

UNIVERSIDAD POLITÉCNICA DE CARTAGENA



PROYECTO FIN DE CARRERA

Desarrollo de un Modelo de Funcionamiento Colaborativo usando Dispositivos TelosB



AUTOR: Juan Salvador Galindo Pedreño
DIRECTOR: Felipe García Sánchez



Autor	Juan Salvador Galindo Pedreño
E-mail	jsgp85@wanadoo.es
Director	Felipe García Sánchez
E-mail	felipe.garcia@upct.es
Título del PFC	Desarrollo de un Modelo de Funcionamiento Colaborativo usando Dispositivos TelosB.
<u>Resumen</u> El propósito de este Proyecto Final de Carrera es la creación de una aplicación Colaborativa para redes de sensores inalámbricas. Para poder efectuarlo se ha realizado un estudio del protocolo IEEE 802.15.4, encargado de gestionar el funcionamiento de este tipo de redes, y de las herramientas necesarias para llevarlo a cabo, tanto hardware como software. La aplicación Colaborativa se va a encargar de encaminar los mensajes según su destino final y el RSSI de los enlaces entre los dispositivos.	
Titulación	Ingeniero Técnico de Telecomunicación, especialidad Telemática
Departamento	Tecnologías de la Información y Comunicaciones
Fecha de Presentación	30/09/08

ÍNDICE

1) Introducción	7
2) Estado del arte	9
2.1) IEEE 802.15.4.....	9
2.1.1) <i>Introducción</i>	9
2.1.2) <i>Capa de aplicación</i>	14
2.1.3) <i>Capa de red</i>	22
2.1.4) <i>Capa MAC</i>	30
2.1.5) <i>Especificación de los servicios de seguridad</i>	34
2.2) NesC	37
2.3) Tinyos	45
2.4) XubunTOS	55
2.5) Dispositivos TelosB	56
3) Desarrollo de la aplicación	59
4) Evaluación del funcionamiento de la aplicación	75
5) Conclusiones y líneas futuras	79
Bibliografía	81

Capítulo 1

Introducción

Los últimos avances tecnológicos han hecho realidad el desarrollo de unos dispositivos de pequeño tamaño, que son capaces tanto de procesar información local como de comunicarse de forma inalámbrica, midiendo variables como luz, humedad y temperatura. Estos dispositivos se interconectan por medio de las denominadas redes de sensores. Cada nodo que forma la red consta de un microcontrolador, unos determinados sensores y transmisor/receptor, formando una red con otros muchos nodos, también llamados motes o sensores.

Este proyecto se centra en los sensores o motes basados en la tecnología Zigbee. Estos dispositivos son capaces de crear redes de sensores de bajo coste y de bajo consumo energético.

En el debe de esta tecnología se encuentra que un sensor de este tipo es capaz de procesar una limitada cantidad de datos. Además, la interferencia de otros sistemas inalámbricos junto a los bajos alcances de estos sensores no permiten a priori su uso generalizado. Pero la realidad es bien distinta. La utilización de sensores Zigbee en entornos tan dispares como la industria, la agricultura, la automoción o la domótica permite que, cada vez más, esta nueva tecnología penetre en entornos que se encontraban monopolizados por otros tipos de tecnologías (bluetooth, wifi, etc).

Este Proyecto Fin de Carrera tiene como el objetivo en primer lugar la formación en este tipo de redes, estudiando las alternativas que se generan. Para ello se ha utilizado motes del tipo TelosB de la empresa Crossbow que permiten crear redes de sensores para el intercambio de información entre los distintos motes. Se puede decir que este proyecto ha de servir como un tutorial de funcionamiento y administración de este tipo de redes particularizado para estos dispositivos.

En una red algo más extendida, el envío de la información desde el dispositivo emisor a los motes destinatarios puede pasar por varios nodos (que también, para minimizar costes, serán motes TelosB, que tendrán otras tareas asignadas) intermedios. Estos nodos intermedios deberán tomar decisiones para reenviar la información al nodo que consideren más conveniente o simplemente al destino final.

Por ello, una de las problemáticas más interesantes a la hora de componer este tipo de redes es la posibilidad de optimizar la transmisión, enviando la información a

través de los que se van encontrando en mejor posición hasta llegar al que va destinado el mensaje, si es conocido. Es necesario conocer por parte del nodo qué nodo es el más apropiado para enviar la información. Para resolver esta problemática se plantea el segundo gran objetivo de este Proyecto Fin de Carrera, el desarrollar una aplicación con tal fin: que permita la decisión activa y en su funcionamiento normal (tiempo real) del mejor enlace para la transmisión ó *routing cualitativo*. Para ello se utilizará como parámetro de decisión, el valor del RSSI (*Received Signal Strength Indicator*) un parámetro que indica la potencia recibida por un nodo, muy utilizado para este tipo de propósitos.

Finalmente, se presentan los resultados de una red de nodos TelosB que avalan las implementaciones realizadas.

Capítulo 2

Estado del arte

2.1 IEEE 802.15.4 (Zigbee)

2.1.1 Introducción

ZigBee es una nueva tecnología inalámbrica de corto alcance y bajo consumo originaria de la antigua alianza HomeRF y que se definió como una solución inalámbrica de baja capacidad para aplicaciones en el hogar como la seguridad y la automatización.

Entre las aplicaciones que puede tener están:

- Domótica.
- Automatización industrial.
- Reconocimiento remoto.
- Juguetes interactivos.
- Medicina.

Objetivo

El objetivo de esta tecnología no es obtener velocidades muy altas, ya que solo puede alcanzar una tasa de 20 a 250Kbps en un rango de 10 a 75 metros, si no que es obtener sensores cuyos transceptores tengan un muy bajo consumo energético. De hecho, algunos dispositivos alimentados con dos pilas AA puedan aguantar 2 años sin el cambio de baterías. Por tanto, dichos dispositivos pasan la mayor parte del tiempo en un estado latente, es decir, durmiendo para consumir mucho menos.

Bandas de operación

ZigBee opera en las bandas libres de 2.4Ghz, 858Mhz para Europa y 915Mhz para Estados Unidos. En la siguiente figura se puede ver el espectro de ocupación en las bandas del protocolo 802 (incluyendo ZigBee).

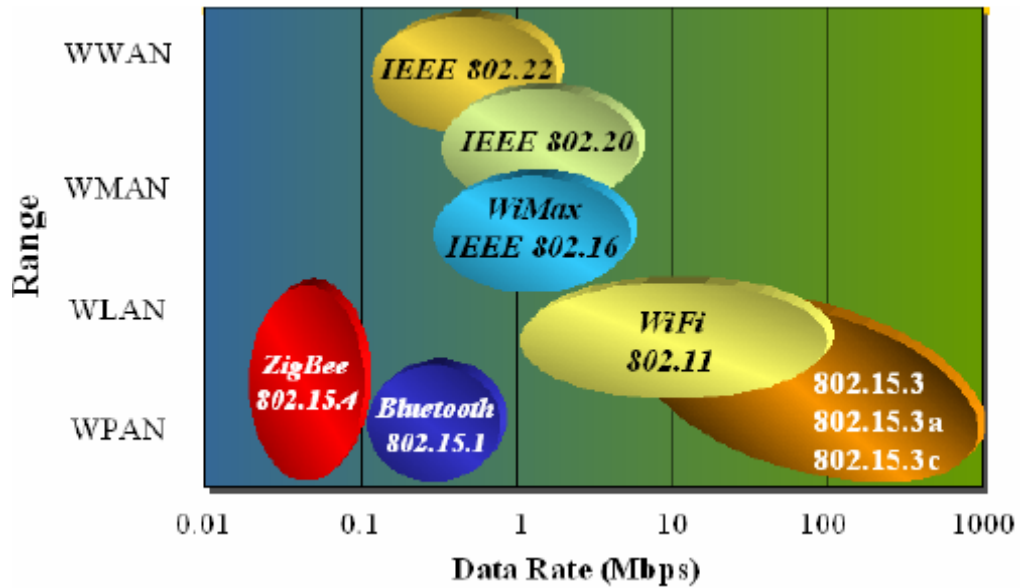


Figura1:Tecnologías en 2.4 GHz

A una velocidad de transmisión de 250Kbps y a una potencia de 1mW cubre aproximadamente unos 13 metros de radio. En la siguiente figura se muestran las características de radio de las señales.

