio de 200m. En el caso que utilicemos repetidores, el alcance puede llegar incluso a 1km. Su aplicación más extendida es la interconexión de ordenadores personales en una oficina. Su principal tecnología es Wifi.
- **PAN** (Personal Area Network): En este caso su alcance esta limitado a 30m. Entre las diferentes soluciones de redes PAN, se encuentran Bluetooth y Zigbee.

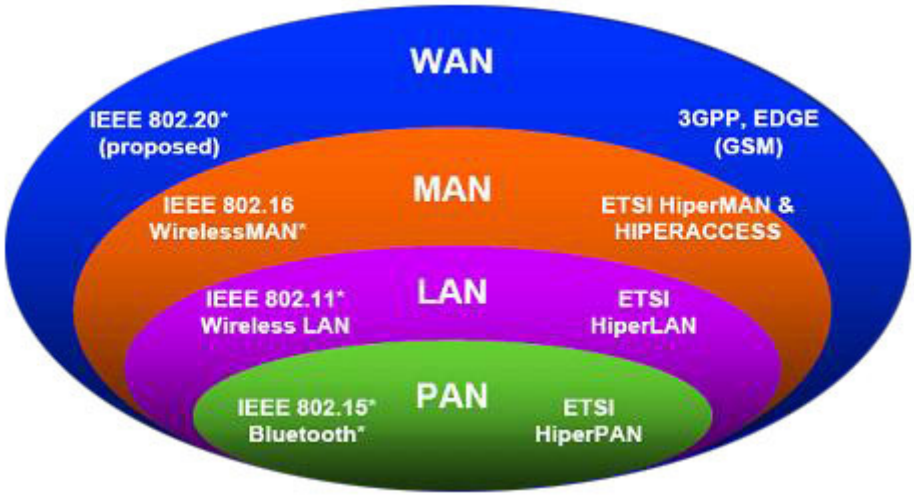


Fig. 1 Clasificación redes de comunicaciones inalámbricas

1.1.3. Canal radio

Como hemos comentado en el apartado 1.1, las comunicaciones inalámbricas propagan su información a través de un canal radio, por el cual la potencia de la señal recibida va disminuyendo en función de la distancia.

En el caso de tratarse de una propagación en el vacío (caso ideal) la potencia de la señal recibida es d^{-2} siendo d la distancia entre el emisor y el receptor.

En cambio, en el caso de que el medio de propagación sea otro la potencia de la señal recibida disminuye en función de d^{-k} donde k es un número mayor a 2 y varía en función del tipo de medio.

Además de este motivo, la calidad de la señal recibida en un medio radio también se ve afectada por otros efectos, como pueden ser la reflexión, la refracción, el scattering, la difracción o el shadowing.

A causa de estos efectos la señal puede llegar al receptor a través de diversos caminos creando una suma de la señal directa y de sus versiones desplazadas en fase, tomando el nombre de "fading".

Los efectos de fading más importantes son el fading corto, conocido como el cambio rápido de la señal resultante recibida y el fading largo, que es un cambio lento de la potencia media recibida.

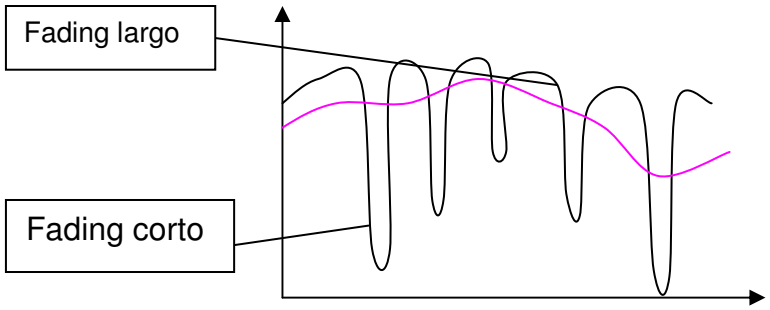


Fig. 2 Tipos de desvanecimiento

1.2. Zigbee

Zigbee es un protocolo de comunicaciones inalámbrico, similar al Bluetooth, basado en el estándar para redes inalámbricas de área personal IEE 802.15.4.

Principalmente se utilizará en el ámbito de la domótica, en entornos industriales y en entornos médicos, debido a su bajo consumo, su sistema de comunicaciones vía radio (con topología MESH) y su fácil integración, su poco caudal de datos, su fiabilidad, el tamaño diminuto de sus componentes y su complejidad reducida.

Zigbee es un simple protocolo de paquetes diminutos para redes inalámbricas ligeras.

Una red Zigbee, a diferencia de los 8 nodos que soporta Bluetooth, puede constar hasta un máximo de 255 nodos en una subred y su rango de distancia es de 10 a 75 metros. Una de sus grandes características es su bajo consumo, que lo consigue debido a que la mayor parte del tiempo está en estado

dormido. Su ancho de banda es inferior al de Bluetooth, 250Kbps (mundial), 20Kbps (Europa) o 40Kbps (EE.UU.) frente a 1Mbps, esto da lugar a unos 20Kbps por canal.

1.2.1. Arquitectura

El estándar IEEE 802.15.4 define una arquitectura basada en el modelo de referencia para la interconexión de sistemas abiertos (sistema de referencia OSI), tal como se puede observar en la Fig. 3.

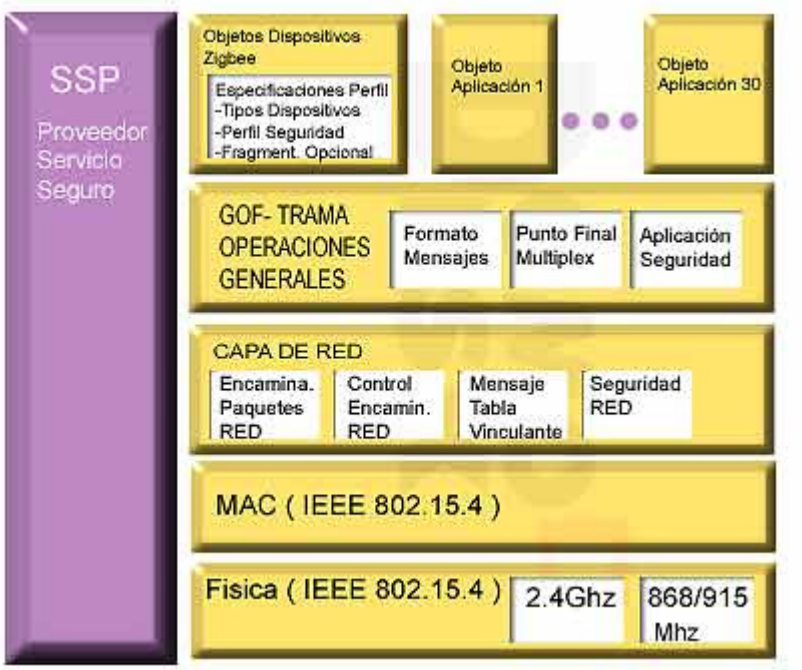


Fig. 3 Modelo de referencia OSI de Zigbee

Esta arquitectura abarca la capa física que contiene el transmisor-receptor de radiofrecuencia (RF) junto con su mecanismo de control de bajo nivel y una subcapa de control de acceso al medio (MAC) que proporciona el acceso al canal físico para todas las transferencias. Las capas superiores son definidas por la Alianza Zigbee.

1.2.2. Protocolo de acceso al medio

En Zigbee el medio es compartido para todos los usuarios, por tanto necesitamos un protocolo de acceso al medio para que controle las transmisiones que se pueden producir en este canal, y evitar así las posibles

colisiones que puedan producirse. Además, en redes Wireless, no se puede escuchar y transmitir a la vez, así que se hubo de escoger un mecanismo de acceso que fuera el adecuado.

En este caso, Zigbee utiliza el protocolo de acceso al medio CSMA-CA, que significa Carrier Sense Multiple Access Collision Avoidance, es decir, Acceso múltiple por detección de portadora con evitación de colisiones.

Cada equipo, antes de transmitir mira el NAV, que es un contador que indica cuanto tiempo estará ocupado el canal. Si este marca 0, quiere decir que el canal está libre y por tanto escucha el medio un intervalo aleatorio pequeño IFS, para asegurarse que otra estación no vaya a transmitir. Si después de este intervalo sigue escuchando el medio libre, inicia la transmisión de la trama, actualizando el NAV al tiempo que está estación tarde en transmitir, y por tanto tenga el canal ocupado. Una vez ha acabado la transmisión, esperamos su reconocimiento, ACK. Si éste llega, hemos podido transmitir satisfactoriamente, si éste no llega esperamos un tiempo de backoff y volvemos a escuchar el NAV para intentar volver a transmitir.

En el caso que cuando escuche el canal el NAV no marque 0, la estación seguirá mirando el valor de NAV hasta que éste marque 0, para intentar transmitir así su trama.

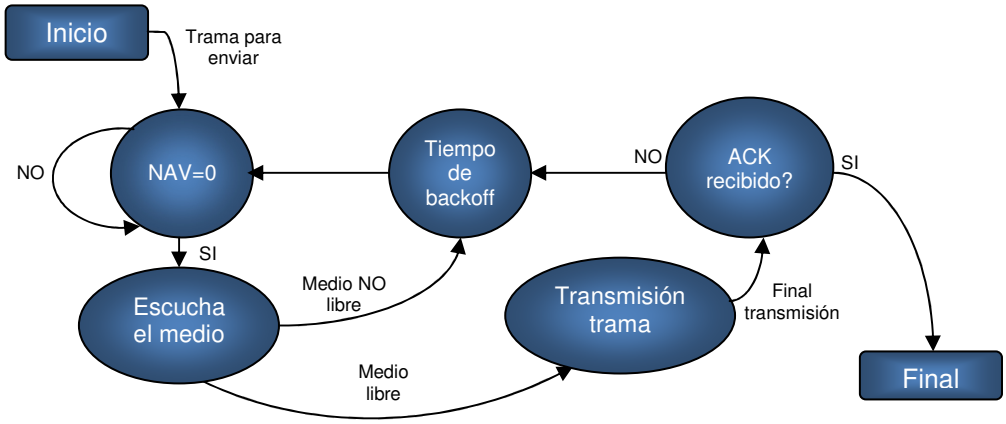


Fig. 4 Esquema CSMA

Este protocolo de acceso al medio, es uno de los más utilizados, pero también tiene un par de puntos débiles, como pueden ser los nodos ocultos y los nodos expuestos. En el primer caso, una estación cree que el canal está libre, pero en realidad está ocupado por otro nodo al que no escucha. El segundo caso es lo contrario, una estación cree que el canal está ocupado, pero está libre, puesto que el nodo al que escucha no la interfiere a ella.

1.3. Redes de sensores

Una red de sensores es una red de ordenadores pequeños, normalmente nombrados nodos, equipados con sensores, que colaboran en una tarea común.

Este tipo de redes están formadas por un grupo de sensores con ciertas capacidades sensitivas y de comunicación inalámbrica, los cuales permiten formar redes ad hoc sin infraestructura física preestablecida ni administración central.

Esta clase de redes se caracterizan por su facilidad de despliegue y por ser auto-configurables, pudiendo convertirse en todo momento en emisor, receptor, ofrecer servicios de encaminamiento entre nodos sin visión directa, así como registrar datos referentes a los sensores locales de cada nodo. Otra de sus características es su gestión eficiente de la energía, que les permite obtener una alta tasa de autonomía que las hacen plenamente operativas.

Las redes de sensores tienen un gran abanico de aplicaciones. A continuación nombraremos las más importantes:

- **Entornos de alta seguridad**: Se suelen utilizar en lugares donde se requiere un alto nivel de seguridad para evitar ataques terroristas, tales como centrales nucleares, aeropuertos, edificios del gobierno de paso restringido. Aquí gracias a una red de sensores se pueden detectar situaciones que con una simple cámara sería imposible.
- **Sensores ambientales**: El control ambiental de vastas áreas de bosque o de océano, sería imposible sin las redes de sensores, las cuales controlan múltiples variables, como pueden ser la temperatura, la humedad, el fuego, la actividad sísmica, etc.
- **Sensores industriales**: Dentro de fábricas existen complejos sistemas de control de calidad, donde el pequeño tamaño de estos sensores les permite estar allí donde se requiera.
- **Automoción**: Las redes de sensores son el complemento ideal a las cámaras de tráfico, ya que pueden informar de la situación del tráfico en ángulos muertos que no cubren las cámaras y también pueden informar a conductores de la situación, en caso de atasco o accidente.
- **Medicina**: Gracias al pequeño tamaño de los sensores, la calidad de vida de pacientes que tengan que tener controlada sus constantes vitales (pulsaciones, presión, nivel de azúcar en sangre, etc.), podrá mejorar substancialmente.
- **Domótica**: Su tamaño, economía y velocidad de despliegue, lo hacen una tecnología ideal para domotizar el hogar a un precio asequible.

1.3.1. Dispositivos para redes de sensores

Las redes de sensores están formadas por Motas, las cuales podemos clasificar en los siguientes grupos:

- **CROSSBOW:** Especializada en el mundo de los sensores, es una empresa que desarrolla plataformas hardware y software que dan soluciones a las redes de sensores inalámbricas. Entre sus productos encontramos las plataformas Mica, Mica2, Micaz, Mica2dot, telos y telosb.
- **MOTEIV:** Joseph Polastre, antiguo doctorando de un grupo de trabajo de la Universidad de Berekeley formó la compañía Moteiv. Ha desarrollado la plataforma Tmote Sky y Tmote Invent.
- **SHOCKFISH:** Empresa suiza que desarrolla TinyNode. A partir de este tipo de mota en Laussane han llevado un proyecto, en el que implementan una red de sensores en todo el campus de la "Ecole Polytechnique Fédérale de Lausanne".

En el caso particular de nuestro proyecto, las motas seleccionadas son del tipo telosb.

1.3.2. Plataforma telosb

Crossbow's TelosB mote es una plataforma de código abierto, de baja potencia diseñada para el estudio y la experimentación.

Esta plataforma incluye capacidad para puerto USB, una antena integrada que trabaja con el protocolo IEEE 802.15.4, bajo consumo de energía MCU con memoria extendida y la posibilidad de incluir más sensores, como por ejemplo de temperatura o de presión.

A continuación vamos a nombrar sus principales características:

- Compatible con IEEE 802.15.4.
- Alta velocidad de datos radio, 250kbps.
- Contiene el microcontrolador TI MSP430 con 10kB RAM.
- Antena integrada en la placa.
- Recolección de datos y programación vía interface USB.
- Sistema operativo de código abierto.
- Sensores opcionales (temperatura, luz).

Fig. 5 Mota telosb de Crossbow

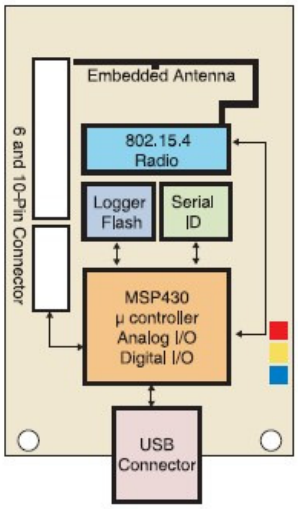


Fig. 6 Esquema interno motas telosb

1.3.3. Chip CC2420

El chip CC2420 es un dispositivo complejo, el cual se encarga de los detalles de bajo nivel de transmisión y recepción de paquetes a través del hardware.

Para poder especificar el comportamiento de este hardware se requiere la implementación de una pila radio bien definida. Aunque muchas de las funcionalidades están disponibles dentro del propio chip, también encontramos varios factores a considerar cuando se implementa la pila radio.

El software de la pila que controla el sistema radio del chip CC2420 consiste en muchas capas, las cuales se encuentran entre la aplicación y el hardware. El nivel más alto de la pila radio modifica datos y cabeceras de cada paquete, mientras que el nivel más bajo determina el comportamiento de emisión y recepción de los paquetes.

Para entender la funcionalidad de cada capa, así como también su arquitectura, es posible extender o minimizar la pila radio para satisfacer los requisitos de cada proyecto particular.

Las principales capas del chip CC2420 son las siguientes:

- ActiveMessageP: Esta es la capa más alta de la pila y por tanto es la encargada de rellenar los detalles de la cabecera y los datos del paquete de aplicación.
- Unique Send: Esta capa se ocupa de generar un byte de número de secuencia de los datos. Este byte se incrementa cada vez que se envía un paquete, partiendo de un primer número de secuencia pseudo-

aleatorio. De esta forma, el receptor puede detectar paquetes duplicados.

- PacketLink: Esta capa proporciona la funcionalidad de retransmisión automática y también se encarga de retransmitir una transmisión de un paquete sino existe reconocimiento desde el receptor.
- DefaultLpl: Esta capa proporciona implementaciones asíncronas de escucha de baja potencia. Esta capa soporta PowerCycleP, la cual se encarga de activar y desactivar el sistema radio y controlar la recepción de paquetes.
- UniqueReceive: Esta capa tiene un historial de la dirección de origen y del byte DSN de los últimos paquetes recibidos. También ayuda a filtrar los paquetes duplicados recibidos.
- TinyosNetwork: Esta capa permite interactuar con otras redes que no son TinyOS.
- CSMA: Esta capa es la encargada de definir el byte de información FCF del paquete saliente, que es un campo de 2 bytes que indica el tipo de trama MAC, proporcionando además un backoff por defecto si el canal esta ocupado en ese momento.
- TransmitP/ReceiveP: Estas capas son las responsables de interactuar directamente con el canal radio, a través del bus SPI, de las interrupciones y de las líneas GPIO.

1.4. Cooperatividad en redes inalámbricas

Una de las técnicas más eficientes para disminuir el efecto causado por el multi-path fading (atenuación por multi-camino) en canales inalámbricos es la diversidad en recepción. La diversidad se basa en el hecho de que existe una baja probabilidad de que las señales que vienen por caminos independientes sufran grandes atenuaciones simultáneamente. Un ejemplo de esto es la diversidad de espacio, en que los nodos usan una red de antenas, separadas una distancia. La coherente combinación de las diferentes señales recibidas llevan al incremento en recepción del SNR un factor M', dónde M' es el número de elementos en la red de antenas.

Sin embargo, en muchos casos no se puede asumir que los nodos puedan soportar múltiples antenas y por lo tanto, que puedan obtener ventajas de los beneficios de la diversidad de espacio. Recientemente, una nueva clase de métodos en el que nodos vecinos con una única antena colaboran en orden de generar virtualmente una multi antena han sido estudiados. En cualquier caso, explotar la diversidad en la capa física lleva a un diseño de radiofrecuencia más complejo.

En este punto, vamos a explicar un poco la diversidad y la colaboración entre nodos vecinos como un mecanismo usado en la mayoría de pilas de comunicación: Automatic Repeat Request (ARQ), sacando ventajas de la colaboración basada en decisiones en lugar de la colaboración basada en datos.

En la mayoría de redes cableadas, los paquetes viajan a través de un único camino de origen a destino. Este camino consiste generalmente en varios saltos entre los nodos de comunicación. En ARQ uno de estos nodos guarda una copia correcta del paquete después de la transmisión del paquete. Si la transmisión falla, este nodo puede retransmitir correctamente la copia del paquete otra vez, esperando que esta vez la retransmisión sea correcta. En TCP, por ejemplo, el nodo que guarda la copia del paquete es el emisor. En otros protocolos, como HDLC y sus derivaciones, la copia del paquete la guarda el último nodo, por el que viaja el paquete. La combinación de Forward Error Correction y ARQ es conocido como Hybrid ARQ schemes.

En redes inalámbricas la suposición de que hay un único camino de comunicación entre el origen y el destino no es normalmente cierta. Cuando un nodo transmite un paquete, debido a la naturaleza inherente de tipo broadcast de los medios sin cables, muchos nodos lo recibirán, aunque la mayoría de ellos lo descartaran después de comprobar que ellos no son los destinatarios. A más a más, la potencia de transmisión es proporcional a la distancia.

En transmisiones inalámbricas puede pasar, por ejemplo, que un nodo vecino del nodo destinatario reciba correctamente el paquete o parte de él, mientras que el destinatario lo haya podido recibir con errores. Hablando de potencia, sería más eficiente, si este nodo vecino entrega la copia del paquete al destinatario final en vez de pedir una retransmisión del paquete al nodo origen.

Este esquema al que llamamos *Collaborative* ARQ, está basado en la idea previa de retransmisión de copias del paquete desde el nodo vecino. Demostramos que M nodos colaboradores pueden conseguir la misma BER frente al SNR en función de una red de M' antenas donde M=p/2xM'≈1.57 M' en canales AWGN y que M=2M'-1 en canales Rayleigh. Estos resultados implican que la colaboración ARQ puede llegar a aumentar el ahorro en términos de potencia de transmisión, así como también consigue una diversidad del orden de (M+1)/2.

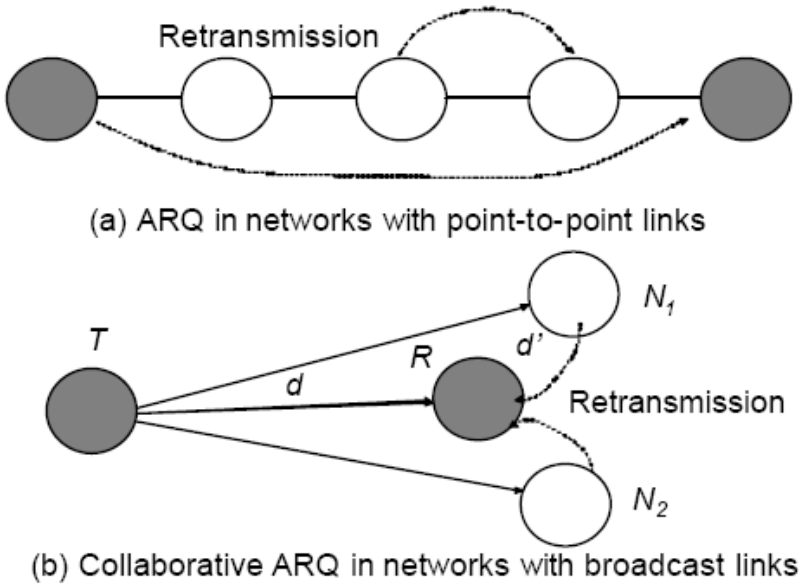


Fig. 7 Método colaborativo ARQ

Aunque en ambos casos M>M', la complejidad del hardware, es mucho más pequeña en el caso de la colaboración ARQ: Para los M nodos vecinos, si usamos *collaborative* ARQ sólo necesitaremos M antenas (1 por nodo), mientras que si usamos diversidad de espacio necesitaremos MxM'. Cada antena adicional en un nodo implica la complejidad de añadir un nuevo dispositivo de radiofrecuencia y un convertidor AD, sin tener en cuenta el uso extra de procesado en el nodo. De hecho la implementación del *collaborative* ARQ sólo requiere cambios de firmware.

Collaborative ARQ, como cualquier otro método (o esquema) de colaboración, es un esquema oportunista, ya que sólo puede funcionar si la transmisión del nodo llega a un grupo de vecinos.

Este esquemas o alguna variante de él, pueden ser aprovechadas en sistemas sin hilos donde la simplicidad del hardware y el consumo de potencia son la mayor limitación en el diseño, como es el caso de las redes de sensores inalámbricas.

1.4.1. El método colaborativo ARQ

Asumimos que la distancia entre dos nodos T y R es d. Hay M-1 nodos situados a una distancia d' de R, con d'<<d, (véase figura 7). Llamaremos a estos nodos, vecinos de R. Aproximaremos la distancia entre T y los nodos de R al valor d. Y también asumimos que los canales entre R, los vecinos de R y T tienen atenuación independiente.

Dejemos que T transmita un paquete con destino R, que será recibido por R y por sus vecinos. Asumimos que esos nodos pueden identificar que el destinatario del paquete es R, incluso cuando hay errores. Después de recibir el paquete, cada nodo comprueba si llega correctamente usando CRC. Sólo si el paquete no estaba destinado a ellos, o el paquete fue recibido con errores, el nodo R o sus vecinos guardaran una copia de él. En el caso que el nodo R encuentre que ese paquete contiene errores, enviará un paquete de señalización a sus vecinos, pidiendo la retransmisión de sus copias del paquete (CFC). Una vez R ha recibido la información enviada por sus vecinos (y asumiendo que ninguna de las copias de los paquetes recibidos tienen un CRC correcto), usará *majority voting* bit a bit para la construcción de un "hipotético" paquete recibido. Si éste paquete reconstruido es aún incorrecto, la retransmisión del paquete desde el nodo T será pedida. (Véase la Fig. 8 y Fig. 9).

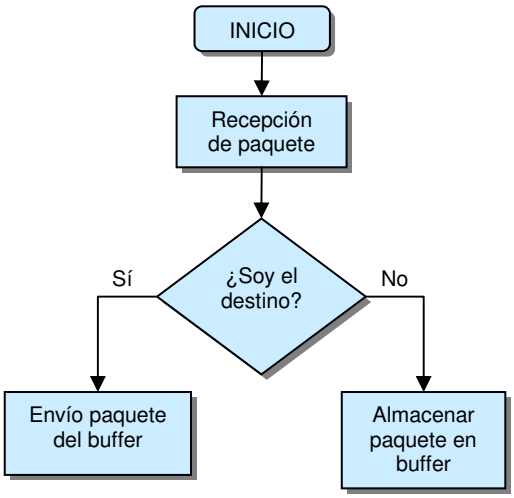


Fig. 8 Diagrama de flujo del nodo vecino.

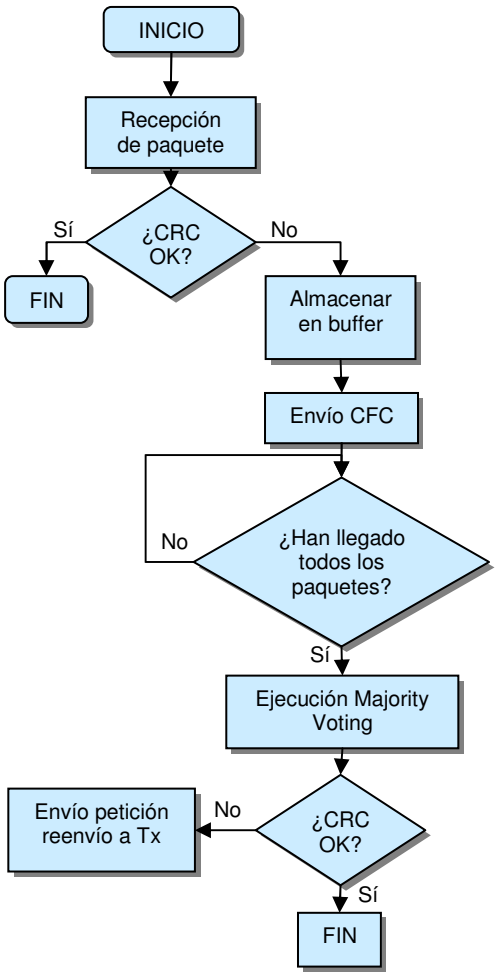


Fig. 9 Diagrama de flujo del nodo receptor

Hay muchas variantes posibles para este mecanismo básico. Por ejemplo, el nodo R puede primero pedir a sus vecinos las copias correctas de los paquetes. Sólo si no encuentra esas copias correctas, solicitará copias incorrectas. Los vecinos que hayan tenido un valor bajo de SNR durante la recepción del paquete, pueden inhabilitarse de mandar una copia (éstos nodos seguramente tendrán un elevado número de bits erróneos). Alternativamente, ellos pueden enviar junto con la copia del paquete, una estimación de su SNR, así durante el proceso de majority voting, las copias con mejor SNR tendrán más peso. Algún método de corrección de errores también puede ser usado. Los mecanismos adicionales requeridos para determinar los vecinos dispuestos a cooperar, para pedir las copias de los paquetes, para así aumentar la probabilidad de recepción de una correcta dirección de destino en el paquete, están fuera de este proyecto.

El proceso de majority voting se trata de una votación por mayoría, mirar bit a bit que valor se ha recibido más veces. El valor que tenga más repeticiones será el escogido para reconstruir ese bit del paquete.

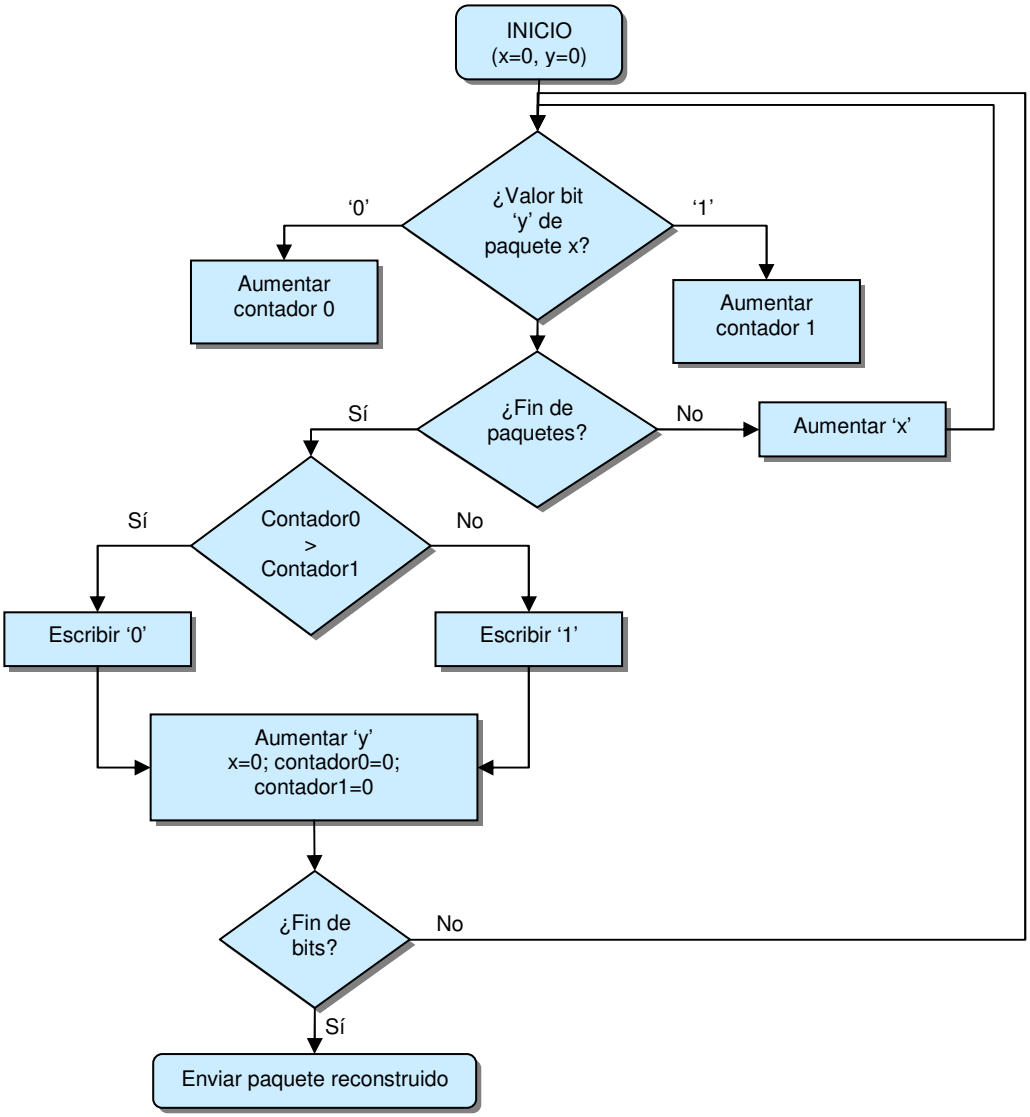


Fig. 10 Diagrama de bloques del 'Majority Voting'

1.5. Entorno de trabajo

Para el desarrollo de este proyecto hemos utilizado dos entornos de trabajo, como son el sistema operativo Tinyos y su lenguaje de programación NesC, los cuales vamos a desarrollar a continuación.

1.5.1. Sistema operativo Tinyos

TinyOS es un sistema operativo de código abierto basado en componentes para redes de sensores inalámbricos. El lenguaje en el que se encuentra programado TinyOS es un meta-lenguaje que deriva de C, cuyo nombre es

NesC. Este lenguaje esta escrito como un conjunto de tareas y procesos que colaboran entre sí y está diseñado para incorporar nuevas innovaciones rápidamente y para funcionar bajo las importantes restricciones de memoria que se dan en las redes de sensores. TinyOS está desarrollado por un consorcio liderado por la Universidad de California en Berkeley en cooperación con Intel Research.

El diseño del Kernel de TinyOS está basado en una estructura de dos niveles de planificación.

- Eventos: Pensados para realizar un proceso pequeño (por ejemplo cuando el contador del timer se interrumpe, o atender las interrupciones de un conversor análogo-digital). Además pueden interrumpir las tareas que se están ejecutando.

- Tareas: Las tareas están pensadas para hacer una cantidad mayor de procesamiento y no son críticas en tiempo (por ejemplo calcular el promedio en un arreglo). Las tareas se ejecutan en su totalidad, pero la solicitud de iniciar una tarea, y el término de ella son funciones separadas.

Con este diseño permitimos que los eventos (que son rápidamente ejecutables), puedan ser realizados inmediatamente, pudiendo interrumpir a las tareas (que tienen mayor complejidad en comparación a los eventos).

El enfoque basado en eventos es la solución ideal para alcanzar un alto rendimiento en aplicaciones de concurrencia intensiva. Adicionalmente, este enfoque usa las capacidades de la CPU de manera eficiente y de esta forma gasta el mínimo de energía.

1.5.2. Lenguajes de programación

1.5.2.1 NESC

Las aplicaciones para TinyOS se escriben en nesC, un dialecto del lenguaje de programación C optimizado para las limitaciones de memoria de las redes de sensores.

Además soporta un modelo de programación que integra el manejo de comunicaciones, las concurrencias que provocan las tareas y eventos y la capacidad de reaccionar frente a sucesos que puedan ocurrir en los ambientes donde se desempeña.

También realiza optimizaciones en la compilación del programa, detectando posibles carreras de datos que pueden ocurrir producto de modificaciones concurrentes a un mismo estado, dentro del proceso de ejecución de la

aplicación. Además simplifica el desarrollo de aplicaciones, reduce el tamaño del código, y elimina muchas fuentes potenciales de errores.

Básicamente NesC ofrece:

- Separación entre la construcción y la composición. Hay dos tipos de componentes en NesC: módulos y configuraciones. Los módulos proveen el código de la aplicación, implementando una o más interfaces. Estas interfaces son los únicos puntos de acceso a la componente. Las configuraciones son usadas para unir las componentes entre sí, conectando las interfaces que algunas componentes proveen con las interfaces que otras usan.
- Interfaces bidireccionales: las interfaces son los accesos a las componentes, conteniendo comandos y eventos, los cuales son los que implementan las funciones. El proveedor de una interfaz implementa los comandos, mientras que el que las utiliza implementa eventos.
- Unión estática de componentes, vía sus interfaces. Esto aumenta la eficiencia en tiempos de ejecución, incrementa la robustez del diseño, y permite un mejor análisis del programa.
- NesC presenta herramientas que optimizan la generación de códigos. Un ejemplo de esto es el detector de “carreras de datos”, en tiempo de compilación.

En conclusión, los principales aspectos que el modelo de programación NesC ofrece y que deben ser entendidos para el entendimiento del diseño con TinyOS son:

- El modelo de NesC está formado por interfaces y componentes.
- Una interfaz puede ser usada o puede ser provista, las componentes son módulos o configuraciones.
- Una aplicación se verá representada como un conjunto de componentes, agrupados y relacionados entre sí, tal como se observa en la Fig. 11.

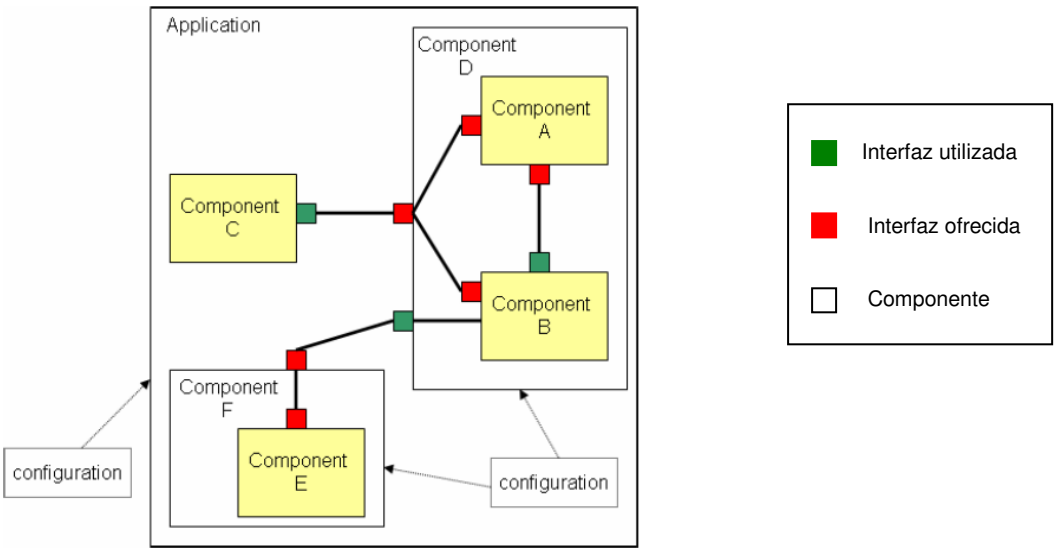


Fig. 11 Modelo NESC

1.5.2.2 Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

El lenguaje fue diseñado con las siguientes características en mente:

- Simple. Elimina la complejidad de los lenguajes como "C" y da paso al contexto de los lenguajes modernos orientados a objetos.
- Familiar. Como la mayoría de los programadores están acostumbrados a programar en C o en C++, la sintaxis de Java es muy similar al de estos.
- Robusto. El sistema de Java maneja la memoria de la computadora por ti. No te tienes que preocupar por apuntadores, memoria que no se esté utilizando, etc. Java realiza todo esto sin necesidad de que uno se lo indique.
- Seguro. El sistema de Java tiene ciertas políticas que evitan se puedan codificar virus con este lenguaje. Existen muchas restricciones, especialmente para los applets, que limitan lo que se puede y no puede hacer con los recursos críticos de una computadora.
- Portable. Como el código compilado de Java (conocido como byte code) es interpretado, un programa compilado de Java puede ser utilizado por cualquier computadora que tenga implementado el interprete de Java.
- Independiente a la arquitectura. Al compilar un programa en Java, el código resultante un tipo de código binario conocido como byte code. Este código es interpretado por diferentes computadoras de igual manera, solamente hay que implementar un intérprete para cada plataforma. De esa manera Java logra ser un lenguaje que no depende de una arquitectura computacional definida.
- Multithreaded. Un lenguaje que soporta múltiples threads, es decir, es un lenguaje que puede ejecutar diferentes líneas de código al mismo tiempo.
- Interpretado. Java corre en máquina virtual, por lo tanto es interpretado.
- Dinámico. Java no requiere que compiles todas las clases de un programa para que este funcione. Si realizas una modificación a una clase Java se encarga de realizar un Dynamic Bynding o un Dynamic Loading para encontrar las clases.

CAPÍTULO 2. Estudio de transmisiones cooperativas

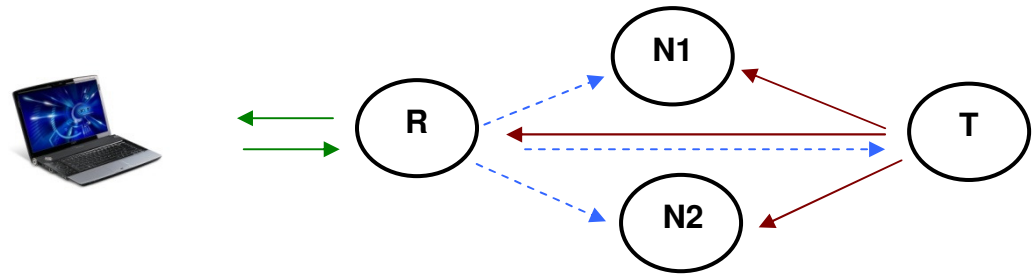
2.1. Funcionamiento del escenario de estudio

Este proyecto esta centrado en la utilización de técnicas de cooperación para optimizar las redes de sensores inalámbricos.

El objetivo de nuestro trabajo es mejorar el número de paquetes recibidos correctamente en recepción, utilizando nodos intermedios para retransmitir el paquete. En especial nos centraremos en escenarios donde la cobertura entre el emisor y el receptor sea mínima.

Por ello, queremos demostrar que utilizando este tipo de mecanismos de cooperación, el número de paquetes recibidos en recepción será mayor.

Para poder demostrar esta premisa, hemos diseñado el escenario mostrado en la Fig. 12.



T - Transmisor **N** - Nodo intermedio **R** – Receptor

Sentido de la comunicación PC-Estación Base y Estación Base-PC

Sentido de la comunicación de los paquetes de sincronismo.