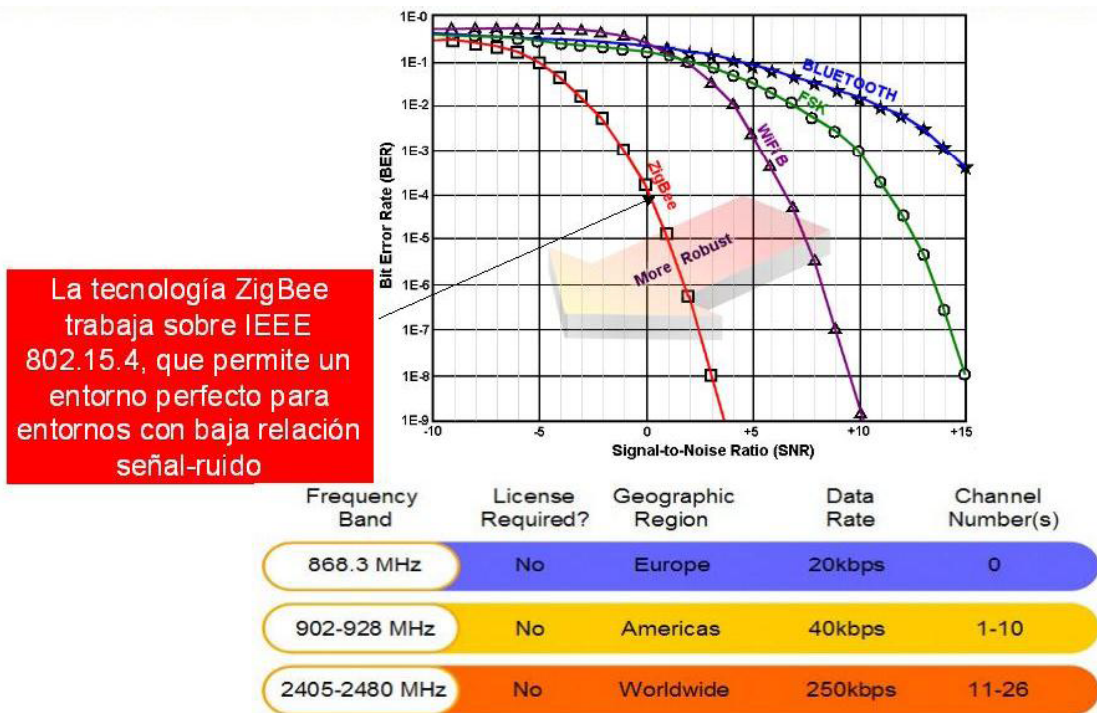


Figura 2:Características de radio

En la siguiente tabla se puede observar la distancia en función de la potencia transmitida y la velocidad de transmisión:

Potencia(mW) / Velocidad(Kbps)	1mW	10mW	100mW
28 Kbps	23m	54m	154m
250 Kbps	13m	29m	66m

Tabla 1: Distancia de transmisión

En cuanto a la gestión del control de acceso al medio hace uso de CSMA/CA (Carrier Sense Multiple Acces with Collision Avoidance) y es posible usar ranuras temporales TDMA (Time Division Multiple Access) para aplicaciones de baja latencia.

Nodos y topología de red

En una red ZigBee pueden haber hasta 254 nodos, no obstante, según la agrupación que se haga, se pueden crear hasta 255 conjuntos de nodos con lo cual se puede llegar ha tener 64770 nodos para lo que existe la posibilidad de utilizar varias topologías de red: en estrella, en malla o en grupos de árboles, como puede verse a continuación:

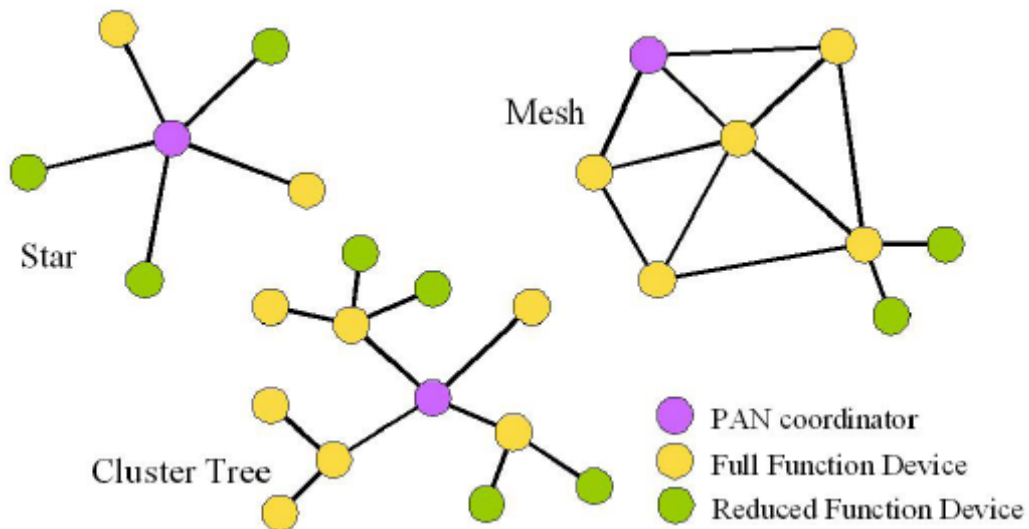


Figura 3: Topologías de Red

Se permite un encaminamiento o enrutamiento de saltos múltiples, también conocido como multi-hop, que permite que estas redes abarquen una gran superficie.

En ZigBee hay tres tipos de dispositivos:

- Coordinador
 - Sólo puede existir uno por red.
 - Inicia la formación de la red.
 - Es el coordinador de PAN.

- Router
 - Se asocia con el coordinador de la red o con otro router ZigBee.
 - Puede actuar como coordinador.
 - Es el encargado del enrutamiento de saltos múltiples de los mensajes.
- Dispositivo final
 - Elemento básico de la red.
 - No realiza tareas de enrutamiento.

Una posible configuración de una red sería la siguiente:

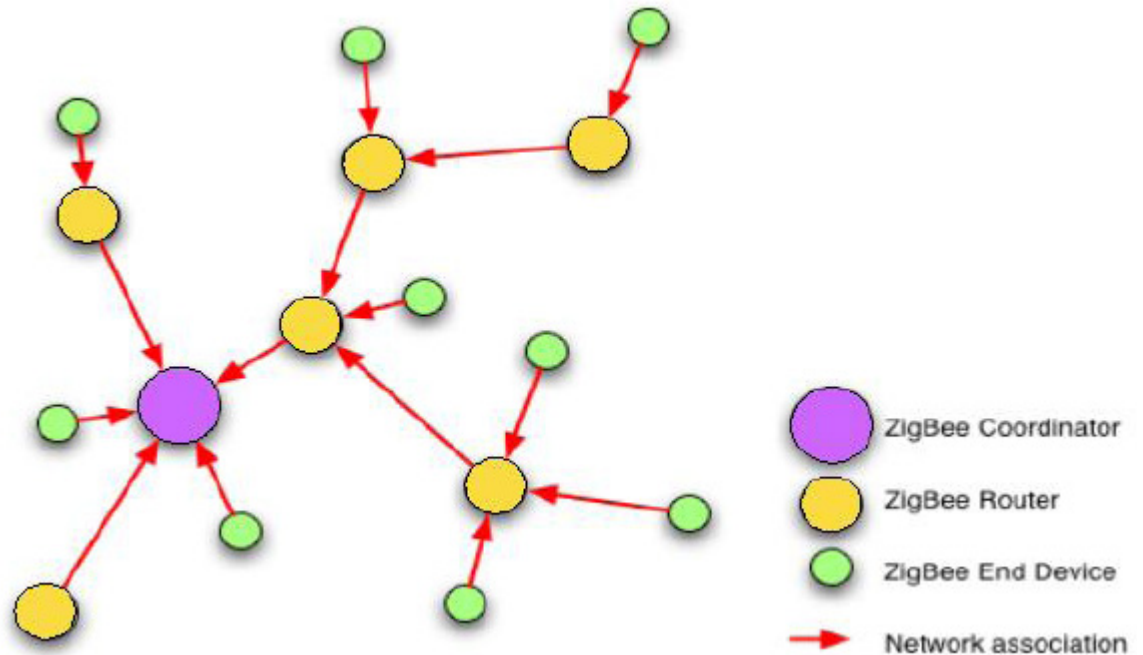


Figura 4: Ejemplo de red ZigBee

Otro punto importante es el soporte y la disponibilidad total de la malla, es decir, que ante caídas de nodos, la red busca caminos alternativos para el intercambio de mensajes, un ejemplo se puede ver a continuación. Supongamos que disponemos de una red en la cual los nodos están conectados en malla y se intercambian datos entre un interruptor y una lámpara.