Sentido de la comunicación de los paquetes de transmisión

Fig. 12 Funcionamiento escenario sin errores en recepción

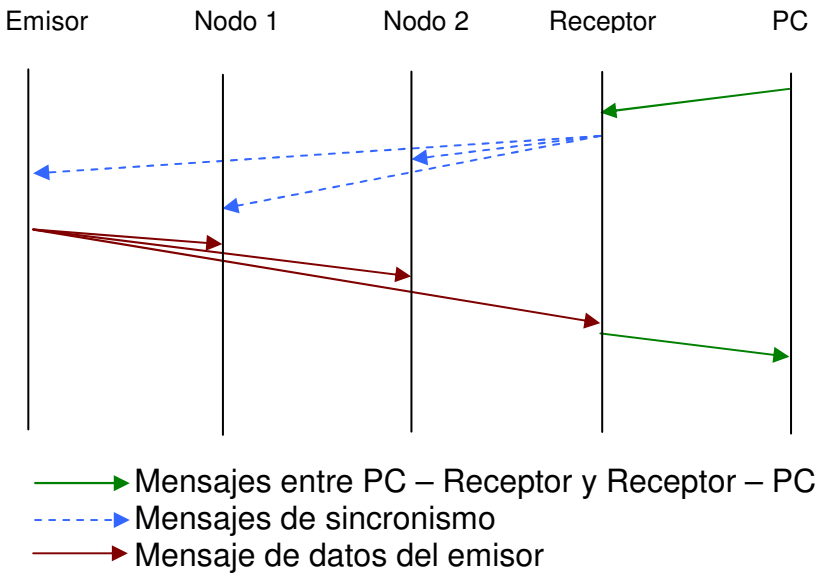


Fig. 13 Esquema general del escenario sin errores en recepción

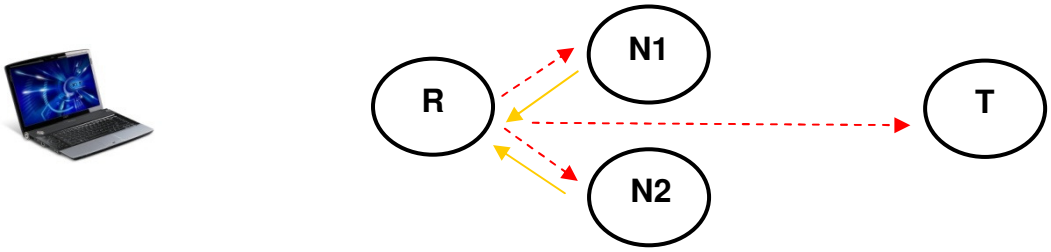
En el caso del escenario de estudio, el usuario configura el número de paquetes a enviar y la potencia de envío. Esta información se envía en un mensaje de sincronismo en modo broadcast, tal como se puede apreciar en la Fig. 12. Una vez este mensaje haya llegado al transmisor, éste envía los paquetes en modo broadcast. Los nodos intermedios ya que no son el destinatario de dicho mensaje, se guardarán el paquete en una posición de buffer junto con el número de secuencia. Sin embargo, el receptor al ser el destinatario, una vez recibe el paquete comprueba el CRC, y si es correcto, envía los datos al PC con el propósito de que la aplicación lo muestra por pantalla.

En cambio, cuando el receptor comprueba el CRC y es incorrecto, sabemos que el paquete es corrupto y por lo tanto, debemos reconstruirlo.

En este caso, el receptor enviará un paquete de sincronismo broadcast. El transmisor obviará este paquete y solo los nodos intermedios lo tratarán. Éstos reenvían el último paquete, el cual tienen guardado en su buffer, al receptor para que éste intente reconstruirlo junto con el paquete que ha recibido del emisor. Véase Fig.10.

Por último, el receptor aplicará el método de *majority voting* (desarrollado en el apartado 2.1.2) para poder reconstruir el paquete. Una vez el paquete ha sido reconstruido lo enviará a la aplicación para mostrarlo por pantalla.

Por tanto, el funcionamiento de solicitud y reenvío del paquete se muestra en la Fig. 14:



T - Transmisor **N** - Nodo intermedio **R** - Receptor

Sentido de la comunicación de los paquetes de sincronismo de solicitud de paquete.

Sentido de la comunicación del paquete N reenviado al receptor

Fig. 14 Funcionamiento solicitud-reenvío de paquete erróneo

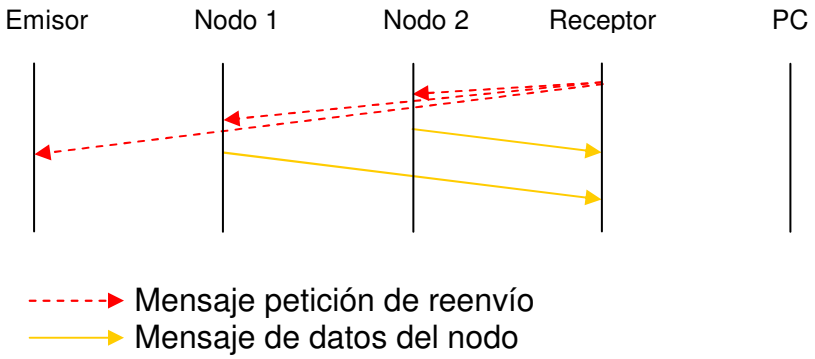
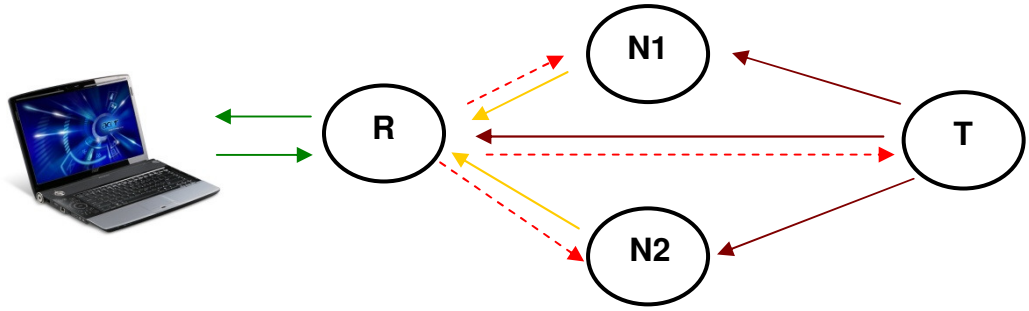


Fig. 15 Esquema general del escenario de solicitud-reenvío de paquete erróneo

Una vez explicado todo el proceso, el esquema general cuando en recepción llega un paquete corrupto se muestra en la Fig. 16:



T - Transmisor **N** - Nodo intermedio **R** – Receptor

Sentido de la comunicación de los paquetes de transmisión

Sentido de la comunicación de los paquetes de solicitud de paquete

Sentido de la comunicación del paquete N reenviado al receptor

Sentido de la comunicación PC-Estación Base y Estación Base-PC

Fig. 16 Funcionamiento escenario con errores en recepción

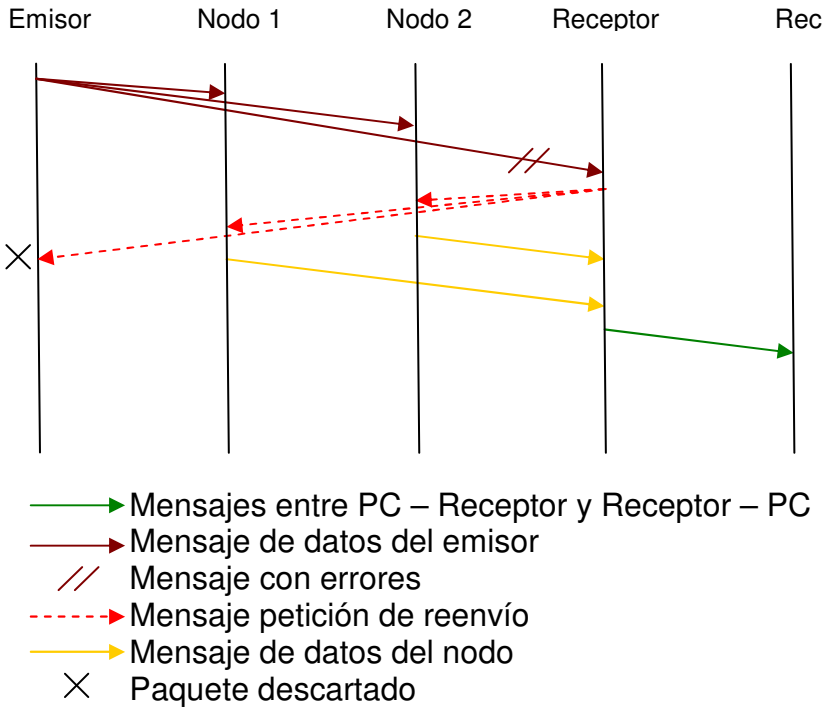


Fig. 17 Esquema general del escenario con errores en recepción

2.1.1. Recepción de paquetes erróneos

En cualquier transmisión de datos pueden ocurrir tres casos cuando se recibe un paquete:

- El paquete llega a destino correctamente
- El paquete no llega a destino
- El paquete llega a destino con errores

Por lo general, cuando una transmisión llega a destino con errores el paquete se descarta, ya que éste es corrupto. En cambio, en nuestro caso particular, nos interesa recibir este paquete para poder examinarlo y tratarlo de forma diferente al resto e intentar reconstruirlo en un paquete correcto.

Para detectar que un paquete contiene errores, se revisa un campo CRC que viene incluido en la cabecera del mismo paquete. El CRC es una función que se calcula a través de los datos del mensaje y se incluye en la cabecera antes que el transmisor envié el mensaje. Una vez el receptor ha recibido el mensaje, vuelve a calcular el CRC a través del contenido del mismo y lo compara con el CRC incluido en la cabecera. Si ambos CRC son iguales, significa que los datos no han variado, y por tanto el paquete es correcto. En el caso contrario, quiere decir que los datos han cambiado y por tanto el paquete contiene errores.

En nuestro caso particular, el encargado de verificar si el paquete contiene errores es el chip CC2420, el cual esta incorporado a las motas telosb, como ya se ha mencionado en el apartado 1.3.3.

Cuando se detecta que llega un paquete a la mota, éste se guarda en un buffer de recepción del tipo FIFO del chip CC2420 i automáticamente el CRC es verificado en hardware, poniendo un bit a 0 si detecta que el paquete es erróneo. Este bit es el bit más significativo del último byte de cada trama, tal como se muestra en la Fig. 18:

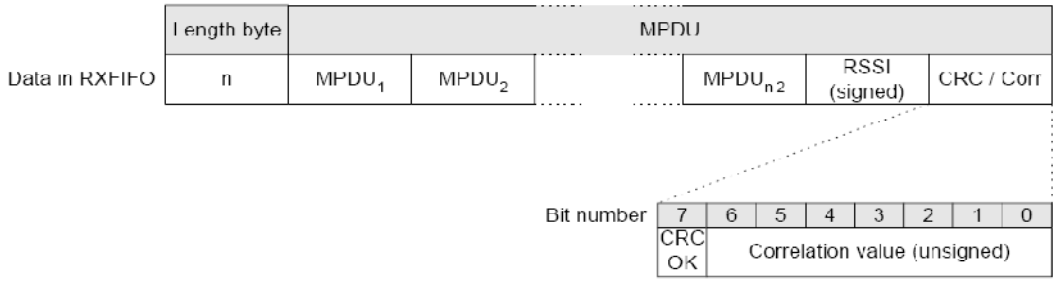


Fig. 18 Datos del paquete en la cola de recepción del chip CC2420

De todas formas, en hardware solo se modifica este bit si el mensaje contiene errores, pero es el software el encargado de descartar o no el paquete. El

driver del tinyos para CC2420 inspecciona este bit y por defecto descarta el paquete si el bit no es 1.

Puesto que en este proyecto nos interesa recibir también los paquetes con errores, fue necesario modificar esta parte del software. Dicha modificación esta descrita y explicada en el apartado 2.2.2.

2.1.2. Majority Voting

El mecanismo para reconstruir los paquetes erróneos que hemos decidido utilizar en nuestro proyecto, es el conocido como Majority Voting. La regla de la votación por mayoría es un mecanismo para tomar decisiones en grupo. Por la misma se establece que para establecer cual habrá de ser la decisión grupal o colectiva, se debe verificar cual es la opinión al respecto de un cierto número de miembros y adoptar aquella opción que cuente con el mayor número de apoyos.

Para poder desarrollar éste mecanismo, hemos implementado dos opciones diferentes.

2.1.2.1. Carácter a carácter

El primer mecanismo que desarrollamos fue carácter a carácter, ya que en nuestro caso, a la hora de enviar un paquete, en el campo datos enviamos un vector de caracteres. Ésta es una primera versión del majority voting ya que era la forma más sencilla de implementarlo.

En este caso, comparamos el primer carácter del paquete de datos recibido, tanto directamente del transmisor como de los nodos intermedios. Nos quedamos con el carácter que más veces haya salido repetido y en caso de empate se escoge la primera letra que esté más veces repetida. Este proceso se repite sucesivamente hasta llegar al final de los datos del paquete.

Un ejemplo de este mecanismo se puede ver en la Fig. 19:

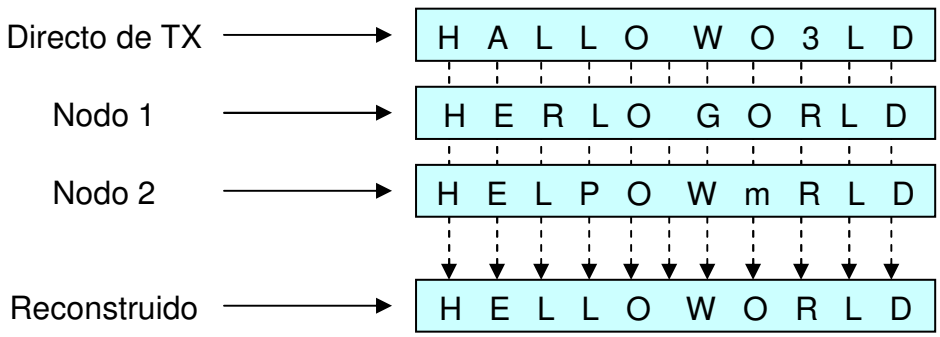


Fig. 19 Ejemplo Majority Voting carácter a carácter

Como podemos observar, tenemos tres paquetes erróneos recibidos, uno directamente del transmisor y los otros dos restantes desde los nodos intermedios. Al aplicar el método de majority voting carácter a carácter, se comprueba como se puede reconstruir un paquete.

2.1.2.2. Bit a bit

El mecanismo de mayority voting carácter a carácter no es del todo eficaz ya que los datos se transmiten bit a bit y sólo habiendo un fallo en uno de ellos obtenemos un carácter completamente diferente. Por tanto, es obvio pensar que desarrollar un mecanismo de comparación bit a bit optimizará el sistema.

Para poder simplificar el ejemplo, en este caso los datos a enviar, sólo seria un carácter, el cual podría ser la letra “H”. Por lo tanto, quedaría como se puede observar en la Fig. 20:

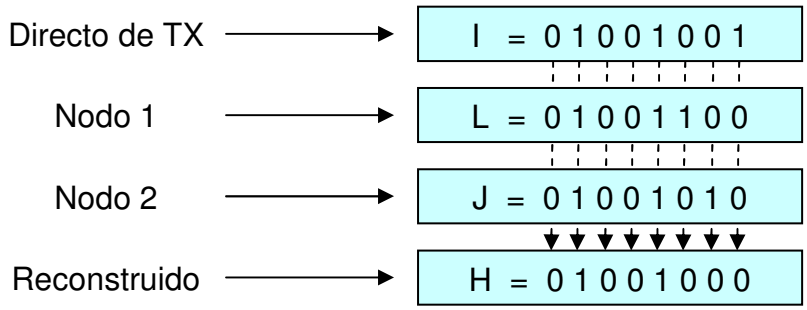


Fig. 20 Ejemplo Majority Voting bit a bit

En este caso se puede apreciar como ninguna de los tres mensajes recibidos contenían la letra “H”, debido a posibles interferencias del canal. Al aplicar el método bit a bit vemos como se puede reconstruir el paquete ya que sólo falla un bit en cada paquete.

En cambio, si aplicáramos el mecanismo de carácter a carácter, la reconstrucción del paquete seria la letra “I”, ya que los tres paquetes contienen letras diferentes y por tanto el mecanismo selecciona la primera dentro del empate.

Por tanto, con este ejemplo podemos demostrar que el mecanismo de bit a bit es más eficiente que el mecanismo carácter a carácter.

2.1.3. Algoritmos

En nuestro escenario existen tres tipos de motas. Cada una tiene un funcionamiento y un código distinto, según la función que desempeñan.

Para poder entender y analizar el comportamiento de cada una de ellas, vamos a describir sus funciones por separado.

2.1.3.1. Mota receptora o estación base

Para empezar cada experimento, el usuario debe introducir en el ordenador unos parámetros, como la duración deseada del mismo, la potencia, etc. Estos parámetros llegan a la mota receptora a través del puerto USB en el cual está conectada al PC.

Esta información se ha de enviar al resto de motas, para que estas puedan configurarse adecuadamente, por tanto, esta mota envía un mensaje de sincronismo broadcast al resto de las motas.

Otra de las funciones más importantes de esta mota es la recepción de los paquetes. Cada vez que llega un paquete es la encargada de verificar si el paquete tiene errores. Si el paquete no contiene errores, extrae la información necesaria para el usuario y la envía a través del puerto USB de nuevo al PC para mostrarla por pantalla. En cambio, si el paquete contiene errores es el encargado de realizar la reconstrucción del paquete. Para ello, lo primero que realiza es enviar un mensaje de sincronismo broadcast solicitando el último paquete que han almacenado en su buffer (excepto la mota emisora, que descarta el mensaje). Una vez le han llegado los paquetes de los nodos intermedios comprueba que éstos y el paquete que ella misma recibió, tienen el mismo número de secuencia. Si es así, es la encargada de realizar el majority voting con tal de reconstruir el paquete. Y en el caso, que los números de secuencia no fueran los mismos o no reciba todos los paquetes esperados de los nodos intermedios, no reconstruye nada y por tanto este paquete no se cuenta como recibido correctamente**.**

Una vez ya tenemos el paquete reconstruido, enviamos al PC los datos con tal de poder mostrarlos por pantalla. Tal como se muestra en el diagrama de flujo de la Fig. 21 :

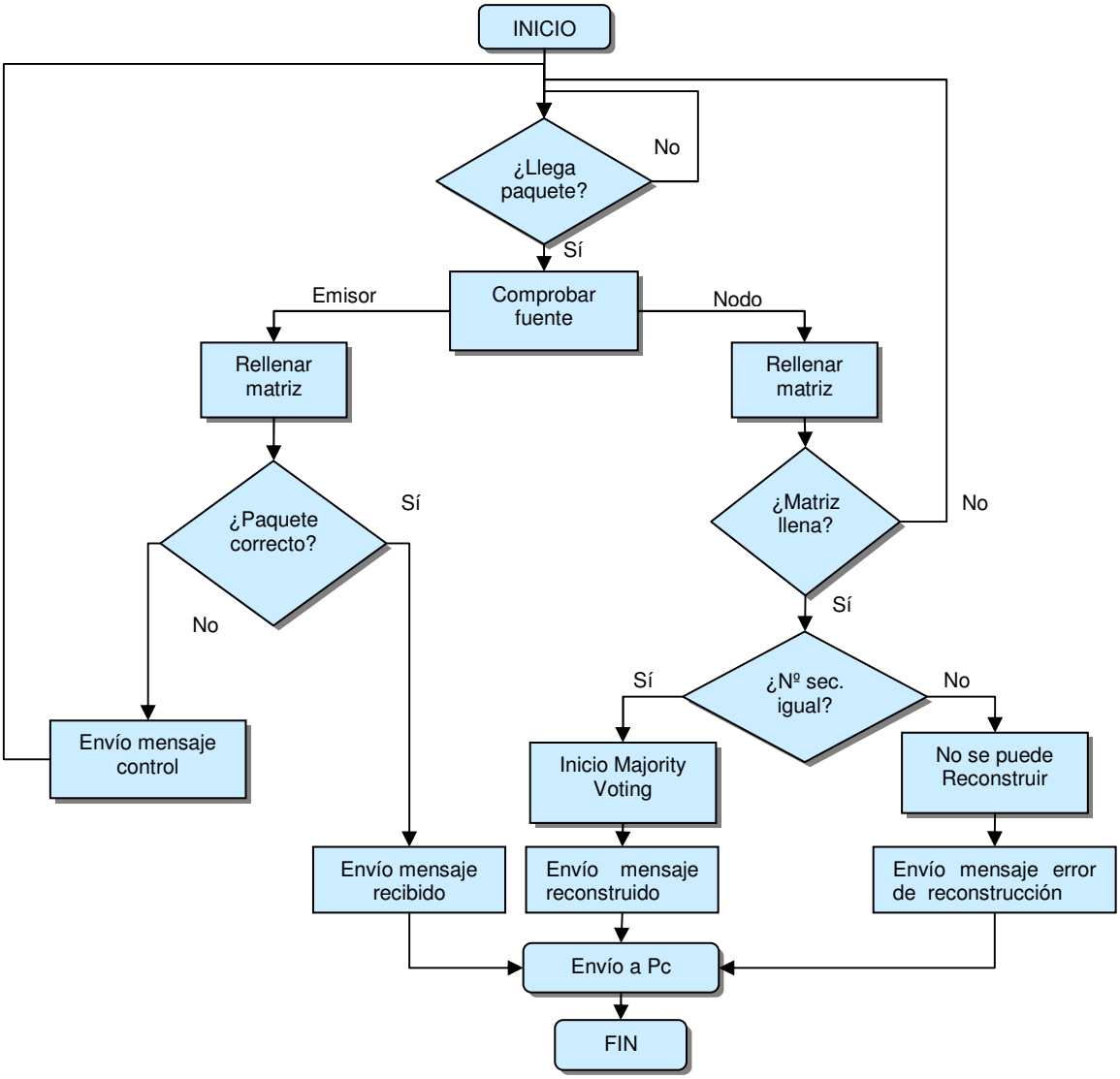


Fig. 21 Diagrama de flujo del sistema receptor

2.1.3.2. Mota emisora

La mota emisora es la encargada de enviar los paquetes a la mota receptora. Para ello, recibe un mensaje de la estación base con la orden explícita de transmitir mensajes periódicamente con una cierta potencia, definido previamente en la aplicación JAVA.

La mota contiene un contador interno que se va incrementado una unidad por cada paquete que envía. Dicho contador es enviado como un dato en el campo de información de los paquetes para ser analizado a posteriori por la estación base.

Para poder realizar más adelante pruebas con la técnica del majority voting, en el campo de datos de cada paquete se ha incluido un texto de prueba.