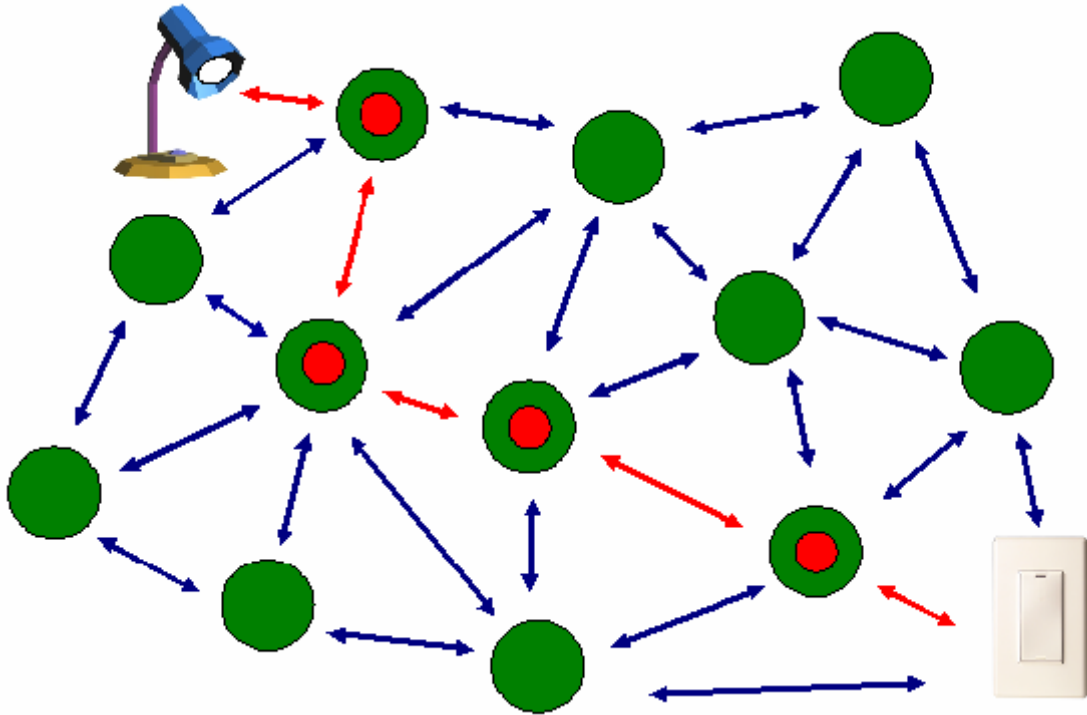


Figura 5: Camino de comunicación (interconexión)

Si algunos de los nodos que contiene falla y dichos nodos formaban parte del camino que seguían los mensajes en la comunicación, la red podría sufrir una caída:

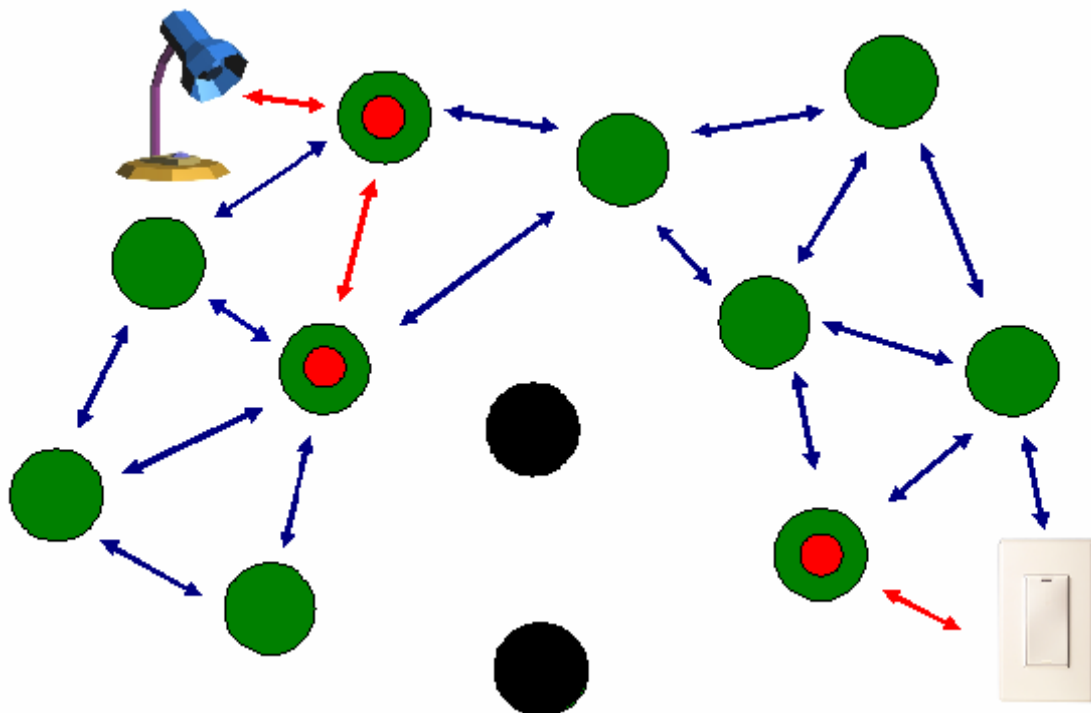


Figura 6: Caída de dos nodos de red

ZigBee permite que se puedan establecer rutas alternativas para seguir comunicando los dispositivos:

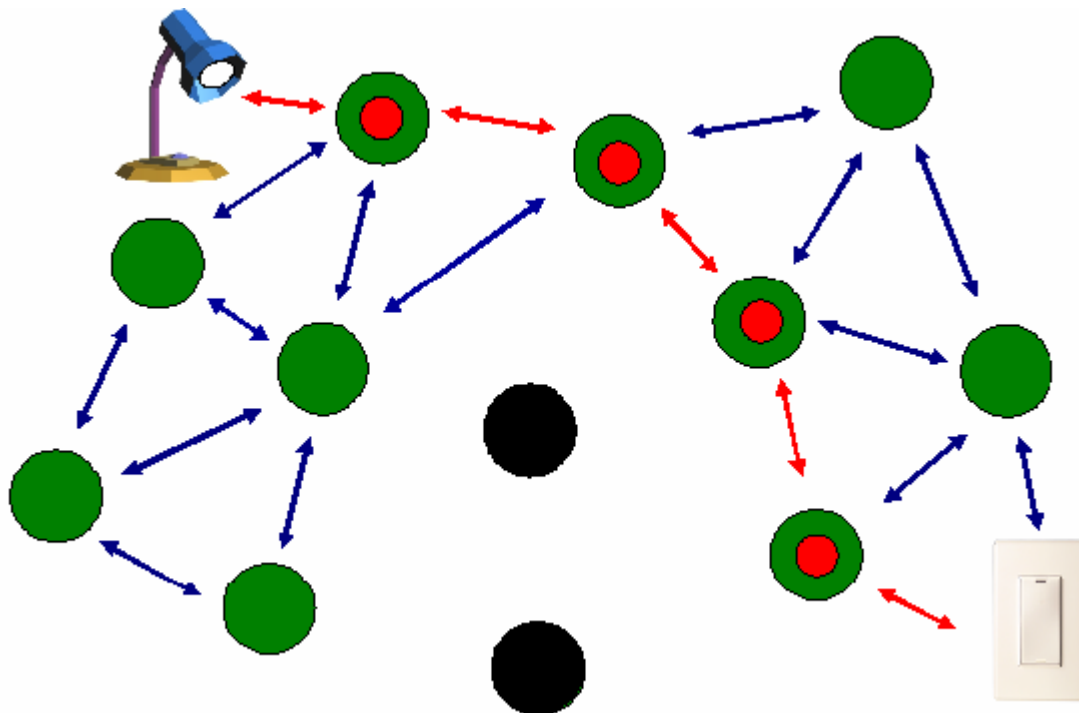


Figura 7: Creación de un camino alternativo

Seguridad

En cuanto a seguridad, ZigBee puede utilizar la encriptación AES de 128bits, que permite la autenticación y encriptación en las comunicaciones. Además, existe un elemento en la red llamado Trust Center (Centro de validación) que proporciona un mecanismo de seguridad en el que se utilizan dos tipos de claves de seguridad, la clave de enlace y la clave de red.

2.1.2 Capa de Aplicación

La pila de arquitectura ZigBee consta de varios componentes en capas como IEEE 802.15.4 2003 en la capa de Control de Acceso al Medio (MAC), la capa física (PHY) y la capa de red Zigbee (NWK).

La capa de aplicación de ZigBee se subdivide en la subcapa APS, la capa ZDO (Zigbee Device Objects) y los objetos de aplicación definidos por cada uno de los fabricantes.

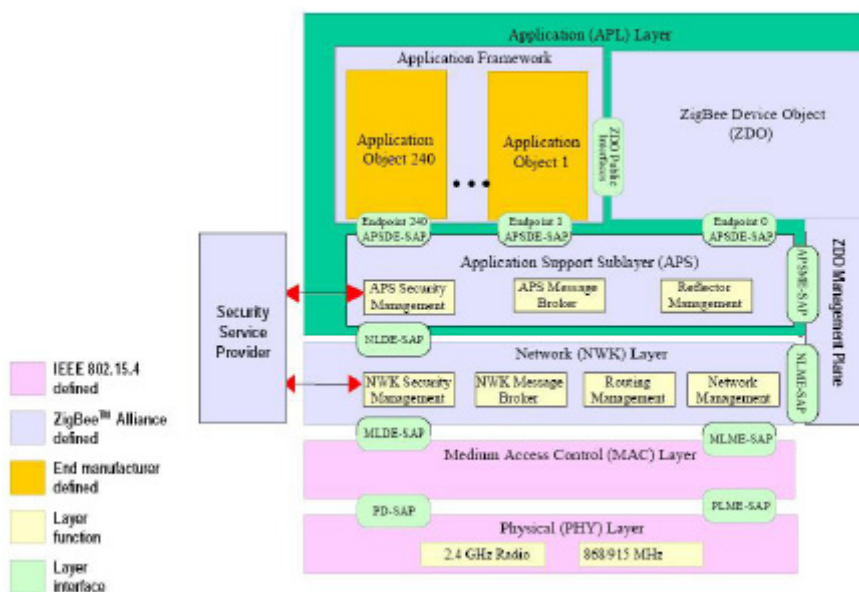


Figura 8: Pila de protocolo ZigBee

2.1.2.1 Subcapa de Soporte

La subcapa de soporte de aplicación (APS) proporciona un interfaz entre la capa de red (NWK) y la capa de aplicación (APL) a través de un conjunto de servicios que se utilizan junto a los ZDO y otros objetos que hayan sido definidos por los fabricantes. Los servicios los ofrecen dos entidades: la entidad de datos APS (APSD) a través del servicio de punto de acceso APSDE (APSDE-SAP) y la entidad gestora del APS (APSME-SAP) a través de un servicio que ofrece el punto de acceso APSE-SAP. APSDE proporciona el servicio necesario para la transmisión de datos y el transporte de de datos de aplicación entre dos o más dispositivos en la misma red. APSME proporciona el servicio de descubrimiento y enlace de dispositivos y mantiene una base de datos de los objetos llamado “APS Information Base (AIB)”.

2.1.2.2 Estructura de Aplicación

Dentro de la estructura de aplicación, los objetos de envían y reciben datos a través del APSDE-SAP. El control y la gestión de los objetos de aplicación son llevados a cabo por los interfaces de los ZDO. El servicio de datos ofrecido por el APSDE-SAP, incluye primitivas de petición, confirmación, respuesta e indicación (request, confirm, response, indication) para la transferencia de datos. La primitiva *request* soporta la transferencia de datos entre pares de entidades objeto de aplicación. La primitiva *confirm* da los resultados de una llamada de la primitiva *request*. La primitiva *indication* se usa para indicar la transferencia de datos desde un APS a la entidad objeto de aplicación. Se pueden definir más de 240 objetos de aplicación llamados terminales, con interfaces que para cada uno de los terminales se enumeran del 1 al 240. Hay dos terminales adicionales que utiliza el APSDE-SAP; el 0 está reservado para el interfaz de datos de los ZDO y el 255 se reserva para que el interfaz de datos realice las peticiones de broadcast de datos para todos los objetos de aplicación. Los terminales que van del 241 al 254 se reservan para usos futuros.

Servicio de Parejas Clave-Valor

El servicio de pares key-valor (KVP) permite a los atributos definidos, que en los objetos de aplicación se puedan utilizar primitivas como *get*, *get response*, *set* y *set response*. Además, KVP utiliza estructuras de datos de marcado XML en una versión más reducida. Esta solución proporciona un mecanismo de instrucciones y control para la gestión de pequeños dispositivos que permiten ser el germen de la difusión de los datos XML.

Servicio de Mensajes

Existen varias áreas de aplicación en ZigBee que tiene protocolos de direccionamiento propietarios y que no funcionan bien con KVP. Por tanto, existen cabeceras que KVP asume y que sirven para controlar el estado de las variables, que permitan seleccionar, obtener o realizar las acciones necesarias que se puedan producir ante ciertos eventos que requieran los dispositivos para mantener las variables de estado de comunicación.

2.1.2.3 Direccionamiento de Terminales

ZigBee proporciona un subnivel de direccionamiento, que se usa de manera conjunta con otros mecanismos como es el protocolo IEEE802.15.4. Por ejemplo; hay un número de terminales (endpoints) que se pueden utilizar para identificar interruptores y bombillas. El terminal 0 está reservado para la gestión de dispositivos y es utilizado para direccionar los descriptores del nodo. Cada subunidad que se identifica en un nodo (como pueden ser los interruptores y las bombillas) se asigna a un terminal específico dentro del rango 1-240. Para permitir una diferenciación de productos en el mercado, los fabricantes pueden añadir clusters que contengan atributos extra para sus propios perfiles. Estos clusters específicos no forman parte de la especificación de ZigBee y su interoperabilidad no está garantizada. Dichos servicios deben ser indicados en cada uno de los terminales descritos por parte del fabricante, acompañando a poder ser la nueva hoja de especificaciones.

2.1.2.4 Fundamentos de comunicación de la capa de Aplicación

Perfiles

Los perfiles son acuerdos a los que se llega por mensajes. El formato de estos mensajes y las acciones producidas, permiten a las aplicaciones residir en cada uno de los dispositivos individuales para enviar instrucciones, realizar peticiones de datos o procesar instrucciones/datos para crear así una aplicación distributable e interoperable. Los perfiles son desarrollados por cada uno de los fabricantes ZigBee, que en base a las necesidades que existen en el mercado, proporcionan soluciones tecnológicas específicas. Los perfiles por tanto tratan de unificar la tecnología con las necesidades del mercado.

Clusters

Los clusters son identificados por un identificador de cluster (Cluster ID), éste cluster se asocia al dispositivo que produce los flujos de datos. Los identificadores de

clusters son únicos dentro de un mismo perfil. Los enlaces se producen por la relación existente entre identificadores de clusters de salida y de entrada, asumiendo que ambos clusters están dentro de un mismo perfil.

2.1.2.5 Descubrimiento

Dispositivo de Descubrimiento

El servicio de descubrimiento (Device Discovery), es el proceso por el cual un dispositivo ZigBee descubre otros dispositivos. Para ello, realiza preguntas/solicitudes que se envían por broadcast o unicast. Hay dos formas de realizar las peticiones de descubrimiento de servicios y dispositivos: la petición de dirección IEEE y la petición de dirección de NWK. La petición de IEEE es unicast y asume que la dirección NWK es conocida. La petición de dirección NWK es por broadcast y lleva la dirección de IEEE como datos de negociación de parámetros.