El esquema general del funcionamiento de esta mota, se puede ver reflejado en el diagrama de la Fig. 22.

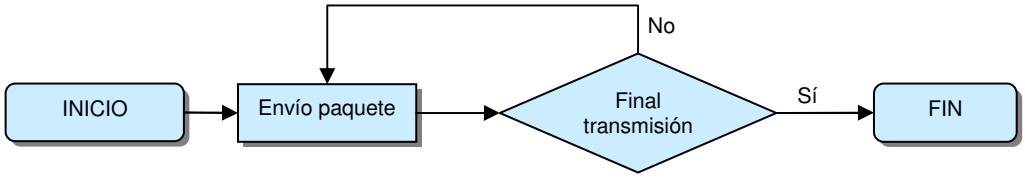


Fig. 22 Diagrama de flujo de la mota emisora

2.1.3.3. Mota nodo intermedio (relay)

Los nodos intermedios (relays) reciben tres tipos de paquetes, dos de sincronismo y uno de datos. Los paquetes de sincronismo se diferencian entre ellos con un campo del paquete, como veremos más adelante en el apartado 2.2.4.1.

El paquete de datos proviene del emisor, ya que son los datos que éste envía al receptor. Como la comunicación inalámbrica es de tipo broadcast de modo inherente, los nodos intermedios también reciben dicho paquete, y lo guardan en un buffer a la espera de una posible petición de reenvío cooperativo por parte del receptor.

El primer tipo de paquete de sincronismo proviene del receptor y contiene los parámetros de configuración de la potencia con la cual reenviará los paquetes de datos.

El segundo tipo también proviene del receptor, y contiene una solicitud de reenvió del último paquete guardado en su buffer para poder enviarlo al receptor y que éste último realice la reconstrucción del paquete.

El esquema general del funcionamiento de esta mota, se puede resumir en el diagrama de flujo de la Fig. 23. Este proceso se repetirá tantas veces como paquetes lleguen al nodo cooperante.

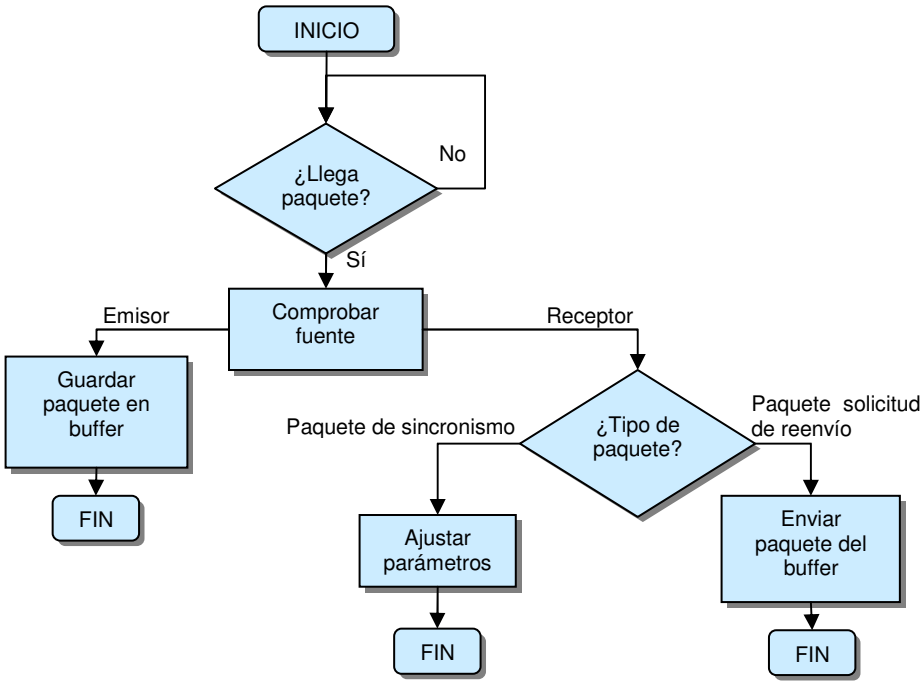


Fig. 23 Diagrama de flujo de la mota con función de relay

2.2. Implementación

A continuación vamos a describir las aplicaciones realizadas en este proyecto, así como su estructura.

A la hora de implementar el código que se ejecutará a posteriori en las motas y crear la interconexión entre la aplicación NesC y la aplicación en Java, se debe tener en cuenta el contenido de cuatro ficheros, que son necesarios para el correcto funcionamiento.

El primero de ellos es el Makefile, que aunque no es un fichero nesC, contiene las reglas para compilar la aplicación en NesC, así como generar los ficheros java necesarios para comunicar la mota con la aplicación java ejecutada en el PC. Una regla describe cómo crear el fichero objetivo a partir de los ficheros de los que éste depende.

El segundo es el fichero de cabecera, es un fichero C donde se definen las distintas estructuras de datos y diversas constantes que utiliza la aplicación tinyOs. Siempre tiene la extensión '.h'.

Por otro lado, NesC ofrece la separación entre la construcción y la composición es decir, hay ficheros módulo y ficheros configuración, por lo tanto hay dos tipos de códigos diferentes, uno para módulos y otro para configuraciones.

Por este motivo el tercer fichero importante es el que más peso tiene dentro de la aplicación NesC, éste es el fichero de configuración. Recibe el nombre de 'NombreAplicacionC.nc' con la C final del nombre, la cual indica que es un fichero de configuración.

Un fichero de configuración es el fichero de más alto nivel en una aplicación tinyOS, y en él se definen qué interfaces va a utilizar la aplicación nesC y que componentes las proporcionan mediante los wirings (conexión entre interfaces). Todos los ficheros de configuración constan de dos partes esenciales:

- Configuración: Es una línea de código donde indica que éste es un archivo de configuración.
- Implementación: En esta parte primero se hace la llamada a todos los componentes que se usaran en la aplicación y seguido a esta llamada se introducen los respectivos wirings.

```
#include "WsnAplication.h"
configuration WsnAplicationC {}
implementation {
  components MainC, WsnAplicationP as App, LedsC ;
  components SerialActiveMessageC as AM;
  components ActiveMessageC;
  components CC2420ControlC;
  components new AMReceiverC(AM_PACKETLINKMSG);
  components new AMSenderC(AM_PACKETLINKMSG);

  App.Boot -> MainC.Boot;
  App.SerialControl -> AM;
  App.RadioSplitControl -> ActiveMessageC;
  App.AMSend -> AM.AMSend[AM_WSN_APLICATION_MSG];
  App.Leds -> LedsC;
  App.Packet -> AM;
  App.radioSend->AMSenderC;
  App.AMPacket -> ActiveMessageC;
  App.ReadRssi -> CC2420ControlC.ReadRssi;
  App.Receive-> AMReceiverC;
  App.serialReceive->AM;
  App.PacketAcknowledgements -> ActiveMessageC;
}
```

Fig. 24 Ejemplo fichero de configuración

Como podemos ver en la Fig. 24, la primera línea de código se hace la llamada al fichero de cabecera WsnAplication.h, que contiene las estructuras de datos del mensaje. En nuestro caso solo llama a un fichero cabecera, pero los ficheros de configuración pueden utilizar más de uno.

La segunda línea de código es la que comentábamos como la parte de configuración, con ésa línea informamos que este fichero es de configuración.
En la tercera línea podemos observar como empieza la implementación, a partir de aquí vemos claramente diferenciadas dos partes. La primera, son todas las líneas que empiezan por la palabra reservada 'components' las cuales definen todos los componentes que intervendrán en la aplicación. Por

último, la segunda parte son los wirings. Por ejemplo la línea App.Receive->AMReceiverC indica que la interficie receive que usa App la proporciona el componente AMReceiverC.

El último tipo de fichero es el fichero módulo, el cual contiene el código del programa con todas las funcionalidades que debe realizar la mota. En este fichero también se pueden diferenciar dos partes, los módulos y la implementación.

La parte de módulos se declaran todas las interfaces que usa, y la implementación es el código en si del funcionamiento del sistema. Esta parte esta explicada a continuación ya que según la función de la mota, esta será la parte en la que nos centramos, ya que es la parte más modificada del proyecto anterior.

```
#include "Timer.h"
#include "WsnAplication.h"
 module WsnAplicationP
{     uses
      {     interface Leds;
            interface Boot;
            interface AMSend;
            interface SplitControl as SerialControl;
            interface SplitControl as RadioSplitControl;
            interface AMSend as radioSend;
            interface Packet;
            interface AMPacket;
            interface Receive;
            interface Read<uint16_t> as ReadRssi;
            interface Receive as serialReceive[am_id_t id];
            interface PacketAcknowledgements;
      }}
Implementation

{…}
```

Fig. 25 Ejemplo fichero módulo

2.2.1. Aplicación Previa

Para el desarrollo de este proyecto partimos de una aplicación previa, desarrollada en dos trabajos finales de carrera anteriores.

El escenario implementado en el primer proyecto consiste en dos motas, una emisora y otra receptora, las cuales se envían información. En cambio, en el segundo trabajo el escenario consiste en tres motas, una emisora, una receptora y un nodo intermedio. En este caso está configurado el envío de ACK's para la confirmación de los paquetes y la retransmisión del paquete perdido.

En nuestro caso particular, nos interesa una mezcla de ambos trabajos, por tanto, partimos del escenario del primer proyecto, donde sólo hay un emisor y un receptor, pero añadimos el nodo intermedio del segundo trabajo. El cual ha sido modificado para que simplemente guarde en un buffer el último paquete recibido por el receptor, por lo que no utilizamos técnicas de ACK, ya que nos centramos solamente en las posibles reconstrucciones de paquetes y no en las pérdidas de los paquetes globales.

Por tanto, nuestro TFC se centra en mejorar el número de paquetes recibidos en recepción utilizando la técnica de la votación por mayoría absoluta cuando una transmisión llega con errores en su contenido. De esta forma, se aumentará el porcentaje de paquetes recibidos con éxito.

2.2.2. Recepción de paquetes erróneos

Como ya hemos explicado en el apartado 2.1.1 el chip CC2420 por defecto descarta todos los paquetes que contienen errores tras la comprobación del CRC. Por consiguiente, hemos de modificar la parte de código que descarta dichos paquetes. Este cambio se aplica dentro del sistema operativo de tinyos, por tanto todas las aplicaciones implementadas bajo este sistema operativo, se verán afectadas por el cambio.

Esta modificación la hemos de realizar en la cola de recepción del chip CC2420 (RXFIFO). Para ello, se debe entrar dentro del módulo cc2420ReceiveP.nc y modificar el evento RXFIFO.readDone.

El módulo cc2420ReceiveP.nc se encuentra dentro de la novena capa del software de la pila radio que controla el sistema radio del chip cc2420, tal como se puede apreciar en la Fig. 26.

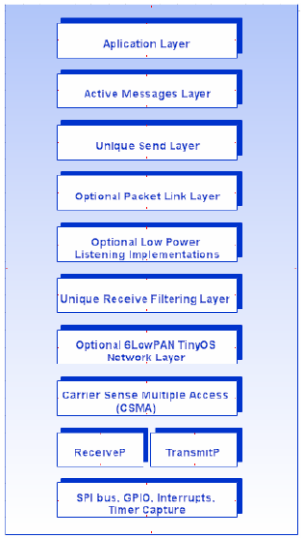


Fig. 26 Pila de capas del chip cc2420

El nivel más alto de la pila modifica los datos y la cabecera del paquete, mientras que el nivel más más bajo determina el comportamiento de la transmisión.

En el código original comprueba el bit del CRC, descartando el paquete si éste es 0, tal como se puede apreciar en la Fig. 27:

```
// We may have received an ack that should be processed by Transmit
// buf[rxFrameLength] >> 7 checks the CRC
      if ( ( buf[ rxFrameLength ] >> 7 ) && rx_buf ) {
        uint8_t type = ( header->fcf >> IEEE154_FCF_FRAME_TYPE ) & 7;
        signal CC2420Receive.receive( type, m_p_rx_buf );
            if ( type == IEEE154_TYPE_DATA ) {
                  post receiveDone_task();
                  return;
            }
      }
      waitForNextPacket();
      break;
```

Fig. 27 Código inicial del módulo cc2420ReceiveP.nc

Por ello, hemos de modificar este código con tal que no descarte este paquete. Es decir, no se ha de realizar la comprobación del bit de CRC (bit más significativo del último byte del paquete) y así devolver a la siguiente capa el paquete tanto si contiene errores como si no.

El código final del módulo cc2420ReceiveP.nc se puede apreciar en la Fig. 28:

```
// We may have received an ack that should be processed by Transmit
// buf[rxFrameLength] >> 7 checks the CRC
      if ( rx_buf ) {
        uint8_t type = ( header->fcf >> IEEE154_FCF_FRAME_TYPE ) & 7;
        signal CC2420Receive.receive( type, m_p_rx_buf );
            if ( type == IEEE154_TYPE_DATA ) {
                  post receiveDone_task();
                  return;
            }
      }
      waitForNextPacket();
      break;
```

Fig. 28 Código final del módulo cc2420ReceiveP.nc

2.2.3. Implementación del Majority Voting

La técnica del *majority voting* se aplica en el receptor, es decir, en el código WsnAplicationP.nc, ya que éste es el que recibe el paquete y el encargado de enviar el mensaje final a la aplicación.

La técnica de la mayoría absoluta sólo se aplica cuando un paquete del emisor llega al receptor con errores. En ese caso, el receptor envía un mensaje de sincronismo a los nodos intermedios para que le reenvíen dicho paquete, tal como habíamos explicado anteriormente en el apartado 2.1.

Una vez tenemos todos los paquetes en el receptor (tanto del origen como de los nodos intermedios) los guardamos todos en una matriz para ser procesados, tal como explicaremos en el apartado 2.2.4.1. Una vez hecho esto, se implementa una de las dos técnicas diferentes de mayoría absoluta, carácter a carácter o bit a bit, las cuales vamos a desarrollar a continuación.

2.2.3.1. Carácter a carácter

La primera técnica que se ha llevado a cabo para realizar una técnica de mayoría absoluta es carácter a carácter, por su sencillez sirve como base para realizar unas pruebas preliminares.

Una vez tenemos completada la matriz, hemos de verificar que todos los paquetes tengan el mismo número de secuencia. Si esto no fuera así, enviaríamos un mensaje al ordenador indicando que no hemos podido reconstruir el paquete. En caso contrario, vamos a ir comparando el primer carácter de cada paquete y así sucesivamente con el resto de caracteres, con tal de poder reconstruir el paquete. (Veáse Fig. 10).

```
for(j=0;j<NUM_CARACT;j++){
for(i=0;i<(N_MOTAS+1);i++){
            do{
            if(matriz[i][j]==matriz[cl][j])
                  similar++;
            cl++;
            }while(cl<(N_MOTAS+1));
            coincidencias[i]=similar;
            cl=0;
            similar=0;
      }

mayor=coincidencias[0];
      pos=0;
      for(i=1;i<(N_MOTAS+1);i++){
            if(coincidencias[i]>mayor){
            mayor=coincidencias[i];
            pos=i;
            }
      pktOut[j]=matriz[pos][j];
}
}
```

Fig. 29 Código de Majority Voting carácter a carácter

Como podemos apreciar, a la hora de recorrer la matriz por columnas lo hacemos hasta la posición final, la cual es el número de caracteres del mensaje de datos. El número de filas, en cambio, es una variable que al principio de cada experimento cambia el propio usuario indicando el número de motas (N_MOTAS). Le sumamos 1 ya que la mota receptora tiene un mensaje recibido directamente del emisor que también se tiene en cuenta a la hora de reconstruir el paquete.

A continuación, comparamos el primer carácter del primer paquete con todos los primeros caracteres del resto de paquetes e incluiremos en otro vector el número de veces que esta repetido. Seguidamente volveremos a realizar el mismo procedimiento con el primer carácter del segundo paquete y así sucesivamente con el resto.

Al finalizar este procedimiento obtenemos un vector con el número de veces que esta repetido cada carácter de la primera posición de cada paquete, por tanto hemos de recorrer este vector para conseguir cual es el carácter mayoritario. En el caso de empate, se escogerá como mayoritario el que esté repetido primero.

Por ejemplo, una posible matriz donde el mensaje de prueba podría ser “Hello World TF” en un escenario de una mota receptora, una mota emisora y dos nodos intermedios se puede ver en la Fig. 30:

H	E	m	L	L		W	O	R	U	D		T	T
U	E	L	L	O		h	L	R	L	D		T	F
H	I	L	O	O		W	O	r	L	D		T	F

Fig. 30 Ejemplo de matriz para reconstruir paquete

En este caso, compararíamos el primer carácter del primer mensaje (H) con el primer carácter del segundo y tercer mensaje (U, H). Como este carácter esta repetido dos veces, escribiremos este número en la primera posición de un vector adicional. A continuación comparamos el segundo carácter con el resto, y como éste esta repetido solo una vez, escribiremos este valor en la segunda posición del vector adicional. Para finalizar comparamos el tercer carácter con el resto y vemos que esta repetido dos veces, por lo que copiaremos este valor en la tercera posición del vector. En el caso que hubiera más de dos nodos intermedios se seguiría el mismo proceso sucesivamente.

Debido a este proceso, obtendríamos el vector adicional completado de la forma que se aprecia en la Fig. 31:

2	1	2

Fig. 31 Ejemplo vector para decidir cual es el valor predominante

Al recorrer este vector vemos que hay empate, lo cual es evidente, ya que como mínimo habrá empate tantas veces como la letra mayoritaria esté repetida en la matriz. A la hora de decidir cual es la letra predominante, como hemos explicado con anterioridad, nos quedaremos con la que está repetida primero en el vector, por lo cual se decide que ha ganado por mayoría el carácter que esta en la primera fila (H).

A continuación se realiza el mismo proceso con las demás columnas de la matriz, para poder obtener un mensaje final reconstruido.

2.2.3.2. Bit a bit

Más adelante se ha desarrollado una técnica de mayoría absoluta bit a bit, más compleja de implementar pero más eficiente que la anterior.

```
for(j=0;j<NUM_CARACT;j++){
      for (n=0;n<8;n++){
            for (i=0;i<(N_MOTAS+1);i++){
                  aux=1<<n;
                  if ((matriz[i][j]&aux)==0)
                        cont0++;
                  else
                        cont1++;
            }
            if (cont0<cont1)
                  pktOut[j]=pktOut[j]|aux;
            cont0=0;
            cont1=0;
      }

}
```

Fig. 32 Código de Majority Voting bit a bit

En este caso volvemos a tener una matriz igual a la del caso anterior, pero tal y como se puede apreciar en la Fig. 32 el código es substancialmente diferente.

Se vuelve a recorrer la matriz hasta la posición final, dependiendo del número de caracteres que tenga el texto del contenido, y también las columnas hasta N_MOTAS + 1 por lo mencionado anteriormente, pero incluimos un nuevo recorrido de 8 posiciones, ya que cada carácter esta formado por 8 bits y en este experimento queremos ir comprobando bit a bit.

Para esta comprobación las líneas de código centradas en esto son:

```
aux=1<<n;
if ((matriz[i][j]&aux)==0)
```

Fig. 33 Parte del código Majority Voting bit a bit

Es decir, entramos en el primer bit de datos del primer paquete, y miramos si es un '0' o un '1'. Según lo que sea lo apuntamos en un contador correspondiente. Luego vamos al primer bit del segundo paquete y hacemos lo mismo. Así con todos los paquetes que tengamos. Una vez el primer bit de cada paquete haya sido contado, miramos el contador y los comparamos, el que sea mayor será el '0' o '1' que escribiremos en el primer bit del paquete reconstruido. A la hora de inicializar la variable donde guardamos el paquete que estamos reconstruyendo, lo inicializamos todo a '0', por tanto sólo habrá que escribir cuando gane el contador de '1'. Este mismo proceso se va haciendo con todos los bits del paquete.

Para poder saber si un bit es 0 o 1, hemos creado una variable auxiliar donde escribimos un 1 movido n posiciones según la posición que tenga el bit que nos interesa saber. Una vez tenemos esto, hacemos una suma lógica con el bit que queremos. Según la tabla siguiente sabemos si es 0 o 1.

Entrada A	Entrada B	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Tabla. 1 Tabla de la verdad de la puerta AND

Para esta figura consideramos que la entrada A es el bit que está escrito en la posición *n.* En cambio, la entrada B siempre será 1, ya que en nuestro código indicamos que la variable auxiliar es un 1 movido *n* posiciones. Como podemos comprobar, en nuestro caso sólo nos interesa la segunda y cuarta fila de la tabla. De esta manera podemos comprobar si la entrada es un 0 la salida seguirá siendo 0 y si es un 1 seguirá siendo 1 también. Esta salida, será comparada con un 0, para si es igual incrementar el contador de 0, o si por el contrario es diferente, aumentar el contador de 1.

Una vez ya sabemos si el bit que gana por mayoría es 0 o 1 hemos de escribirlo en la variable en la cual guardamos el paquete final reconstruido, tal como se puede apreciar en las siguientes líneas de código.

```
if (cont0<cont1)
pktOut[j]=pktOut[j]|aux;
```

Fig. 34 Parte del código de majority voting bit a bit

La variable 'pktOut' está inicializada a 0, por lo que sólo escribiremos cuando gane el contador de '1'. Para realizar dicha operación utilizamos la variable 'aux' y el signo de operación OR.

Entrada A	Entrada B	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Tabla. 2 Tabla de la verdad de la puerta OR

En este caso sólo nos interesa la segunda fila ya que de entrada A siempre será 0 porque se ha inicializado así y la entrada B es '1'. Por tanto la tabla OR es la indicada, debido a que la salida es uno tal y como nos interesa.

Una vez finalizado todo este procedimiento, obtenemos la variable 'pktOut' con el mensaje reconstruido listo para ser enviado al ordenador y ser mostrado por pantalla.

Todo este procedimiento se puede ver resumido en la

Fig. 35.

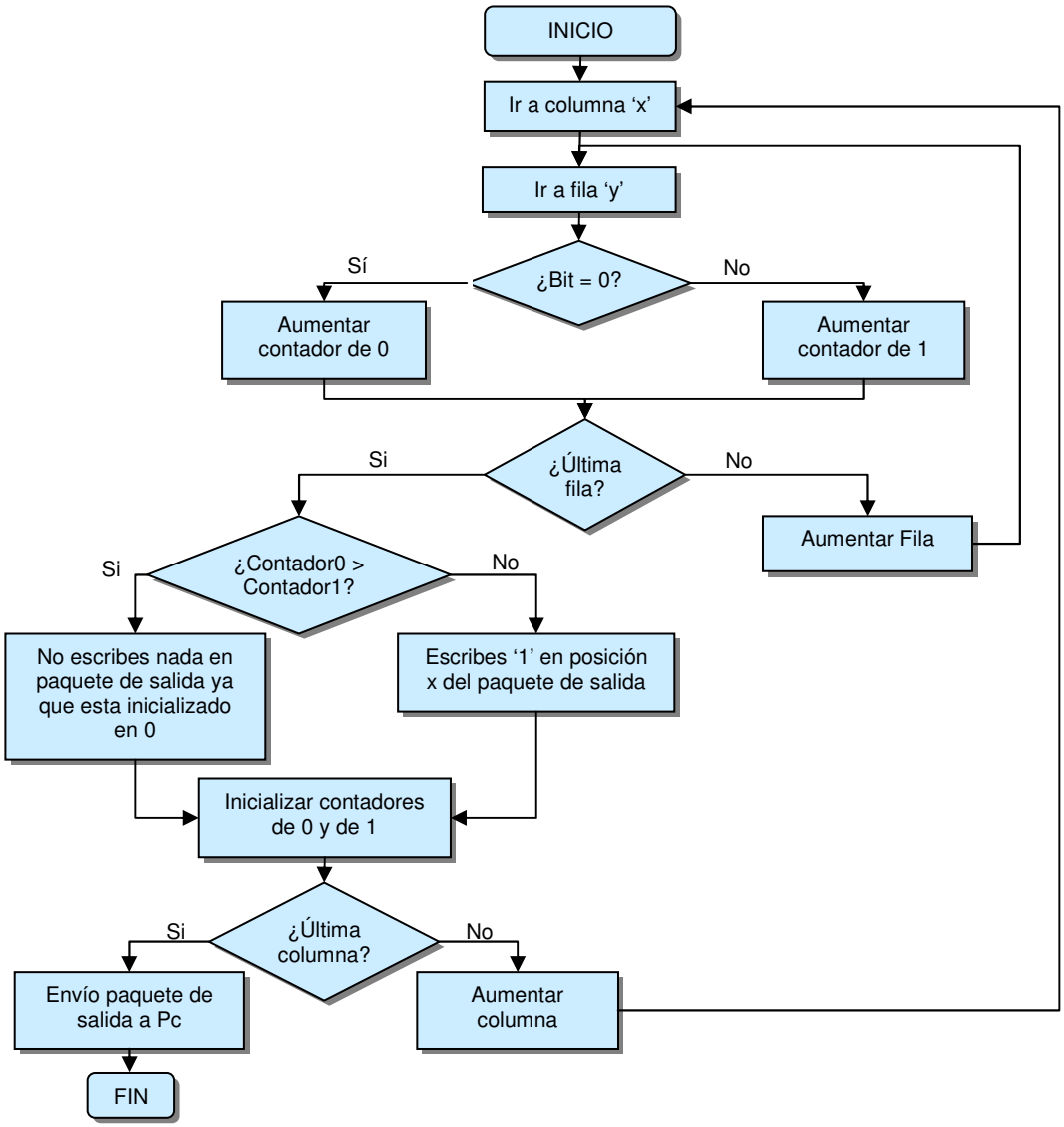


Fig. 35 Diagrama de flujo de la escritura del majority Voting bit a bit

2.2.4. Algoritmos

Como ya se ha mencionado en el apartado 2.1.3, en nuestro escenario existen tres tipos de motas, cada una de las cuales tiene un funcionamiento y un código distinto, según la función que desempeñan.

A continuación, se van a describir el funcionamiento y el código de cada mota, con tal de poder entender la función que cada una desempeña. También se van a explicar los tipos de mensajes que envía cada mota, para de poder comunicarse con las demás.

2.2.4.1. Mota receptora o estación base

Los tipos de mensaje que utiliza la mota receptora o estación base para comunicarse son tres. El primero de ellos es el mensaje nombrado 'Wsn_Aplication_Msg', el cual se encarga de enviar la información por el USB a la aplicación Java y viceversa. El segundo tipo de mensaje recibe el nombre de 'PacketLinkMsg'. Este mensaje es el conocido como datos, y es el enviado desde la mota emisora a la mota receptora. Por último, el tercer mensaje recibe el nombre de 'PacketSynchroMsg' y es el paquete de sincronismo, el cual es utilizado, entre otros, para enviar los datos que introduce el usuario final a la mota emisora para ajustar los valores de la transmisión de datos.

```
typedef nx_struct Wsn_Aplication_Msg {
  nx_uint8_t power;
  nx_uint8_t rstRssi;
  nx_uint32_t transmitted;
  nx_uint32_t received;
  nx_uint8_t retryDelay;
  nx_uint8_t retries;
  nx_uint8_t iniBackoff;
  nx_int8_t rssiVal;
  nx_uint8_t congBackoff;
  nx_uint16_t sync;
  nx_uint8_t mote_id;
  nx_uint8_t datarx[14];
  nx_uint16_t npok;
  nx_uint16_t errores;
  nx_uint16_t contador;
  nx_uint16_t errbase;
} Wsn_Aplication_Msg_t;
```

Fig. 36 Estructura mensaje 'Wsn_Aplication_Msg'

Para poder empezar cada experimento, el usuario final ha de introducir un cierto número de datos a la aplicación Java para fijar los valores de transmisión.

En la Fig. 36 se muestran todos los datos que se envían dentro del paquete 'Wsn_Aplication_Msg' donde los más importantes para nuestro experimento son:

- Power: Se indica la potencia a la cual serán transmitidos los paquetes.
- Transmitted: Indica el número de paquetes transmitidos
- Received: Indica el número de paquetes recibidos
- Retries: Es un campo en el cual introducimos diferentes valores para diferenciar diferentes tipos de mensajes de sincronismo. Valdrá 0 si el campo introducido en el campo datarx es el recibido directamente en el receptor, sin errores. Valdrá 41, si el campo datarx es el valor del paquete reconstruido por el 'Majority Voting' y valdrá 42 si el mensaje recibido directamente al receptor contenía errores y no ha sido posible reconstruirlo.
- Datarx: En este campo se introduce el mensaje reconstruido por el mecanismo de mayoría absoluta para ser mostrado por pantalla en la aplicación.
- Npok, errores, contador y errbase: Son variables, las cuales son utilizados como contadores. Nos serán de gran utilidad más adelante, para realizar medidas y estadísticas del comportamiento del sistema.

Después que el usuario final haya seleccionado los parámetros, se guardan en los campos anteriores y se envían a través del USB a la mota receptora.

Una vez este paquete llegue a la mota receptora, la información anterior se guarda dentro de un mensaje de sincronismo y se envía por vía broadcast y a través del medio radio a las demás motas, tal como se puede ver en la Fig. 37:

```
event message_t *serialReceive.receive[am_id_t id](message_t *msg,
void *payload, uint8_t len)
      {
            serialReceivedMsg = (Wsn_Aplication_Msg_t*) payload;
            PacketSync =(PacketSynchroMsg *) call
radioSend.getPayload(&myMsg);
            radioDest=serialReceivedMsg->mote_id;
            PacketSync->flag=serialReceivedMsg->sync;
            PacketSync->power=serialReceivedMsg->power;
            PacketSync->retries=serialReceivedMsg->retries;
            PacketSync->retryDelay=serialReceivedMsg->retryDelay;
            PacketSync->congBackoff=serialReceivedMsg->congBackoff;
            PacketSync->iniBackoff=serialReceivedMsg->iniBackoff;
            PacketSync->transmitted=serialReceivedMsg-
>rssiLargestValue;
            Received[radioDest]=0;
            Transmitted[radioDest]=0;
            post sendToRadio();
            return msg;
      }
```

Fig. 37 Código envío del paquete de sincronismo

La estructura del paquete de sincronismo 'PacketSynchroMsg' es diferente a la anterior, y en este caso esta definida de la siguiente forma:

```
typedef nx_struct PacketSynchroMsg {
  nx_uint16_t flag;
  nx_uint16_t transmitted;
  nx_uint8_t power;
  nx_uint8_t retries;
  nx_uint8_t retryDelay;
  nx_uint8_t iniBackoff;
  nx_uint8_t congBackoff;
  nx_uint8_t end;
} PacketSynchroMsg;
```

Fig. 38 Estructura mensaje 'PacketSynchroMsg'

Donde los campos más destacados son:

- Power: Se indica la potencia a la cual serán transmitidos los paquetes.
- Retries: Es un campo en el cual introducimos diferentes valores para diferenciar diferentes tipos de mensajes de sincronismo. Valdrá 0 cuando sea un mensaje de sincronismo convencional, es decir, enviar los valores predefinidos por el usuario para realizar la transmisión. Por el contrario, valdrá 41 cuando el mensaje de sincronismo se utilice para pedir el reenvío del paquete con número de secuencia N, el cual ha llegado con errores a la mota receptora.
- End: Indica que es el último paquete de la transmisión

Después de enviar el mensaje de sincronismo al resto de motas, la mota receptora espera recibir los paquetes que la mota emisora vaya a enviar, tal como se puede ver en la Fig. 12.

```
event message_t *Receive.receive(message_t *msg, void *payload,
uint8_t len)
```

Fig. 39 Evento de recepción de mensajes

En este caso, los paquetes enviados serán del tipo 'PacketLinkMsg' y también tendrán una estructura de paquete diferente, tal como se muestra a continuación:

```
typedef nx_struct PacketLinkMsg {
  nx_uint32_t count;
  nx_uint32_t loss;
  nx_uint16_t src;
  nx_uint8_t cmd;
  nx_uint16_t sync;
  nx_uint8_t datatx[14];
} PacketLinkMsg;
```

Fig. 40 Estructura del mensaje 'PacketLinkMsg'

En este paquete, los campos más importantes son:

- Count: Indica el emisor del paquete
- Datatx: Aquí se introduce el mensaje de datos que envía el emisor al receptor. Por limitación de las motas tan solo es posible enviar 28 bytes. Como cada carácter son 2 bytes, los caracteres máximos son 14. En nuestro proyecto utilizamos como mensaje de prueba 'Hello World TF'.

Una vez recibe un paquete de datos, la primera comprobación que realiza es ver si el mensaje contiene errores, ya que si es así, deberá utilizar el mecanismo de votación por mayoría. Para comprobar esto, compara si lo que recibe en el campo datatx del mensaje 'PacketLinkMsg' es igual al mensaje de prueba que se espera recibir. (Véase Fig. 21).

Si el campo de datos no tiene errores, se envía un mensaje del tipo 'Wsn_Aplication_Msg por USB a la aplicación, para mostrarlo por pantalla. Para eso, se debe copiar la información necesaria en ese mensaje, rellenando el campo retries a 0, ya que el mensaje es correcto directamente en recepción. En el campo datarx se copiará el contenido del mensaje 'PacketLinkMsg' (en nuestro código esta guardado en la variable datapc).

```
rsm->rssiLargestValue = largest;
rsm->mote_id=source;
rsm->rssiVal=rssi;
rsm->rstRssi=1;
rsm->transmitted=Transmitted[source];
rsm->received=Received[source];
strcpy(rsm->datarx, datapc);
rsm->retries=0;
rsm->npok=npok;
rsm->errores=errores_total;
rsm->contador=contador_total;
rsm->errbase=errores_base;

call AMSend.send(AM_BROADCAST_ADDR, &packet,
sizeof(Wsn_Aplication_Msg_t))
```

Fig. 41 Relleno paquete 'Wsn_Aplication_Msg y envío a la aplicación Java

En cambio, si el paquete recibido en recepción contiene errores, el procedimiento es distinto, ya que en este caso, se ha de reconstruir el paquete mediante el 'Majority Voting'.

Por tanto, si al comparar el campo de datos recibido en el mensaje de datos (datatx), no es igual al mensaje de prueba, este dato se guarda en una matriz de (Numero de nodos + 1) filas y 15 columnas. Al recibirse directamente del emisor siempre será guardado en la primera fila. A continuación se envía un mensaje de sincronismo vía broadcast solicitando el reenvió de dicho paquete.

En dicho mensaje de sincronismo, ponemos el campo de retries a 41, para identificar que es un mensaje de sincronismo de petición de paquete. Como enviamos el mensaje en modo broadcast, y en realidad no nos interesa que el

emisor nos vuelva a enviar el paquete, con este campo en 41, descartará el mensaje.

```
((PacketSynchroMsg*)call radioSend.getPayload(&myMsg))->retries=41;
call radioSend.send(radioDest, &myMsg, sizeof(PacketSynchroMsg)
```

Fig. 42 Relleno campo retries y envío del mensaje

A continuación de enviar el paquete de sincronismo, se reciben los mensajes reenviados por los nodos, guardándolos también en la matriz anterior. Por ejemplo, si el mensaje lo recibimos desde el nodo intermedio 2, lo guardaremos en la fila 2, y así sucesivamente. Por tanto, si tenemos un escenario donde hay 2 nodos intermedios, la matriz quedaría tal y como se ve en la Fig. 43:

H	E	m	L	L		W	O	R	U	D		T	T	1
U	E	L	L	O		h	L	R	L	D		T	F	1
H	I	L	O	O		W	O	r	L	D		T	F	1

Fig. 43 Ejemplo de matriz de un posible primer paquete erróneo

Una vez rellenada la matriz, comprobamos si la última columna tiene el mismo número, es decir, tienen el mismo número de secuencia, ya que en esa posición se copia el número de secuencia de cada paquete. Esto debe ser así, ya que sino no realizaríamos la reconstrucción correcta, sino que estaríamos mezclando diferentes transmisiones.

Por tanto, si los números de secuencia de la matriz son iguales, se hará el mecanismo de 'Majority Voting' tal como se explicó en el apartado 2.2.3. A continuación, se copiará el resultado obtenido (guardado en la variable datapc) en la variable datarx del mensaje 'Wsn_Aplication_Msg'. También pondremos el retries del mismo mensaje a 41, para indicar que el mensaje ha sido reconstruido. Por último, lo enviaremos a través del USB a la aplicación, con tal de mostrarlo por pantalla, tal como se puede apreciar en la Fig. 44:

```
rsm->rssiLargestValue = largest;
rsm->mote_id=source;
rsm->rssiVal=rssi;
rsm->rstRssi=1;
rsm->transmitted=Transmitted[source];
rsm->received=Received[source];
strcpy(rsm->datarx, datapc);
rsm->retries=41;
rsm->npok=npok;
rsm->errores=errores_total;
rsm->contador=contador_total;
rsm->errbase=errores_base;

call             AMSend.send(AM_BROADCAST_ADDR,             &packet,
sizeof(Wsn_Aplication_Msg_t))
```

Fig. 44 Relleno paquete 'Wsn_Aplication_Msg y envío a la aplicación Java

En cambio, si los números de secuencia no coinciden o no han llegado todos los reenvíos esperados por parte de las motas, el mensaje no se deberá reconstruir. En este caso, se enviará el mensaje por USB a la aplicación, con el valor retries a 42 y en el campo de datos se indicará que no se ha podido reconstruir el mensaje.

Además esta mota también es la encargada de comunicarle a la mota emisora el momento en el que ya no debe enviar más paquetes, enviándole una solicitud para ello.

En nuestro caso particular, nos interesa recibir siempre el mismo número de paquetes erróneos para poder estudiar mejor el sistema y obtener así unos buenos resultados. Por lo tanto, en nuestro experimento al recibir el paquete con el campo '*errores_base*' deseado, se enviará un mensaje del tipo 'PacketSynchroMsg' a la mota emisora con el campo 'end' a 20, ya que este campo es el destinado a detener el envío de mensaje, tal como se explicará más adelante en el apartado 2.2.4.2.

```
event message_t *Receive.receive(message_t *msg, void *payload,
uint8_t len)
{
      if(errores_base>=250)
      {
            ((PacketSynchroMsg*) call radioSend.getPayload(&myMsg))-
>retries=0;
            ((PacketSynchroMsg*) call radioSend.getPayload(&myMsg))-
>end=20;
            radioDest=1;
            post sendToRadio();
      }
}
```

Fig. 45 Evento detener envío de mensajes

2.2.4.2. Mota emisora

En este tipo de mota, los tipos de mensajes utilizados son 'PacketLinkMsg' y 'PacketSynchroMsg', donde en ambos casos, la estructura del mensaje es igual que la ya explicada en el punto anterior.

Lo primero que recibe la mota emisora es un mensaje de sincronismo enviado por la mota receptora, incluyendo los datos que el usuario final ha introducido para fijar los valores de transmisión.

```
event message_t *Receive.receive(message_t *msg, void *payload,
uint8_t len) {
      PacketSynchroMsg *SyncMsg = (PacketSynchroMsg *) payload;
            if (SyncMsg->retries!=41){
                  parar=SyncMsg->end;
                  total=SyncMsg->flag;
                  power=SyncMsg->power;
                  timerPeriod=SyncMsg->transmitted;
                  call CC2420Packet.setPower(&myMsg, power );
                  call PacketLink.setRetries(&myMsg, SyncMsg-
>retries);
                  call PacketLink.setRetryDelay(&myMsg, SyncMsg-
>retryDelay);
                  atomic {
                        iniBackoff=SyncMsg->iniBackoff;
                        congBackoff=SyncMsg->congBackoff;
                         }

                  call Timer.startPeriodic(timerPeriod);
                  return msg;
            }
}
```

Fig. 46 Código de recepción de un paquete de sincronismo

Como se puede comprobar en la Fig. 46, lo primero que se comprueba al recibir un mensaje de sincronismo es el campo 'retries', ya que si vale 41 significará que es un mensaje de solicitud de reenvío de paquete con errores, y que por tanto, la mota emisora deberá descartar.

En cambio, si el campo 'retries' no vale 41, quiere decir que es un mensaje de sincronismo convencional, y por tanto, copia en variables locales los parámetros que ha introducido el usuario final.

A continuación, lanza un temporizador que se repite cada timerPeriod. Es decir, cuando expire dicho timerPeriod llama a la tarea de enviar mensajes y automáticamente empieza de nuevo el mismo temporizador. Por lo cual, cada timerPeriod enviará un mensaje y no parará de hacerlo hasta que no le llegue una petición para ello.

```
event void Timer.fired()
      {
            post send();
      }
```

Fig. 47 Evento timer expirado

```
task void send()
      {
            strcpy(((PacketLinkMsg *) call
AMSend.getPayload(&myMsg))->datatx, resultado);
            sendMsg=PLINK;

            ((PacketLinkMsg *) call AMSend.getPayload(&myMsg))-
>count=transmitted;
            if(call AMSend.send(AM_BROADCAST_ADDR, &myMsg,
sizeof(PacketLinkMsg)) != SUCCESS)
            {
                  post send();
            }
      }
```

Fig. 48 Tarea envío de mensajes

Una vez se ha llamado a la tarea de enviar un mensaje copiamos en el campo 'datatx' del paquete 'PacketLinkMsg' el contenido de los datos que queremos enviar. Estos datos están guardados en una variable local llamada 'resultado'. A continuación también copiamos en el campo 'count' del mismo mensaje el valor de la variable transmitted, ya que será nuestro número de secuencia. Finalmente, enviamos el mensaje vía broadcast al resto de las motas, tal como se puede apreciar en la Fig. 48.

La variable 'transmitted' o número de secuencia en cada experimento se inicializa previamente a 0 y se va aumentado un valor cada vez que se envía un paquete con éxito.

```
event void AMSend.sendDone(message_t *msg, error_t error) {
            if(sendMsg==PLINK)
                  transmitted++;
}
```

Fig. 49 Evento de un envío con éxito

Esta tarea de envío se repite cada timerPeriod y no se detiene hasta que le llega una solicitud para detenerse, es decir, el campo 'end' del paquete 'PacketSynchroMsg' posterior vale un número particular, en este caso 20.

Cuando esto sucede así, tal como se aprecia en la Fig. 46 se copia dicho valor en una variable local, llamada 'parar'. Cada vez que la tarea send se realiza con éxito, a parte de aumentar el número de secuencia, se comprueba si esta variable tiene dicho valor, ya que si es cierto se parará el temporizador, y por tanto el experimento.

```
event void AMSend.sendDone(message_t *msg, error_t error) {
      if (parar==20)
      {
            call Timer.stop();
      }
}
```

Fig. 50 Evento de un envío con éxito

2.2.4.3. Nodo intermedio

La mota que realiza la función de nodo intermedio o vecino utiliza los mismos tipos de mensaje que la mota emisora, 'PacketLinkMsg' y 'PacketSynchroMsg', incluyendo los mismos campos ya comentados anteriormente en el apartado 2.2.4.1.