Las respuestas al elemento que ha realizado las peticiones broadcast o unicast de mensajes de descubrimiento pueden variar según provengan de un tipo de dispositivos lógicos u otros, como se indica a continuación:

- **Terminal:** responde a las peticiones de descubrimiento de dispositivos enviando su propia dirección IEEE o la dirección NWK (dependiendo de la petición).
- **Coordinador:** responde a la petición enviando su dirección IEEE o NWK y las direcciones IEEE o NWK que tiene asociadas como coordinador ZigBee (dependiendo del tipo de petición).
- **Router:** responde a peticiones enviando su dirección IEEE o NWK y las direcciones IEEE o NWK de todos los dispositivos que tiene asociados como router ZigBee (dependiendo de la petición).

Servicio de Descubrimiento

El servicio de descubrimiento es el proceso por el cual los servicios que en un instante de tiempo están disponibles en los terminales o en los dispositivos receptores y que son descubiertos por dispositivos externos. El servicio de descubrimiento realiza las peticiones de sondeo para cada terminal de cada dispositivo o por el uso de servicios de sondeo tipo broadcast o unicast. El proceso del servicio de descubrimiento en ZigBee es la clave para interconectar dispositivos dentro de una red. A través de dichas peticiones de los descriptores de cada nodo especificado, las peticiones por broadcast para preguntar a los dispositivos cuales son los objetos de aplicación que tienen sus dispositivos.

2.1.2.6 Enlace

En ZigBee, hay un concepto de nivel de aplicación que utiliza los identificadores de clusters en los terminales de manera individual en cada uno de los nodos. Se llama enlace a la creación de un vínculo entre los dispositivos de aplicación de la red y los terminales. La información de cómo los clusters se emparejan con los nodos se indica en una tabla de enlace (binding table).

El enlace se lleva a cabo después de que el enlace de comunicaciones se haya establecido. Una vez el enlace se establece es ya en la implementación en la que se decide de qué manera un nodo puede llegar a formar parte de la red o no. Además, también depende de la seguridad definida para realizar la operación y de cómo se haya implementado. El enlace sólo se permite si la implementación de la seguridad de la red de todos y cada uno de los dispositivos lo permite.

La tabla de enlace se implementa en el coordinador ZigBee. Esto es porque se necesita que la red esté continuamente operativa y disponible, con lo que es más probable que el coordinador sea el que pueda ofrecer este servicio. Por otro lado, algunas aplicaciones pueden necesitar tener esta tabla de enlace duplicada, para que esté disponible por si ocurre un fallo de almacenamiento de la tabla original. Las copias de seguridad de la tabla de enlace y/o de otros elementos de datos sobre el coordinador ZigBee ya no pertenecen a la especificación de ZigBee 1.0, por lo que es responsabilidad del software de aplicación.

2.1.2.7 Mensajes

- Direccionamiento Directo

Una vez los dispositivos se han asociado, las instrucciones entre los elementos se pueden enviar y recibir, de forma que ya pueden ser enviadas de un dispositivo a otro. El direccionamiento directo asume que el descubrimiento del dispositivo y el servicio de descubrimiento tienen identificados un dispositivo con un terminal, el cual quiere realizar peticiones de servicios. El direccionamiento directo define una manera de realizar el direccionamiento en el que se envíen mensajes a los dispositivos incluyendo su dirección y la información de los terminales que contiene.

- Direccionamiento Indirecto

El uso del direccionamiento directo requiere de un dispositivo controlador que tenga el conocimiento de todas las direcciones, de los terminales, de los clusters identificadores y de los atributos identificadores de un dispositivo que quiere comunicarse y que tiene su información almacenada en una tabla de enlace (binding table) en un coordinador ZigBee. Este coste de almacenamiento es mayor que el que se produce en la creación de un mensaje de direccionamiento indirecto entre pares de dispositivos. La dirección IEEE compuesta de 10bytes además de un byte adicional que se necesita son suficientes para que dispositivos sencillos como pudieran ser interruptores (switches) alimentados por baterías, se sobrecargarían al almacenar toda la información que hay en la tabla. Para estos dispositivos, el direccionamiento indirecto resulta más adecuado.

Cuando un dispositivo fuente/emisor contiene varios atributos, el identificador de cluster se utiliza para realizar el direccionamiento y los atributos identificadores se usan para identificar un atributo en particular incluido en el cluster.

- Direccionamiento Broadcast

Una aplicación puede enviar mensajes broadcast a todos los terminales de un dispositivo dado. Este direccionamiento forma parte del direccionamiento broadcast llamado broadcast de aplicación. La dirección de destino está formada por 16 bits de la dirección broadcast de la red y hay que indicar el flan de broadcast en la trama APS dentro del campo de control. El origen debe incluir el identificador de cluster, el perfil identificador y el campo del Terminal origen en la trama APS.

2.1.2.8 Dispositivos ZigBee

Coordinador

- Inicialización

Normalmente se crea una única copia de los parámetros de configuración de la red para los objetos pertenecientes a los ZDO. Además, se pueden definir parámetros para describir el Node Descriptor, Power Descriptor, Simple Descriptor, e incluso los terminales activos.

La aplicación del dispositivo realiza una petición en la lista de canales para realizar una búsqueda o escaneo de los canales indicados. La confirmación resultante obtiene una lista detallada de los PANs activos. La aplicación del dispositivo compara la lista de canales con la lista de red y selecciona uno de los canales que se encuentre libre. Una vez se identifica el canal, la aplicación del dispositivo selecciona los atributos de seguridad de la capa y trama correspondientes a los parámetros de configuración. Después la aplicación chequea si se ha podido establecer el PAN en el canal.

- Operación Normal

En este estado, el coordinador ZigBee debe permitir que otros dispositivos se unan a la red basándose en sus parámetros de configuración; como pudieran ser la duración de la incorporación del dispositivo a la red o el número máximo de elementos que se pueden unir.

El coordinador ZigBee debe responder a cualquier dispositivo u operaciones del servicio de descubrimiento de su propio dispositivo o de cualquier dispositivo que tenga asociado y que esté dormido. La aplicación del dispositivo debe asegurarse de que el número de entradas de enlace no excede de los indicados en los parámetros de configuración. Por tanto, el coordinador ZigBee tiene que soportar el control del proceso de incorporación a la red de cualquier dispositivo.

El coordinador tiene que mantener una lista de los dispositivos asociados y facilitar el soporte para elementos huérfanos, permitiendo que se vuelvan a unir a la red, permitiendo que los dispositivos se incorporen directamente en la red. Por otro lado, el coordinador ZigBee debe soportar primitivas que permitan eliminar o desasociar los dispositivos que estén bajo su control. El coordinador procesa las peticiones de solicitud del router o de los dispositivos finales. Una vez recibida la solicitud de desconexión el coordinador espera un tiempo para recibir una segunda petición de desconexión. Si le llega en un tiempo determinado, el coordinador ZigBee pasará a examinar el

identificador del perfil (Profile ID) para ver si coincide. Si coincide, lo incluye en una lista llamada AppOutClusterList para que deje de pertenecer a la red. Si no coincide se enviará un error al dispositivo que solicita la desconexión, es decir, el dispositivo seguirá perteneciendo a la red.

- Operación del Centro de Validación

El coordinador ZigBee tiene la función de ser el Centro de Validación (Trust Center) cuando la seguridad está habilitada en la red. Al centro de validación se le notifica si existen nuevos dispositivos en la red por medio del APSME. El centro de validación puede permitir que el dispositivo permanezca en la red o bien se le fuerce a salir de ella. Si el centro de validación decide permitir que el dispositivo permanezca en la red, debe establecer una clave maestra con el dispositivo a no ser que ya exista una clave maestra previa entre ellos. Una vez intercambiada dicha clave, el centro de validación y el dispositivo ahora negociarán una clave para establecer la conexión. El centro de validación entonces proporciona al dispositivo la clave de red (NWK) para que el dispositivo pueda establecer peticiones al coordinador.

Router

- Inicialización

Por regla general se crea una única copia de los parámetros de configuración de la red para los objetos pertenecientes a ZDO. Si se puede, se crean los elementos de configuración para el Complex Descriptor, el User Descriptor, el número máximo de entradas de enlace y la clave maestra.