La mota nodo intermedio o vecino para empezar recibe un mensaje de sincronismo, como en el caso anterior, para configurar sus parámetros de envío en caso que tenga que reenviar algún paquete más adelante.

A continuación va recibiendo los mensajes de datos que la mota emisora envía a la mota receptora, ya que ésta los envía a través de vía broadcast. Como estos mensajes no tienen como destino final esta mota, lo único que hace es guardarse en un buffer de una posición el paquete de datos, con tal de reenviarlo si le llega un paquete de solicitud para ello. (Véase Fig. 23).

```
event message_t *Receive.receive(message_t *msg, void *payload,
uint8_t len)
{
            if (len==sizeof(PacketLinkMsg))
            {
                  radioQueue[radioIn] =*msg;
            }
}
```

Fig. 51 Evento recibir mensaje en mota nodo intermedio o vecino

En el momento que llegue a la mota receptora un mensaje que contenga errores, ésta enviará un mensaje de sincronismo broadcast a todas las motas con tal de pedir el reenvío de este mensaje para intentar reconstruirlo en uno correcto. Para diferenciar que este mensaje de sincronismo es de solicitud de reenvío y no un mensaje de sincronismo habitual se guardará en el campo retries del paquete 'PacketSynchroMsg' el valor 41.

Cuando este mensaje llegue a la mota con función de nodo intermedio o vecino, ésta debe comprobar que realmente se trata de este tipo de mensaje, comprobando que el mensaje le llega desde el receptor (es decir, su origen es 0) y que el campo 'retries' vale 41, como se puede apreciar en la Fig. 52. Además este mensaje deberá tener el mismo número de secuencia que el mensaje con error que tiene la mota receptora, por ello esto será comprobado en la mota receptora, tal como se explico en el apartado 2.1.3.1.

```
event message_t *Receive.receive(message_t *msg, void *payload,
uint8_t len)
{
      if (len==sizeof(PacketSynchroMsg))
      {
            source = call AMPacket.source(msg);
            if(source==0){
                  PacketSynchroMsg *SyncMsg = (PacketSynchroMsg *)
payload;
                        if (SyncMsg->retries == 41){
                        post send();
                        }
            }
      }
}
```

Fig. 52 Evento recibir mensaje en mota nodo intermedio o vecino

Después de haberse comprobado que se trata de una petición de reenvío, las motas con función de nodos intermedios deberán enviar el mensaje guardado en su buffer a la mota receptora, tal como se observa en la Fig. 53.

```
task void send()
{
msgx = radioQueue[radioIn];
if(call AMSend.send(0, &msgx, sizeof(PacketLinkMsg)) != SUCCESS)
      post send(); //Volvemos a llamar a la tarea send().
}
```

Fig. 53 Tarea enviar mensaje de reenvío a la mota receptora

2.3. Aplicación Java

En nuestro caso es necesario el uso de una aplicación Java para poder manipular los datos que proceden de las motas y mostrarlos al usuario final a través de una interfaz gráfica.

2.3.1. Aplicación previa

Para el desarrollo de nuestro proyecto contamos con una aplicación previa, realizada en dos proyectos anteriores, la cual deberá ser adaptada a las necesidades de nuestro trabajo.

La aplicación está desarrollada con eclipse y como todo proyecto Java desarrollado en este entorno, tiene una carpeta principal que engloba todo el proyecto y unas subcarpetas que viene a ser los packages (agrupación de clases Java) de java. En este caso, el proyecto se llama WSNtest, que es el nombre de la capeta global; y se tienen varias subcarpetas, entre ellas src, en

la cual se encuentran la aplicación Java y los ficheros NesC de la aplicación Tinyos que funcionan en la mota base.

Las funciones de las que se encarga esta aplicación son la medición de la RSSI y de la PER en función del número de nodos intermedios que existan en el escenario, distribuida en diferentes paneles. Además el usuario final puede configurar ciertos parámetros, como la potencia de transmisión de cada mota.

2.3.1.1. Panel RSSI

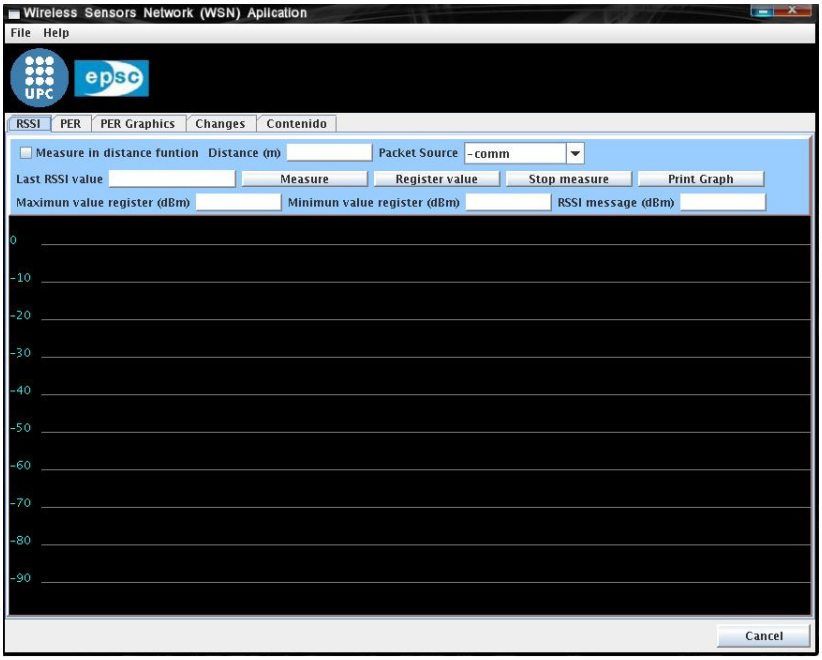


Fig. 54 Panel RSSI

En la Fig. 54 se muestra el Panel RSSI, el cual se utiliza para la medición y visualización de la RSSI, este panel a su vez esta compuesto de otros dos paneles. Uno de ellos es el que contiene la botonera y elementos para introducir datos. El segundo es el monitor, el cual sirve tanto para visualizar la grafica del RSSI en tiempo real como para la visualización de una gráfica de la medida del RSSI en función de la distancia.

Para que la aplicación comience a funcionar se debe pulsar el botón ‘measure’, con el cual se empezará a medir la RSSI.

2.3.1.2. Panel PER

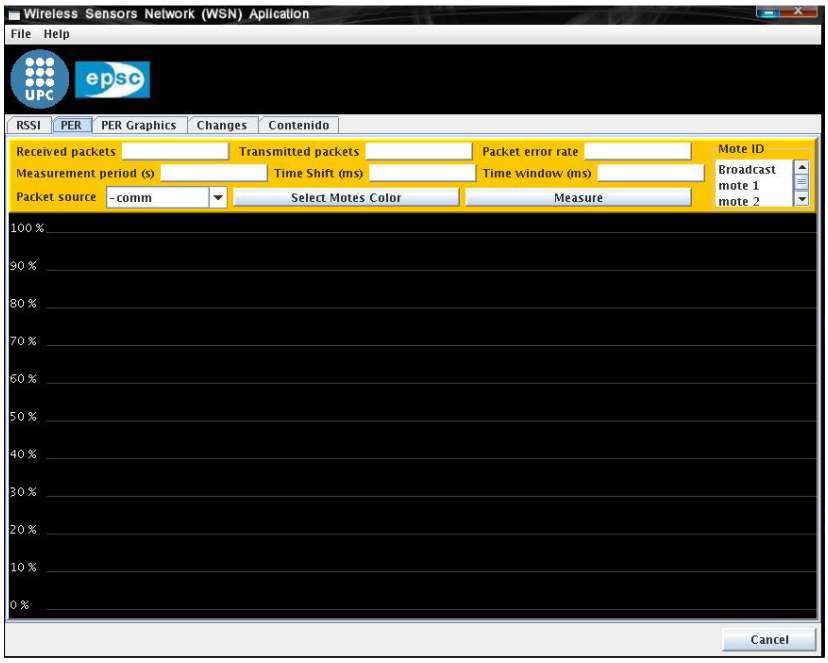


Fig. 55 Panel PER

El panel PER, en cambio, se utiliza para la medición de la probabilidad del error en el paquete (Packet Error Rate). Como el panel anterior también esta compuesto por dos paneles, uno de control y otro de visualización de grafica.

El panel de control dispone de los siguientes elementos:

- ‘Received Packets’: Cuenta el número de paquetes recibidos en la mota receptora.
- ‘Transmitted packets’: Cuenta el número de paquetes enviados a la mota receptora.
- ‘Packet error rate’: Es la tasa de error de paquete instantánea de la mota seleccionada. Al final de cada experimento muestra el valor de PER de la mota seleccionada.
- ‘Measurement Period’: En este campo se debe introducir el tiempo en segundos que se desea que dure la medición de la PER.
- ‘Time shift’: Es el tiempo que tarda en inicializarse cada ventana temporal de medición de la PER, después de haberse inicializado la anterior.
- ‘Time window’: Es el tiempo deseado de duración de una ventana de medición temporal de la PER.

Una vez rellenados los parámetros deseados el usuario debe pulsar el botón ‘Measure’ para empezar el experimento.

2.3.1.3. Panel ‘Changes’

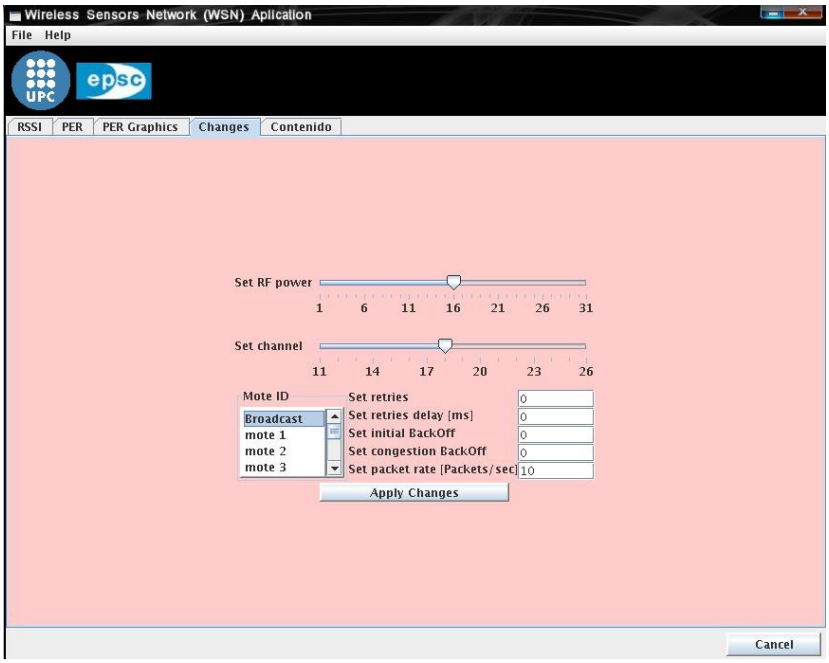


Fig. 56 Panel de cambios

Este panel es un panel creado para la modificación de ciertos parámetros de comunicaciones para la mota o motas que previamente se seleccionan, como por ejemplo la potencia de transmisión.

Como se puede observar en la Fig. 56 los elementos implementados son:

- 'Mote ID': Este elemento grafico se utiliza para seleccionar las motas, a las cuales se les va a cambiar algún parámetro. Se puede seleccionar una mota en concreto o varias a la vez. Pero si queremos que todas las motas envíen a la misma potencia de transmisión se deberá poner broadcast.
- 'Set RF Power': Sirve para modificar el nivel de transmisión de la mota seleccionada, según la Tabla. 3.

Nivel de transmisión	Potencia de salida (dBm)	Potencia consumida de batería (mW)
31	0	31.3
27	-1	29.7
23	-3	27.4
19	-5	25
15	-7	22.5
11	-10	20.2
7	-15	17.9
3	-25	15.3

Tabla. 3 Tabla de equivalencia de nivel de transmisión con potencias

Una vez el usuario final ha introducido todos los parámetros deseados de configuración debe pulsar el botón 'Apply Changes' para aplicarlos en las motas seleccionadas.

2.3.2. Modificaciones

Para el desarrollo de nuestro proyecto se ha modificado la aplicación java, creando un nuevo panel y modificando los anteriores con tal de cubrir las necesidades actuales.

A continuación, se explica el panel PER, ya que es el que ha sufrido las modificaciones principales y el nuevo panel implementado.

2.3.2.1. Panel PER

La primera modificación realizada ha sido a la hora de insertar los valores de configuración, ya que anteriormente en el campo 'Measurement Period' introducíamos el tiempo en segundos que deseábamos que durara la medición de la PER. En cambio, para nuestro proyecto no nos interesa mostrar la gráfica de la PER en función del tiempo, sino que nos interesa mostrarla en función del número de paquetes erróneos en la base receptora. Es decir, el experimento durará hasta que se reciba un cierto número de paquetes erróneos. Este número no se fijará en dicha aplicación java, sino que previamente se ha realizado en el código NesC, como hemos explicado en el apartado 2.2.4.1.

En este panel hemos tenido que añadir nuevo código con tal de poder mostrar la información importante para nuestro proyecto. Esta información será mostrada en el debug de java ya que, nos será de gran utilidad para obtener los resultados finales. En cambio, los datos recibidos también serán mostrados en otro panel, ya que realmente ésta es la información que le interesa visualizar al usuario final.

La información importante que queremos mostrar por pantalla para realizar a posteriori un estudio del sistema son:

- Paquetes recibidos
- Paquetes transmitidos
- PER
- Datos enviados por el emisor
- Datos recibidos por el receptor,
- Número de secuencia
- Diferentes contadores de paquetes erróneos y reconstruidos

Estas variables se envían a la aplicación Java a través del mensaje 'Wsn_Aplication_Msg', tal como se ha explicado en el apartado 2.2.4.1.

Una vez se recibe el mensaje en la aplicación Java hemos de leer dichos valores. Previamente ya leía algunos de ellos, como por ejemplo el número de paquetes recibido, por tanto hemos tenido que modificar el código para añadir los nuevos valores.

2.3.2.2. *Panel Contenido*

El nuevo panel Contenido se ha diseñado para mostrar por pantalla los datos que se reciben en la mota receptora.

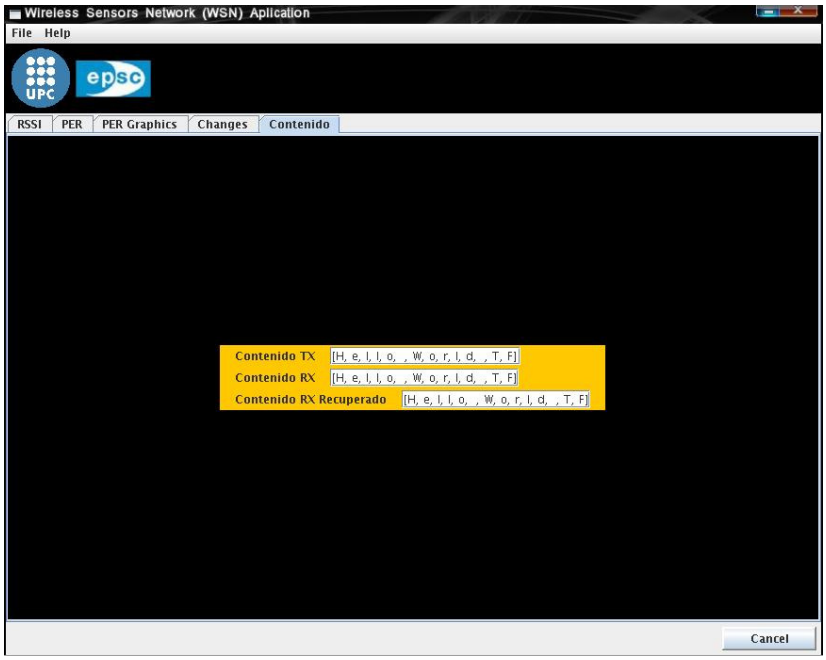


Fig. 57 Panel Contenido

Como se puede observar en la Fig. 57, se muestran 3 tipos de datos por pantalla. El primero de ellos son los datos que la mota emisora envía, los cuales en nuestra prueba fueron 'Hello World TF' Esto se muestra para poder comparar mejor si el paquete recibido contiene errores o no.

Este valor no se obtiene directamente de la mota emisora, sino que se introduce manualmente en Java, ya que a priori sabemos que éste será el mensaje enviado.

El segundo tipo son los datos recibidos directamente del receptor, por lo cual podremos visualizar si éste contiene errores, comparándolo con el anterior.

En el caso que contenga errores se llama a la función de mayoría absoluta, la cual una vez finalizado todo el proceso nos envía el mensaje reconstruido. Este último es el mostrado en el tercer recuadro.

Si este último recuadro es igual al primero, se puede afirmar que el mensaje se ha reconstruido con éxito. Sin embargo, si no es igual, el mensaje se ha podido

reconstruir pero sin éxito. También es posible que no se pueda realizar la reconstrucción, por lo cual, en este caso aparecerá un mensaje indicando lo ocurrido.

2.3.2.3. Código del debug

Además de realizar el nuevo panel, explicado en el apartado 2.3.2.2, también hemos creado un nuevo código para mostrar por pantalla toda la información útil a la hora de realizar las pruebas y poder analizarlas. Esta información aparece en la pantalla del debug de Java, ya que ésta no es necesaria para el usuario final, sino solamente para estudiar el funcionamiento del sistema.

Un ejemplo de la información que se muestra por pantalla cuando un paquete se reconstruye correctamente se ve reflejada en la Fig. 58:

```
0
Los paquetes recibidos ok (destination ok or reconstruct) son 118
El #paquetes total con errores (relays+destination) son 3
El #paquetes total recibidos (relays+destination) son 126
El #paquetes total con errores en la base son 3
Recibo del emisor: [v, g, U, b, e, f, g, U, V, U, u, W, %, V] del
paquete 124

41
Los paquetes recibidos ok (destination ok or reconstruct) son 119
El #paquetes total con errores (relays+destination) son 3
El #paquetes total recibidos (relays+destination) son 128
El #paquetes total con errores en la base son 3
Recuperado con las motas: [H, e, l, l, o,  , W, o, r, l, d,  , T,
F] del paquete 124
```

Fig. 58 Ejemplo del debug de un paquete reconstruido con éxito

En la aplicación fijamos un identificador para distinguir si el mensaje llega del emisor directamente, o bien es el paquete reconstruido que envía la mota receptora. Por tanto, cuando el paquete llega sin errores la aplicación Java muestra el número 0, ya que éste es su identificador. En cambio, cuando se trata del paquete reconstruido el identificador seleccionado es 41.

En la Fig. 58 se puede ver la información del paquete que recibe la mota receptora directamente desde el emisor, ya que viene precedido del número 0. En este caso vemos que este paquete contiene errores, ya que los datos no son como el paquete de prueba, en este caso concreto 'Hello World TF'. Debido a esto, aumenta el contador de paquetes con errores en base, el número total de paquetes erróneos y el número total de paquetes recibidos. Por el contrario, no aumenta el contador de paquetes recibidos correctamente, ya que este paquete contiene errores.

A continuación se muestra la información que envía la mota receptora a través del USB al ordenador, con la información que ha reconstruido ésta. Al ser una reconstrucción viene precedida por el número 41.

En este caso, se puede observar que el contador de paquetes recibidos correctamente se ha incrementado una unidad, ya que el paquete ha sido reconstruido con éxito. También se puede apreciar que el contador de paquetes totales recibidos ha aumentado dos unidades, debido a que en este ejemplo había dos nodos intermedios y ambos han reenviado el paquete a la mota receptora.

Para realizar correctamente la reconstrucción del paquete erróneo, la mota receptora previamente ha comprobado que todos los mensajes tengan el mismo número de secuencia, el cual en este caso particular es 124. Si esto no fuera así, la reconstrucción del paquete no sería posible. Para más detalle revisar la sección 2.1.3.1.

También es posible que la mota receptora pueda reconstruir el paquete pero el mensaje de datos resultante no sea el correcto, debido a que las motas vecinas tampoco han recibido los datos correctamente. En esta situación mostraríamos los datos resultantes, aunque no fueran correctos y aumentaríamos el número total de paquetes con errores, en vez del número de paquetes recibidos correctamente. Obviamente también se aumentaría el número de total de paquetes recibidos dos unidades, ya que sigue recibiendo los dos paquetes reenviados desde los dos nodos intermedios.

```
0
Los paquetes recibidos ok (destination ok or reconstruct) son 118
El #paquetes total con errores (relays+destination) son 3
El #paquetes total recibidos (relays+destination) son 126
El #paquetes total con errores en la base son 3
Recibo del emisor: [v, g, U, b, e, f, g, U, V, U, u, W, %, V] del
paquete 124

41
Los paquetes recibidos ok (destination ok or reconstruct) son 118
El #paquetes total con errores (relays+destination) son 5
El #paquetes total recibidos (relays+destination) son 128
El #paquetes total con errores en la base son 3
Recuperado con las motas: [s, A, l, l, o,  , K, o, r, m, d,  , T,
F] del paquete 124
```

Fig. 59 Ejemplo del debug de un paquete reconstruido sin éxito

Para poder comprobar si esta implementación funciona correctamente, se han realizado una serie de experimentos y un estudio de los resultados obtenidos con tal de verificar el grado de mejora del sistema.

CAPÍTULO 3. RESULTADOS OBTENIDOS

Con el fin de certificar el correcto funcionamiento de los planteamientos de nuestro proyecto y evaluar las ventajas que pueden obtenerse de las técnicas de transmisión cooperativa propuestas, hemos desarrollado diferentes escenarios para poder realizar pruebas y experimentos. De este modo se puede verificar y evaluar el funcionamiento del sistema.

Se han definido e implementado diferentes escenarios para comprobar el comportamiento en cada uno de ellos y poder compararlos entre sí para llegar a una conclusión coherente y poder determinar el número idóneo de nodos vecinos en cada situación.

El principal problema que se ha encontrado a la hora de realizar dichas pruebas ha sido encontrar un punto en el cual se pierden bastantes paquetes, ya que el canal radio tiene una zona de cobertura en la cual los paquetes suelen llegar con una alta probabilidad de éxito a su destino. Fuera de esta zona los paquetes no llegan a su destino, pero la zona límite donde algunos paquetes llegan y otros no es difusa y difícil de tener controlada en un experimento donde se quiere ejecutar de forma repetida e intensiva los algoritmos cooperativos. Por tanto, una primera conclusión práctica interesante obtenida de la experimentación es que en un escenario real es difícil encontrar el punto exacto de la zona del límite de cobertura, ya que será un punto en el cual los paquetes tengan más probabilidades de llegar con errores en su contenido.

A la hora de realizar las pruebas se cambiaba rápidamente de una zona de cobertura correcta a otra sin cobertura, por lo cual se ha tenido que ir variando la distancia y la potencia de transmisión de las motas de forma continua, lo que complica y alarga la obtención de resultados estadísticos relevantes.

En el caso particular de nuestro proyecto hemos bajado la potencia de transmisión al mínimo puesto que el espacio del laboratorio del que disponemos no es muy grande. De esta forma el sistema es mucho más sensible a la distancia de unas motas a otras y a los movimientos de las mismas. Además se ha intentado crear algún tipo de interferencia en el canal para forzar más el envío de paquetes erróneos.

3.1. Primer escenario

En el primer escenario hemos realizado varias tandas de mediciones dentro de nuestro laboratorio de investigación variando el número de nodos vecinos con tal de comparar los resultados y obtener el número de nodos intermedios óptimos para que el sistema funcione lo mejor posible.

Las motas que operaban como nodos intermedios o *relays* se han situado todas ellas muy cerca de la mota receptora. En cambio, la mota emisora se ha situado a una cierta distancia de las demás motas, la cual se encuentra lo suficiente alejada para que los paquetes lleguen a la mota receptora con errores en su contenido.

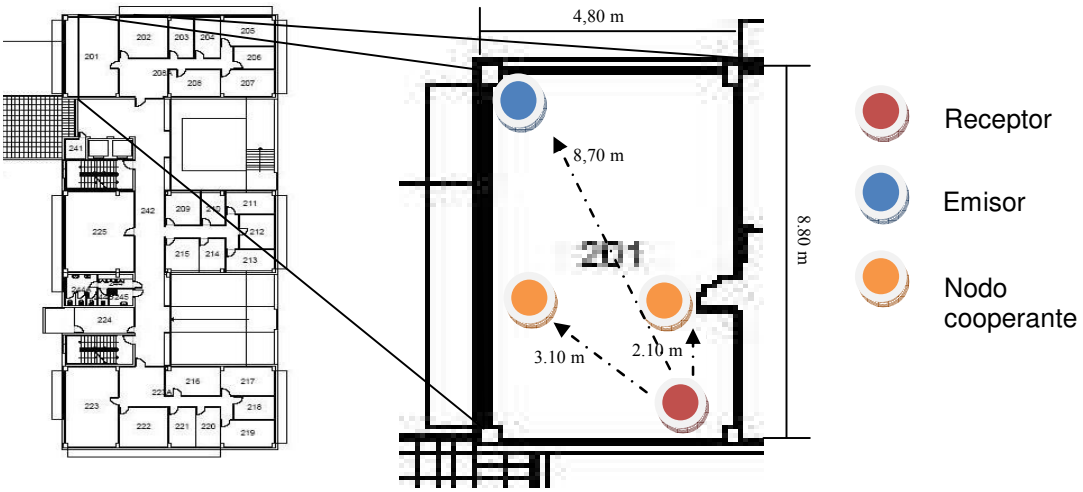


Fig. 60 Situación de las motas en el laboratorio 201 de la torre de profesores de la EPSC

En este escenario se ha fijado que se envíen los paquetes necesarios para que la mota receptora reciba 100 paquetes erróneos y para cada uno de estos paquetes se ejecute el algoritmo cooperativo. Este número ha sido escogido ya que es un número lo suficientemente elevado y no conlleva un tiempo de transmisión real demasiado grande.

Los resultados obtenidos en este escenario han sido:

# Nodos coop	# Paquetes Correctos	# Paquetes Recibidos	%Paquetes Recibidos OK	%Paquetes Recibidos OK Sin coop	# Paquetes Reconstruidos
2	10967	11064	0,9912	0,9910	3
3	10052	10147	0,9906	0,9901	5
4	11345	11435	0,9921	0,9913	10
5	10793	10888	0,9913	0,9908	5
6	9918	10017	0,9901	0,9900	1

Tabla. 4 Resultados medidas del primer escenario (I)

# Nodos coop	# Paquetes Erroneos totales	# Paquetes Recibidos totales	# Lanzamientos del Majority Voting	% Éxito del Majority Voting
2	214	11295	87	3,4483
3	321	10528	78	6,4103
4	553	11990	62	16,1290
5	437	11613	43	11,6279
6	645	10918	39	2,5641

Tabla. 5 Resultados medidas del primer escenario (II)

La descripción de los valores de cada columna es la siguiente:

- **# Paquetes correctos**: Es el número de paquetes recibidos correctamente, ya sea directamente del receptor o bien una vez reconstruidos después de aplicar la cooperación.
- **# Paquetes recibidos**: Número de paquetes recibidos al receptor directamente del emisor.
- **% Paquetes recibidos OK**: Es el porcentaje del número de paquetes recibidos correctamente (correctos desde el emisor o bien reconstruidos con la cooperación).
- **% Paquetes recibidos OK sin cooperación**: Es el porcentaje de paquetes recibidos sin errores, sin el uso del 'Majority Voting'.
- **# Paquetes reconstruidos**: Es el número de paquetes reconstruidos en el escenario a través del 'Majority Voting'.
- **# Paquetes erróneos totales**: Es el número de paquetes erróneos que recibe el receptor, ya sea directamente del emisor o bien de los nodos intermedios en caso de reenvío de paquetes.
- **# Paquetes recibidos totales**: Es el número de paquetes recibidos en el sistema, ya sea directamente del receptor o de los nodos intermedios.
- **# Lanzamientos del majority Voting**: Es el número de veces que es lanzado correctamente el protocolo del 'Majority Voting'.
- **% Éxito del majority voting**: Es el porcentaje de veces que el protocolo del 'Majority Voting' es usado con éxito.

En este caso solamente tendremos en cuenta las medidas comprendidas entre dos y seis nodos, ya que con más nodos el uso del mecanismo de reconstrucción de paquetes no mejora el comportamiento del sistema inicial.

Por tanto, con los resultados obtenidos anteriormente, se puede elaborar una gráfica para comparar el porcentaje de los paquetes recibidos sin errores utilizando la técnica de la mayoría absoluta respecto al porcentaje de paquetes recibidos correctamente sin el uso de este mecanismo, tal como se puede apreciar en la Fig. 64.

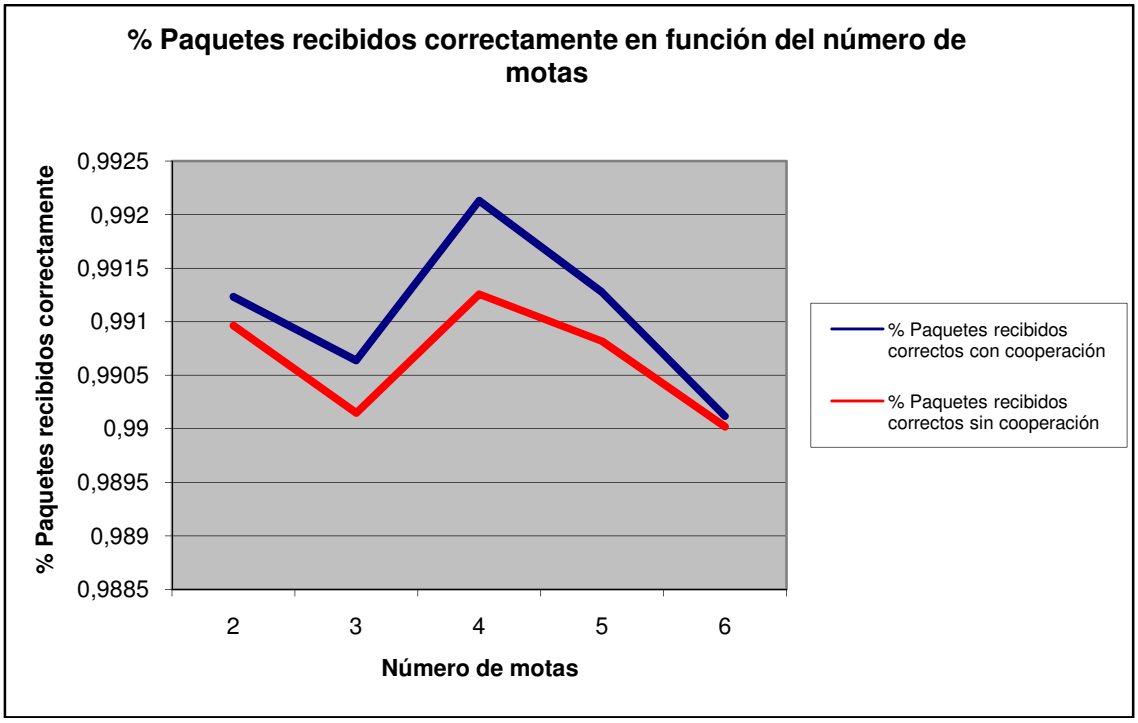


Fig. 61 % Paquetes recibidos correctamente en función del número de motas

Como se puede observar en la gráfica, el uso del mecanismo de mayoría absoluta siempre mejora el porcentaje de paquetes recibidos correctamente.

En este escenario, el número óptimo de nodos cooperantes es de 4, ya que con este valor obtenemos el punto donde el valor absoluto de los paquetes recibidos correctamente es mayor. Además, con este valor también obtenemos la mayor ganancia respecto al mismo escenario sin la utilización del mecanismo del *Majority Voting.*

También podemos observar como la ganancia de utilizar el *Majority Voting* respecto a no usarlo va aumentando hasta llegar al valor máximo. A continuación, vemos como va disminuyendo hasta convertirse en prácticamente nula.

Esto sucede así, ya que si tenemos un número de nodos cooperantes menor al óptimo, no tenemos suficiente diversidad de información, y por tanto es menos posible que se reconstruya el paquete correctamente. A medida que aumenta el número de nodos, hasta alcanzar el valor óptimo, esta ganancia irá aumentado ya que habrá más diversidad y por tanto más probabilidad de reconstruir el paquete correctamente. Sin embargo, cuando el número de cooperantes sobrepasa este número óptimo la ganancia irá disminuyendo, ya que cada vez tendremos más probabilidad de error en alguna de las transmisiones de los nodos cooperantes. Es decir, si disponemos de muchos nodos es posible que éstos tengan mucha información diferente y por tanto, en vez de ayudar, lo que hagan sea, empeorar el sistema.

En conclusión, interesa utilizar un valor óptimo del número de nodos cooperantes para reconstruir el mayor número de paquetes, el cual es distinto

en cada escenario ya que las condiciones de potencia, distancias, interferencias, etc. serán distintas en cada uno de ellos.

También es importante considerar el porcentaje de mejora que proporciona utilizar el sistema de *Majority Voting*, para ello disponemos de la Fig. 62.

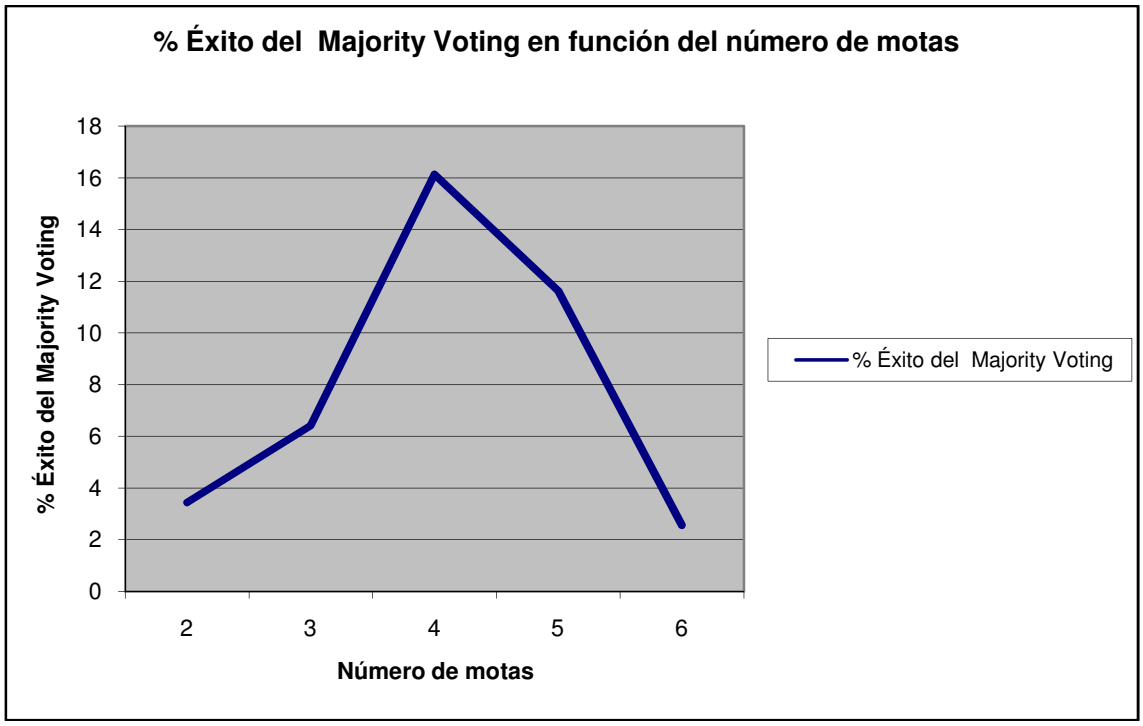


Fig. 62 % Éxito del mecanismo del *Majority Voting* en función del número de motas

Según nuestro estudio el uso del mecanismo de *Majority Voting* mejora en todos los casos respecto a no usarlo. Cuando en el sistema hay 4 nodos cooperantes, tenemos el mayor porcentaje de éxito. En cambio, si el número de nodos cooperantes es diferente a este valor, el porcentaje de éxito varía del mismo modo que en la Fig. 61.

El mayor porcentaje de éxito es 16,13%, pero este valor puede variar incrementándose o disminuyéndose en función del escenario.

3.2. Segundo escenario

El segundo escenario se ha analizado de la misma forma que el primero, es decir, se han efectuado distintas mediciones cambiando el número de nodos vecinos. Este escenario ha sido muy parecido al primero, ya que también se ha realizado dentro del laboratorio, por lo cual la ubicación de las motas se semeja bastante.

En este caso, hemos separado y distribuido las motas intermedias por toda la sala, siempre procurando que estuvieran más cercanas a la mota receptora que a la mota emisora. Ésta última ha sido colocada diagonalmente a la mota

receptora con tal de conseguir la máxima distancia entre las dos. La ubicación de las motas las podemos observar en la Fig. 63.

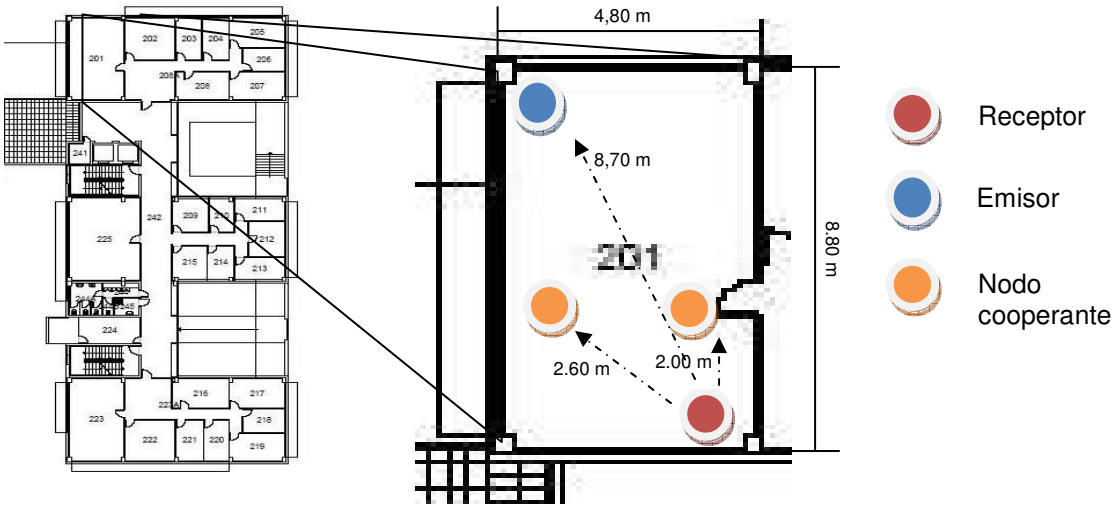


Fig. 63 Situación de las motas en el laboratorio 201 de la torre de profesores de la EPSC (II)

Esta vez se ha fijado el número de paquetes erróneos en 250, para poder obtener resultados estadísticamente más fiables.

Los resultados obtenidos en este escenario han sido:

# Nodos coop	# Paquetes Correctos	# Paquetes Recibidos	%Paquetes Recibidos OK	%Paquetes Recibidos OK Sin coop	# Paquetes Reconstruidos
2	29752	29996	0,9919	0,9917	6
3	29780	30008	0,9924	0,9917	22
4	29814	30041	0,9924	0,9917	23
5	29773	29998	0,9925	0,9917	25
6	29747	29993	0,9918	0,9917	4

Tabla. 6 Resultados medidas del segundo escenario (I)

# Nodos coop	# Paquetes Erroneos totales	# Paquetes Recibidos totales	# Lanzamientos del Majority Voting	% Éxito del Majority Voting
2	539	30610	224	2,6786
3	576	30993	201	10,9453
4	664	31086	176	13,0682
5	898	31352	134	18,6567
6	1026	31702	111	3,6036

Tabla. 7 Resultados medidas del segundo escenario (II)

Como en el caso anterior, solo tendremos en cuenta las medidas comprendidas entre dos y seis nodos, ya que con más nodos el uso del mecanismo de reconstrucción de paquetes no mejora el comportamiento del sistema inicial.

Con los resultados obtenidos anteriormente se puede elaborar una gráfica para comparar el porcentaje de los paquetes recibidos sin errores utilizando la técnica de la mayoría absoluta respecto al porcentaje de paquetes recibidos correctamente sin el uso de este mecanismo, tal como se puede apreciar en la Fig. 64.

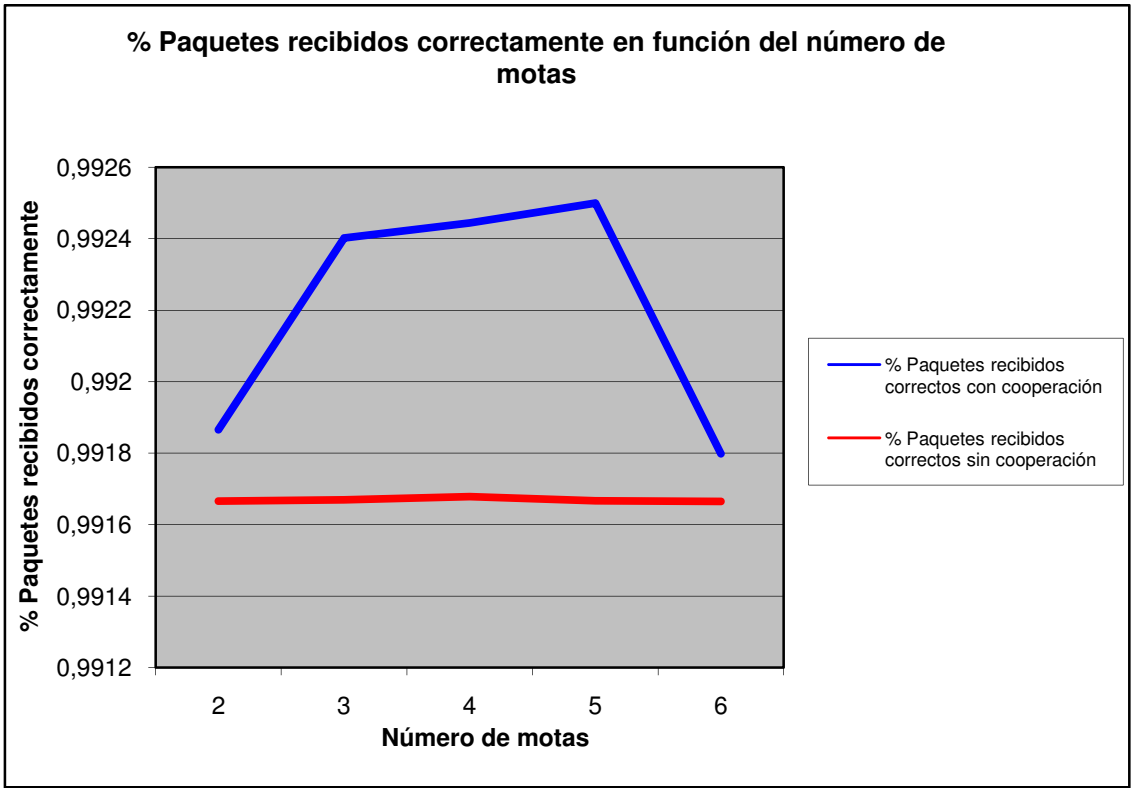


Fig. 64 % Paquetes recibidos correctamente en función del número de motas

Como se puede comprobar en este escenario, la utilización del '*Majority Voting*' mejora el porcentaje de paquetes recibidos correctamente en el nodo receptor.