La aplicación del dispositivo utiliza el ChannelList y sus parámetros de configuración para buscar o escanear los canales que se le indiquen. El resultado permite obtener la lista de red con los PAN activos en la red. Entonces se realizan varias peticiones de descubrimiento para obtener cuales son realmente los elementos que existen en la red y asociar los enlaces en la capa de red. La aplicación del dispositivo compara el ChannelList con la NetworkList para seleccionar los PAN existentes que se deben unir. Una vez que el PAN al que el dispositivo desea unirse se ha identificado, la aplicación del dispositivo debe realizar una petición para asociar el PAN en el canal. Después debe chequear el estado de verificación de la asociación en el coordinador u otros routers seleccionados en ese PAN.

Si la red tiene la seguridad activada, el dispositivo tiene que esperar a que el centro de validación le proporcione la clave maestra y establecer con éste la clave de enlace. Una vez establecido espera a que el centro de validación le pase la clave de red. Ahora ya que está autenticado puede funcionar como un router de la red.

- Operación Normal

En este estado, el router debe permitir que otros dispositivos se unen a la red basándose en los parámetros de configuración que tiene, como el número de elementos máximos o el tiempo en el que puede estar un elemento en la red. Cuando un dispositivo

nuevo se une a la red, la aplicación del dispositivo debe ser informada. Cuando se haya admitido en el PAN, el router debe indicarle la confirmación de la conexión. Si la seguridad está habilitada, el dispositivo debe informar al centro de validación. El router ZigBee debe responder a cualquier dispositivo descubierto o a operaciones del servicio de descubrimiento, tanto de su propio dispositivo como de cualquier otro asociado que pudiera estar dormido.

Si la seguridad está activada, el router debe utilizar la clave maestra para establecer los procedimientos para la gestión de la clave de enlace (Link Keys). El router debe soportar el establecimiento de una clave maestra con el dispositivo remoto y establecer entonces la clave de enlace. El router tiene que poder almacenar y eliminar las claves de enlace para destinos conocidos que requieran que la comunicación sea segura con lo que debe poder recibir las claves del centro de validación. El router debe permitir también la eliminación de la red de dispositivos asociados bajo su control de aplicación.

El router mantiene una lista con los dispositivos asociados y tiene que facilitar el soporte para que los procesos de búsqueda e incorporación de elementos huérfanos de los dispositivos que previamente han estado asociados, puedan volver a unirse a la red.

Dispositivo Final

- Inicialización

La aplicación del dispositivo debe obtener de la lista de canales la configuración para escanear los canales especificados. El resultado debe contener una lista de red (Network List) detallando los PAN activos en la red. Al igual que el router, se realizan varias peticiones de descubrimiento para saber cuantos elementos son los que hay en la red. La aplicación del dispositivo debe comparar la lista de canales con la lista de red para deducir a qué red debe unirse. En el algoritmo debe indicarse entre otras cosas: el modo de operación de la red, identificación del router o coordinador de la red, capacidad del router o coordinador, coste de enrutamiento, etc. Una vez hecho, debe chequear la asociación con el router o el coordinador ZigBee en el PAN. Si la red tiene la seguridad habilitada, el dispositivo tiene que esperar a que el centro de validación negocie primero la clave maestra, seguido de la clave de enlace y finalmente la clave de red (NWK), tras lo que se considerará que estará autenticado y listo para unirse a la red.

- Operación Normal

El dispositivo final ZigBee debe responder a cualquier dispositivo descubierto o a las peticiones de operación del servicio de descubrimiento de su propio dispositivo. Si la seguridad está habilitada, igual que en el apartado anterior, debe negociar primero la clave maestra y seguidamente la clave de enlace, con lo que tiene que poder almacenar también las claves de enlace de los destinos que quieran una comunicación segura. Debe poder gestionar estas claves, tanto para almacenar como para eliminar. Por tanto tiene que poder mantener una comunicación con el centro de validación para actualizar las claves de red (NWK key).

Dispositivos de Gestión

- Gestor de Seguridad

El gestor de seguridad determina que seguridad está habilitada o deshabilitada. Si está habilitada debe permitir:

- Establecer una clave.
- Transportar la clave.
- Autenticación.

- Gestor de Enlace

La función de gestión del enlace soporta:

- El enlace de los dispositivos finales.
- El enlace y desenlace.

- Gestor de Red

La función del gestor de la red debe soportar:

- El descubrimiento de la red.
- La formación de la red.
- Permitir y denegar asociaciones.
- Asociaciones y desasociaciones.
- Descubrimiento de rutas.
- Reseteo de la red.
- Habilitación e Inhabilitación del estado del receptor de radio.

- Gestor de Nodos

El gestor de nodos soporta la petición y respuesta de la funciones de gestión. Estas funciones de gestión solo proporcionan visibilidad a dispositivos externos en cuanto al estado del dispositivo receptor de la petición.

2.1.3 Capa de Red

Las primitivas de confirmación de la capa de red, suelen incluir parámetros encargados de informar acerca del estado de las solicitudes que genera la capa inmediatamente superior, la capa de aplicación. Estos parámetros son los que aparecen en la siguiente tabla.

Nombre	Valor	Descripción
SUCCESS	0x00	La solicitud ha finalizado correctamente.
INVALID_PARAMETER	0xc1	Parámetro inválido.
INVALID_REQUEST	0xc2	La solicitud se deniega en función del estado actual de la capa de red.
NOT_PERMITTED	0xc3	Solicitud no permitida.
STARTUP_FAILURE	0xc4	Fallo en la inicialización de la red.
ALREADY_PRESENT	0xc5	Indica que un dispositivo que ya existe en la red, dispone de la dirección que se pretende obtener.
SYNC_FAILURE	0xc6	Fallo de sincronización (problema con la MAC).
TABLE_FULL	0xc7	Indica que no dispone de más espacio para almacenar direcciones de dispositivos en la tabla de encaminamiento.
UNKNOWN_DEVICE	0xc8	Error porque el dispositivo indicado no aparece en la tabla de encaminamiento del dispositivo.
UNSUPPORTED_ATTRIBUTE	0xc9	Identificador de atributo no soportado o reconocido.
NO_NETWORKS	0xca	Fallo provocado por la inexistencia de redes disponibles.
LEAVE_UNCONFIRMED	0xcb	Fallo en el descubrimiento del propio dispositivo al resto de la red.
MAX_FRM_CNTR	0xcc	Proceso de Seguridad. Trama fuera de rango
NO_KEY	0xcd	Proceso de Seguridad. La solicitud carece de llave de paso.
BAD_CCM_OUTPUT	0xce	Proceso de Seguridad. El sistema de seguridad ha producido errores en su salida.

Tabla2: Primitivas de Confirmación

2.1.3.1 Descripción General

La capa de red es necesaria para ofrecer servicios a la capa inmediatamente superior, la capa de Aplicación, que permitan realizar operaciones sobre la capa inmediatamente inferior a la misma, la sub-capa de MAC, definida en el IEEE 802.15.4-2003. Es decir, la capa de red hace de interfaz entre la capa de Aplicación y la de MAC. Para esto, la capa de red dispone en esta interfaz de dos servicios, con los que cubre las necesidades de la capa de Aplicación.

Estos dos servicios se conocen como Servicio de Datos y Servicio de Control. La comunicación entre la capa de Aplicación y la sub-capa MAC, se lleva a cabo en el SAP de la capa de Red utilizando las interfaces descritas anteriormente. Esto se traduce de forma que, entre la capa de Aplicación y la de Red existen dos SAP, uno por cada servicio que la capa de Red oferta a la de Aplicación. De la misma forma que aparecen otros dos SAP más entre la capa de Red y la sub-capa de MAC.

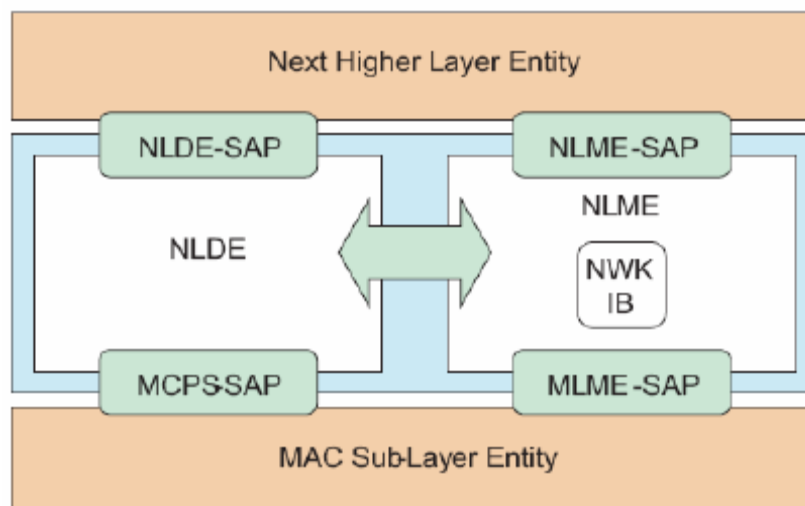


Figura9: Capa de Red

Servicio de Datos

Este servicio de interfaz, es también conocido con NLDE (Network Layer Data Entity). Provee de un servicio de datos que permite a cualquier aplicación comunicarse con las mismas unidades de datos, con dos o más dispositivos. Obviamente todos los dispositivos que intervengan en esta comunicación deberán estar en la misma red de interconexión. Esta interfaz dispone de los siguientes servicios:

- Generación de la PDU de la capa de Red (NPDU).
- Especificación de la topología de encaminamiento.

Servicio de Control

El también conocido como NLME (Network Layer Management Entity), es un servicio ofertado desde la capa de Red a la superior, que permite a la capa de Aplicación interactuar o comunicarse con la pila directamente. Esta interfaz dispone de los siguientes servicios:

- Configuración de un nuevo dispositivo. Esto permitirá la inicialización de un dispositivo Coordinador, así como el descubrimiento de nuevos dispositivos dentro de la red de interconexión.
 - Inicialización de una nueva red.
 - Integración y salida de una red.
 - Direccionamiento.
 - Descubrimiento de vecinos.
 - Descubrimiento de ruta.
 - Recepción de control.

2.1.3.2 Especificación del Servicio

Además de la posibilidad de comunicación entre la capa de Aplicación y la de Red, la capa de Red de ZigBee dispone de un canal de comunicación directa entre los

servicios de la misma capa. Es decir, dispone de una interfaz para comunicar los servicios intermedios de Datos y de Control. Mediante esta nueva interfaz, el servicio de Control podrá utilizar los servicios de su capa contigua, la de Datos.

Dentro de cada uno de los servicios de la capa de Red, en las interfaces de comunicación, se definen las primitivas de comunicación entre las capas de Aplicación y de MAC. De la misma forma que sucede en la comunicación entre los servicios de la propia capa. Estas primitivas son las siguientes:

- **Formación de Red.** Las primitivas que aquí se engloban, definen como la capa superior de un dispositivo ZigBee puede inicializarse a sí mismo como dispositivo coordinador de una nueva red.

- **Admisión de Dispositivos.** Grupo de primitivas que permiten tanto a un dispositivo coordinador como a un router la posibilidad de incorporar dispositivos a su red, mediante descubrimiento de los mismos.

- **Conversión a Router.** Estas primitivas son las que utiliza un dispositivo ZigBee tipo router, tras haber sido admitido en una nueva red, para ejercer como router en la misma, reconfigurando su trama para esto.

- **Incorporación a una Red.** Se trata de una serie de primitivas utilizadas para la incorporación de dispositivos a una red ZigBee. Dentro se clasifican en tres grupos.

- **Incorporación a una red por asociación.** Cuando un dispositivo pretende entrar a formar parte de la red del vecino más cercano que ha encontrado.

- **Incorporación a una red directamente.**

- **Reincorporación a una red.** Esto sucede en el caso de que un dispositivo se despierte y no sea capaz de encontrar su red.

- **Incorporación directa de Dispositivos.** Se trata de una serie de primitivas que permiten tanto a los routers como coordinadores de una red ZigBee la incorporación directa de un dispositivo a su red, sin necesidad de que este dispositivo lo solicite.

- **Abandonar una Red.** Grupo de primitivas utilizadas por los dispositivos para abandonar la red a la que pertenecen. También pueden ser utilizadas por otros dispositivos vecinos para informar al coordinador o router de que algún dispositivo pretende abandonar la red. Así mismo estas primitivas las utilizan los coordinadores para notificar al dispositivo en cuestión, que ha abandonado correctamente la red.

- **Reseteo de Dispositivos.** Primitivas utilizadas para que los dispositivos puedan resetear su capa de red.

- **Sincronización.** Juego de primitivas que los dispositivos utilizan para sincronizar su comunicación con los dispositivos coordinadores o routers.

- **Mantenimiento de la capa de Red.** Este último grupo de primitivas es utilizada por la capa superior de los dispositivos para leer y escribir en la base de información de la capa de red.

2.1.3.3 Funcionalidades

Todos los dispositivos ZigBee disponen de dos funcionalidades:

- Incorporación a una Red.
- Abandonar una red.

Además de estas funcionalidades, los dispositivos Coordinadores y Routers disponen de una serie de funcionalidades adicionales:

- Permitir a otros dispositivos incorporarse a la red. De dos formas distintas:
 - Por indicaciones de la sub-capa de MAC.
 - Por solicitud de incorporación desde la capa de Aplicación.
- Permitir a los dispositivos miembros de la red abandonarla. De la misma forma que sucedía en el caso anterior, dispone de dos posibilidades:
 - Por indicaciones de la sub-capa de MAC.
 - Por solicitud de incorporación desde la capa de Aplicación.
- Asignación de direcciones de red lógicas.
- Mantenimiento de una tabla o lista de dispositivos cercanos o vecinos.

Por último, los dispositivos Coordinadores, disponen de una funcionalidad particular. Esta es la que les permite crear o establecer nuevas redes de datos entre dispositivos.

Creación de una Nueva Red

Este procedimiento sólo puede ser iniciado por dispositivos Coordinadores, que no se encuentren ya dentro de una red ZigBee. Es decir, un coordinador, sólo puede aparecer en una red. Pero en el caso de que cualquier otro tipo de dispositivo o de que un Coordinador asociado ya a una red, iniciase este procedimiento sería denegado por la capa de Red.

Una vez iniciado el procedimiento, desde el interfaz de control de Red, se comunica con la subcapa de MAC para comprobar si existen posibles interferencias (otros coordinadores haciendo la misma operación por ejemplo). Esta comprobación se hace utilizando varios canales, hasta que se encuentra uno disponible, el cuál es reservado para la nueva red. En caso de que no se encuentre ningún canal disponible, se notificará a la capa superior y se abandonará el proceso de establecimiento de la red. Una vez encontrado un canal disponible, este es ocupado y se le asigna un nombre a la subred a partir del ID del PAN. El cuál obviamente no puede ser el de broadcast. Este parámetro es elegido aleatoriamente y siempre dejando 16bits disponibles, reservados para futuras ampliaciones de la red. Al finalizar esta secuencia, el nuevo ID es comunicado a la subcapa inferior (MAC).

Entonces, y sino aparecen conflictos con el ID del PAN, se escoge y establece la nueva dirección de red. Hecho esto, se notifica que el proceso ha finalizado correctamente y se inicializan los parámetros del coordinador en base a los parámetros de identificación obtenidos.

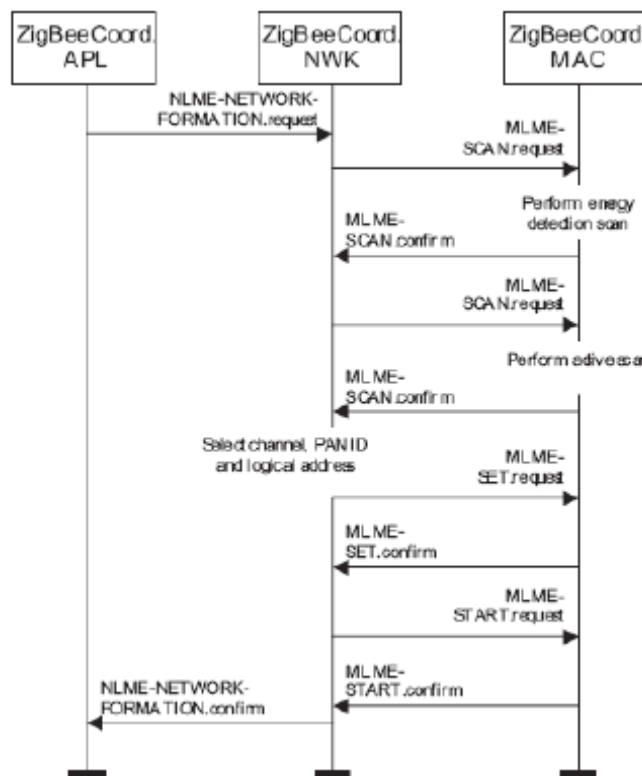


Figura 10: Creación de una nueva red

A continuación se muestra todo este proceso, en un diagrama de comunicación, en el que se puede apreciar además las primitivas utilizadas en todo el proceso.