En este caso, conseguimos el mayor porcentaje de paquetes correctos con 5 nodos cooperantes, por tanto este será el valor óptimo. Además, con este valor también conseguimos la mayor ganancia respecto al mismo escenario sin la utilización del mecanismo de reconstrucción de paquetes.

Del mismo modo que en el primer escenario la gráfica va aumentando hasta llegar a su punto máximo (con 5 nodos) y partir de aquí va disminuyendo por la misma causa que en el escenario anterior (Véase apartado 3.1).

Al comparar estos resultados con los obtenidos del primer escenario corroboramos que el comportamiento general del sistema es el mismo, cambiando únicamente los valores absolutos de forma pequeña debido al cambio de ubicación de las motas.

A continuación, nos centraremos en el porcentaje de mejora que proporciona la técnica de reconstrucción de paquetes en este nuevo escenario, la cual se puede ver representada en la Fig. 65.

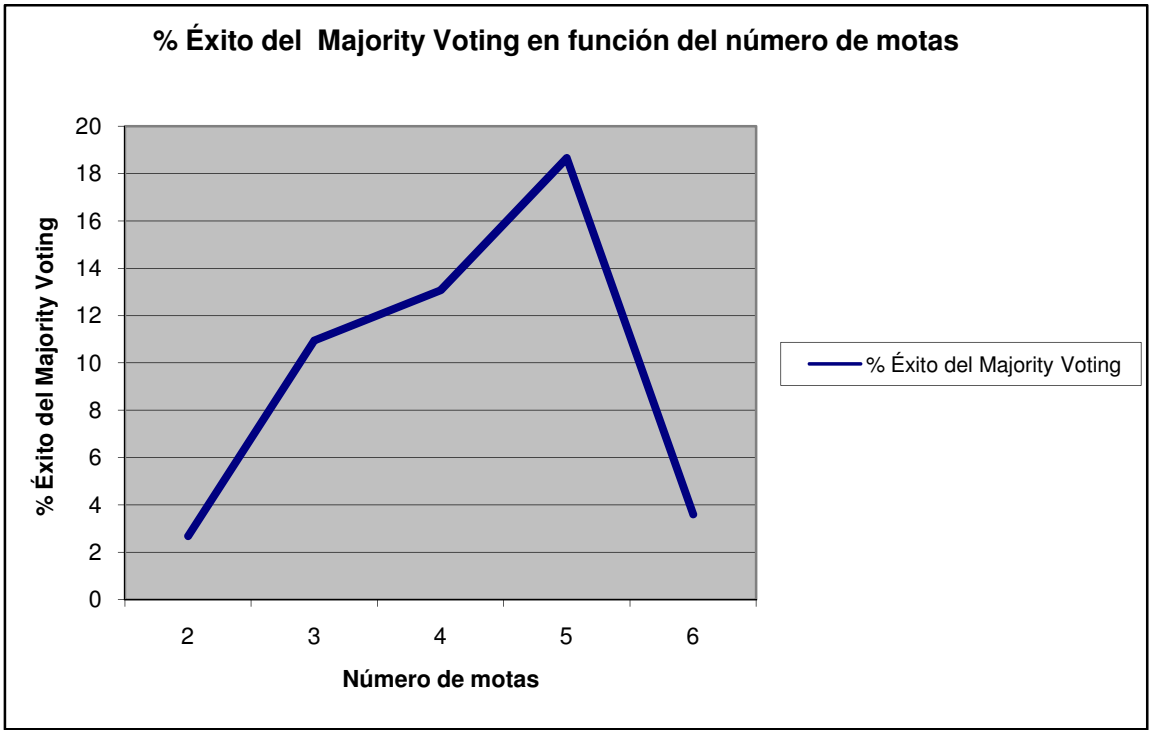


Fig. 65 % Éxito del mecanismo del *Majority Voting* en función del número de motas

En este segundo escenario observamos, que de la misma forma que en el primer escenario, siempre se mejora el sistema al utilizar el '*Majority Voting*'.

El mayor porcentaje de éxito se obtiene con 5 nodos cooperantes, y es del valor de 18,65%. Respecto al primer escenario se ha ganado aproximadamente un 2% utilizando un nodo cooperante más. Estos cambios, son debidos al cambio de ubicación de las motas cooperantes dentro del laboratorio de investigación.

3.3. Tercer Escenario

El tercer escenario se ha realizado en un lugar exterior, en este caso en los exteriores del edificio de la EPSC. Se ha realizado en este tipo de escenario ya que habrá diferentes valores de interferencias.

En este experimento hemos fijado el número de paquetes erróneos a 100, del mismo modo que en el primer escenario, para poder comparar los resultados de dos escenarios distintos. Se ha escogido de nuevo este valor en vez de 250 paquetes debido a que el tiempo invertido para realizar el experimento era menor ya que no podíamos dejar las motas en el exterior realizando medidas toda una noche.

Respecto a los escenarios anteriores, en este caso hemos aumentado a 26 el nivel de transmisión de la información, ya que la distancia entre las motas es mayor.

La situación de las motas en este caso, las podemos ver reflejadas en la siguiente Fig. 66:

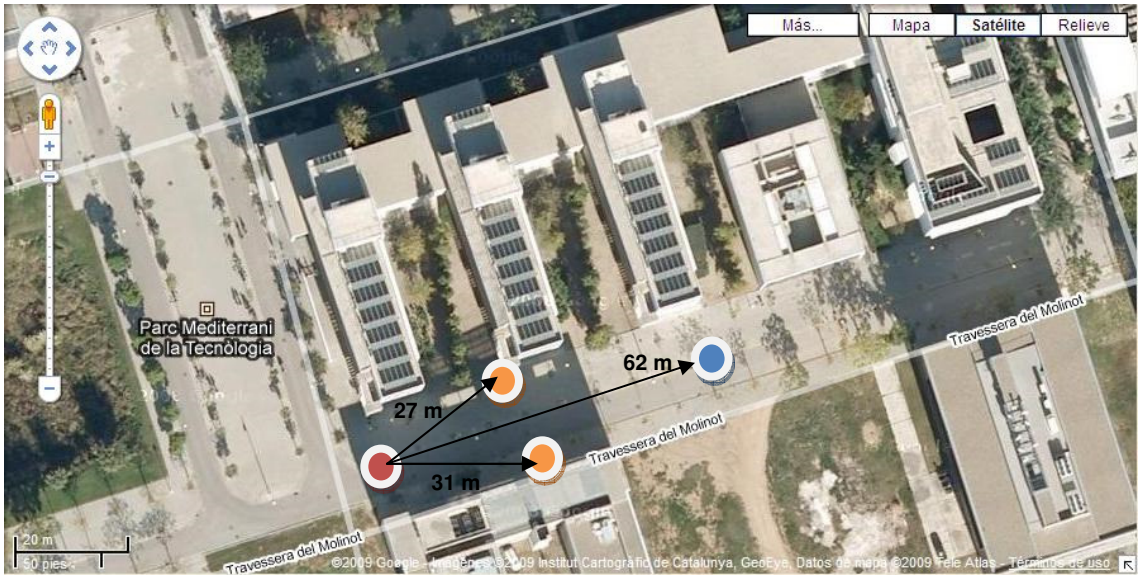


Fig. 66 Situación de las motas en el exterior de la EPSC

Los resultados obtenidos en este tercer escenario se ven reflejados en la Tabla. 8 y en la Tabla. 9:

# Nodos coop	# Paquetes Correctos	# Paquetes Recibidos	%Paquetes Recibidos OK	%Paquetes Recibidos OK Sin coop	# Paquetes Reconstruidos
2	9908	10005	0,9903	0,9900	3
3	9932	10022	0,9910	0,9900	10
4	9920	10012	0,9908	0,9900	8
5	9915	10009	0,9906	0,9900	6
6	9877	9975	0,9902	0,9900	2

Tabla. 8 Resultados medidas del tercer escenario (I)

# Nodos coop	# Paquetes Erroneos totales	# Paquetes Recibidos totales	# Lanzamientos del Majority Voting	% Éxito del Majority Voting
2	249	10228	89	3,3708
3	295	10384	73	13,6986
4	264	10443	60	13,3333
5	307	10549	48	12,5000
6	465	10720	36	5,5556

Tabla. 9 Resultados medidas del tercer escenario (II)

Para poder comparar los resultados con los del primer escenario se han vuelto a tener en cuenta los valores comprendidos entre 2 y 6 nodos, ya que si se utilizan más nodos el porcentaje de éxito se vuelve prácticamente inexistente.

La gráfica obtenida del porcentaje de paquetes recibidos correctamente se puede ver en la Fig. 67:

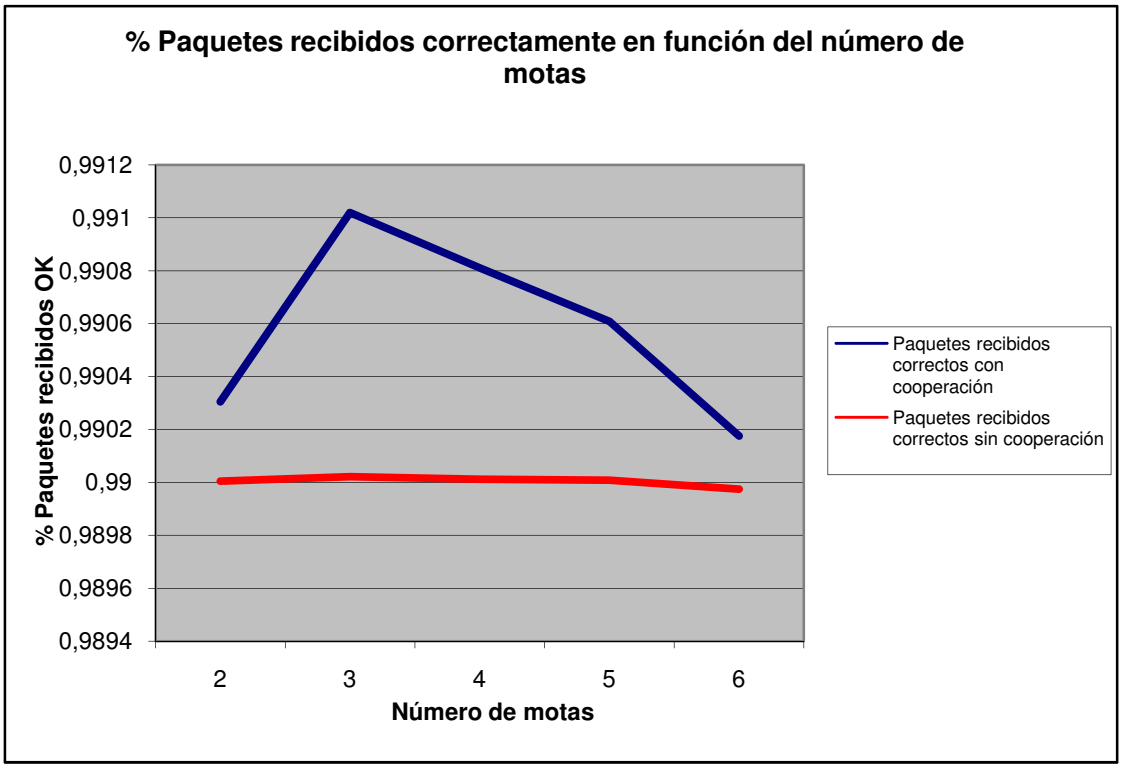


Fig. 67 % Paquetes recibidos correctamente en función del número e motas

En el tercer escenario observamos que el porcentaje de paquetes recibidos correctamente utilizando la técnica de reconstrucción de paquetes esta siempre por encima respecto a no usarla. En este escenario, el valor óptimo de nodos cooperantes son 3, ya que con este valor obtenemos el mayor porcentaje de paquetes recibidos sin errores y además es el punto donde hay una mayor mejoría respecto a no usar dicha técnica.

A la hora de compararlo con el primer escenario vemos que el número óptimo de nodos cooperantes ha variado, respecto a que las motas están ubicadas de una forma diferente, y además al no estar en el mismo espacio las interferencias producidas sean diferentes.

De todos modos, el comportamiento general del sistema es el mismo, ya que los paquetes recibidos correctamente aumentan hasta llegar a su punto máximo, donde vuelve a decaer, debido a la diversidad de la información explicada en el apartado 3.1.

En este escenario, hemos emitido a una mayor potencia que en el laboratorio, ya que disponíamos de un espacio mayor y por tanto podíamos alejar más las motas unas de otras. Debido a esto, el número óptimo de nodos cooperantes ha disminuido a 3.

Para finalizar, en la Fig. 68 se muestra el porcentaje de éxito al utilizar la técnica del '*Majority Voting*'.

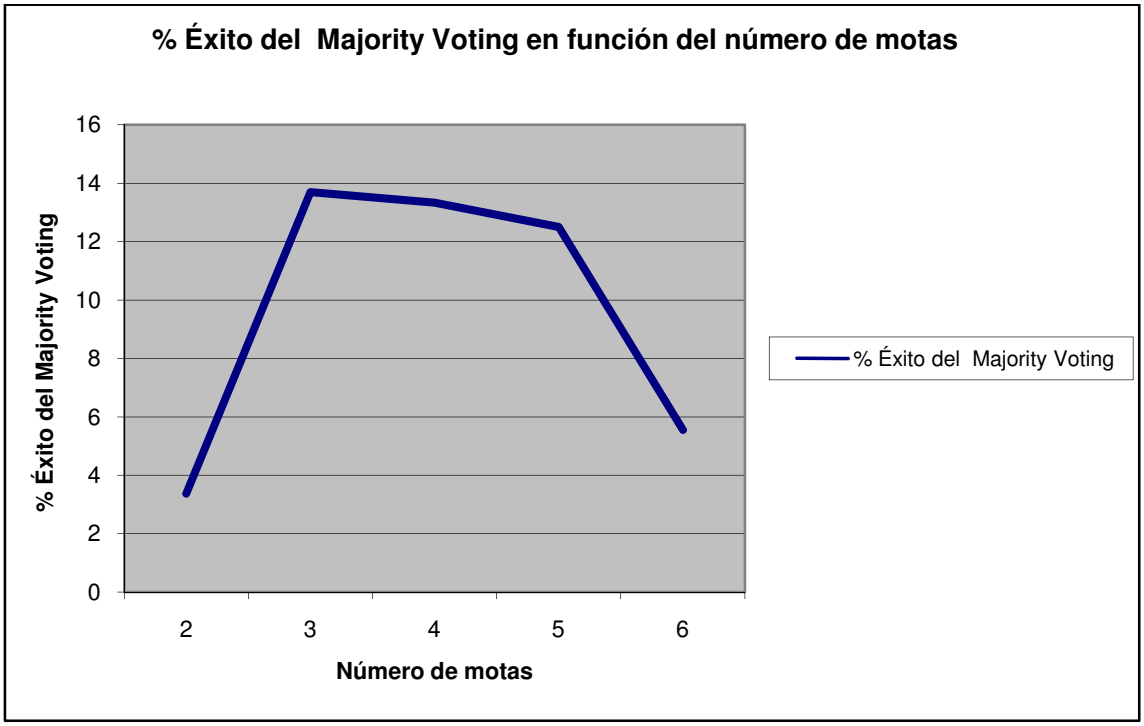


Fig. 68 % Éxito del *Majority Voting* en función del número de motas

En este escenario vemos como el comportamiento general del sistema varia respecto a los dos escenarios anteriores, ya que en este caso el porcentaje se mantiene aproximadamente constante entre 3 y 5 nodos. Fuera de estos valores el porcentaje decae considerablemente, debido en un caso a la falta de diversidad (pocos nodos cooperantes), y en el otro a las interferencias y la ubicación del resto de nodos cooperantes (muchos nodos cooperantes). En cambio, en el primer escenario se puede apreciar claramente un pico del valor máximo.

CONCLUSIONES

Este proyecto se ha basado en implementar una técnica de transmisión cooperativa en una red de sensores inalámbricos, basados en la tecnología telosb de Crossbow.

El objetivo inicial del proyecto era conseguir un sistema implementando la técnica cooperativa del '*Majority Voting'*, el cual mejora el rendimiento respecto al sistema inicial en escenarios donde haya una alta probabilidad de error del contenido de los paquetes.

Una vez finalizado el proyecto, este objetivo ha sido alcanzado, ya que en el mejor caso conseguimos aumentar la eficiencia en un valor aproximado del 20%. Gracias a esta mejora, se reduce el consumo de energía de la mota trasmisora ya que evita la retransmisión de paquetes por parte de ésta de forma innecesaria. De esta forma, queda demostrado que la utilización de nodos cooperantes beneficia el comportamiento del sistema en términos de eficiencia global.

Es importante tener en cuenta el número de nodos cooperantes, ya que en cada escenario el número óptimo es distinto. Con un número de nodos cooperantes menor al óptimo, no hay la suficiente información como para reconstruir el paquete de una forma fiable. En cambio, con un número mayor al óptimo, se obtiene mucha diversidad de la información y ésta puede afectar a la hora de escoger el valor correcto.

La técnica implementada se puede aplicar en diversos campos y escenarios de la vida cotidiana, donde se alargará la vida útil de las baterías y se evitará la retransmisión de paquetes innecesarios a gran potencia especialmente en zonas colindantes del área de cobertura.

El proyecto nos ha servido para aprender y profundizar:

- El manejo del entorno Linux, ya que es el sistema operativo sobre el cual trabaja TinyOS.
- Aprender el lenguaje de programación NesC utilizado por las motas telosb de Crossbow.
- Se ha profundizado el lenguaje de programación orientado a objetos (Java), ya que éste ha sido utilizado para crear la interfaz gráfica del usuario y transmitir la información necesaria del PC a la mota con función de estación base y viceversa.
- Técnicas de transmisión cooperativas en redes de sensores inalámbricos, especialmente la técnica del '*Majority Voting'*.
- Aprendizaje de forma autónoma de toda la información requerida para la utilización de este proyecto, así como resolver problemas inesperados a la hora de implementar el código y realizar las mediciones.
- Demostrar que no es trivial trabajar en el límite de la zona de cobertura, ya que encontrar dicha zona es complicado. Teóricamente es sencillo

hacer cálculos en esta área, en cambio, en la práctica el medio es variable y es difícil encontrar un escenario ideal teóricamente.

- El chip CC2420 utilizado por las motas telosb rechazaba los paquetes con CRC erróneo. Se ha aprendido a trabajar con este chip, para no descartar dichos paquetes y poder tratarlos específicamente.

A partir de este punto, se abre una línea de futuras implementaciones para poder realizar diferentes proyectos, gracias a la gran diversidad de técnicas de cooperación pudiendo mejorar la probabilidad de error de paquete y el consumo de las baterías. Además se podrá realizar comparaciones entre diferentes técnicas para poder verificar cual es la más adecuada según el tipo de escenario y poder seleccionarla automáticamente.

Otra nueva línea de estudio seria mejorar la seguridad del sistema, ya que la información viaja a través del medio radio, el cual es vulnerable ante posibles ataques.

BIBLIOGRAFÍA

[1] Página oficial de Tinyos, www.tinyos.net.
[2] Tutorial de Tinyos, http://www.tinyos.net/tinyos-1.x/doc/tutorial/.
[3] Datasheet CC2420, http://focus.ti.com/lit/ds/symlink/cc2420.pdf.
[4] Datasheet telosB, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.
[5] López José, Mena Sergio, "Desarrollo de un demostrador para evaluar técnicas Cross-Layer en sistemas de comunicaciones inalámbricos", TFC, Universitat Politécnica de Catalunya, Marzo 2008.
[6] Devroyé Santi, "Evaluación de protocolos basados en transmisión cooperativa para sistemas de redes de sensores", TFC, Universitat Politécnica de Catalunya, Octubre 2008.
[7] Tutorial NesC, http://ants.dif.um.es/rm/apuntes/Tutorial-Slides.pdf.
[8] Introducción al lenguaje NesC, http://nescc.sourceforge.net/papers/nesc-pldi-2003.pdf.
[9] Vakil Sam, Liang Ben, "Balancing Cooperation and Interference in Wireless Sensor Network", http://www.comm.utoronto.ca/~liang/publications/SECON06.pdf.
[10] Morillo Julián, García Jorge, Perez Ana, "Collaborative ARQ in Gíreles Energy-Constrained Networks, http://portal.acm.org/citation.cfm?id=1080810.1080813.