Incorporación de Nuevos Dispositivos a la Red

Este procedimiento sólo puede ser iniciado por dispositivos ZigBee que sean Coordinador o Router. En caso de que otro dispositivo iniciase este proceso, sería cancelado por el Servicio de Control de la capa de Red.

Entonces se habilita el parámetro *PermitDuration* y la sub-capa de MAC se configura para permitir la asociación con nuevas direcciones MAC. Desde este momento, el dispositivo está esperando que nuevos dispositivos acepten su oferta para formar parte de la red. Este proceso no tiene una duración determinada, sólo finalizará en el caso de que aparezca otra orden o primitiva que la anule.

A continuación se muestra todo este proceso, en un diagrama de comunicación, en el que se puede apreciar además las primitivas utilizadas en todo el proceso.

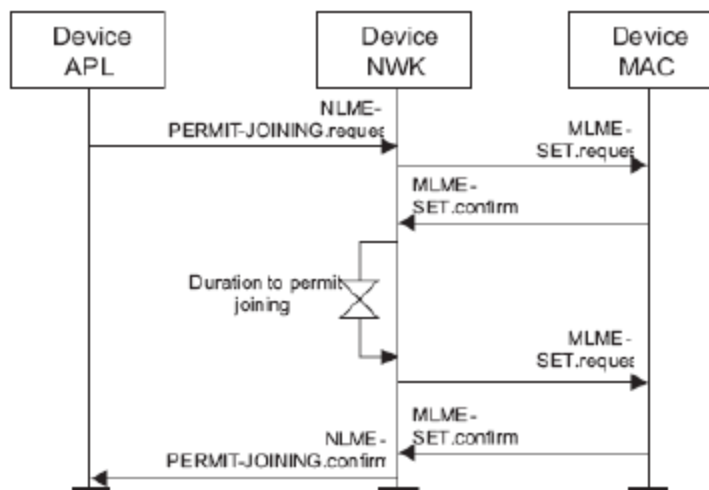


Figura 11: Incorporación de nuevos dispositivos a la red

Incorporación a una Red

En este momento aparece la diferenciación entre *padre* e *hijo*. Llamaremos padre al dispositivo que permite que otros dispositivos se conecten a su red, es decir, se tratará de un dispositivo Coordinador o en su defecto un Router. Mientras que el hijo pasará a ser el nuevo dispositivo que pretende formar parte de la red.

La incorporación a una nueva red, puede hacerse de dos formas distintas.

- Por asociación.
- Directamente.

También hay que tener en cuenta que un dispositivo puede reincorporarse a una red. Bien por haber estado dormido durante un largo periodo de tiempo o bien porque ha perdido su red y busca una nueva.

- Incorporación a una Red por Asociación

Para la incorporación de un dispositivo a una red por asociación, aparecen dos procesos distintos, pero paralelos. Estos son los procesos correspondientes al padre y al hijo.

- Procedimiento del Hijo

El procedimiento empieza cuando el dispositivo escanea desde la sub-cap de MAC los canales disponibles. Es decir, en este caso el dispositivo estará buscando canales en los que haya algún tipo de tráfico. Este proceso de escaneo por canales tendrá una duración determinada por canal, al contrario que sucedía en el caso anterior.

Una vez escogido el canal, se procesan las tramas encontradas en el mismo, en busca de alguna cuya longitud sea distinta de cero. Entonces el dispositivo comprueba si la comunicación efectivamente es entre dispositivos de tecnología ZigBee. De ser así localiza el identificador de la red.

A continuación, toda esta información es procesada por el dispositivo, el número de redes que ha encontrado, dispositivos cercanos, etc. Buscando, en cuál de todas las redes localizadas se le permite la incorporación. Pasando a trabajar con el identificador de dicha red.

En este momento, el dispositivo, también puede decidir desechar las redes encontradas y volver a analizar los canales de comunicación en busca de otras que cumplan sus necesidades. Si el dispositivo es un Router, deberá indicarlo en su siguiente comunicación, en la que intentará finalmente incorporarse a la red. Acto seguido, el dispositivo hace una lista individual de los dispositivos cercanos a él (vecinos). Para comprobar la distancia a la que se encuentra del padre. Si este es muy distante, podría acceder a través de la asociación con otros dispositivos, sólo si el coste de esta asociación no supera una distancia de tres dispositivos asociados. En caso de que esta condición no se cumpla, el proceso se anulará nuevamente.

En caso de encontrar un vecino, que cumpla las condiciones, la capa de MAC se habilita de forma que solicite una dirección de red. Este proceso puede fallar, por diversas causas, como que el dispositivo elegido desaparezca de la red. Si esto ocurriese el proceso sería anulado y se volvería a empezar. Si el proceso se completa satisfactoriamente, es decir, el dispositivo es aceptado en la red. Le es asignada una dirección de red de 16bits única en toda la red. Además se actualiza la tabla de dispositivos de su vecino, para que sepa que ese dispositivo ya forma parte de la red y que accederá a la misma a través de él.

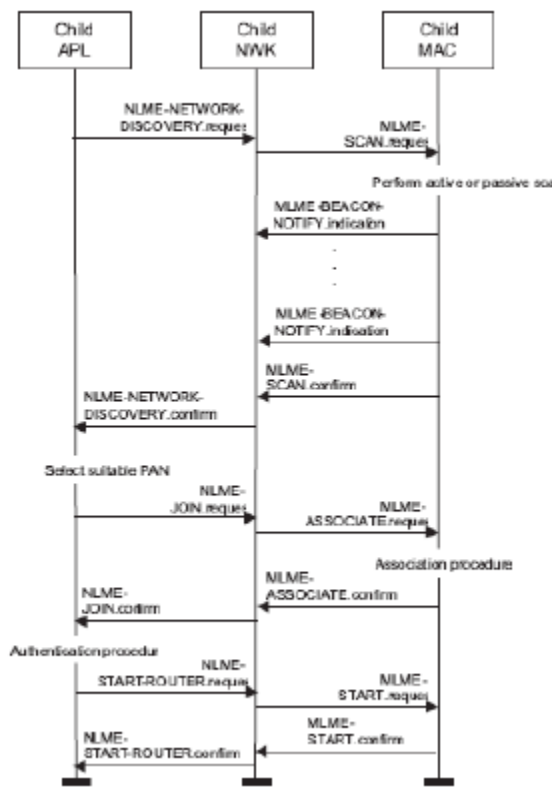


Figura 12: Incorporación a una red por Asociación (Hijo)

- Procedimiento del Padre

Este procedimiento es iniciado por la llegada de una solicitud de incorporación a la subcapa de MAC del dispositivo. Los dispositivos que pueden aceptar estos mensajes y permitir la incorporación a la red son sólo los Coordinadores y Routers. En caso de que otro dispositivo intente aceptar estos mensajes la capa de red los eliminará.

A continuación, el dispositivo comprueba por qué un dispositivo que ya es de su red, solicita su incorporación. Tras esto comprueba que lo que pretende el dispositivo es informar del descubrimiento de un nuevo dispositivo en la red cercano a si mismo. Entonces se asigna a este nuevo dispositivo su dirección lógica y única de red.

Aunque también puede darse el caso de que el dispositivo padre no disponga de espacio de memoria física para recordar a este nuevo dispositivo. Caso en el que la incorporación no podrá ser llevada a cabo y por lo tanto el proceso será anulado.

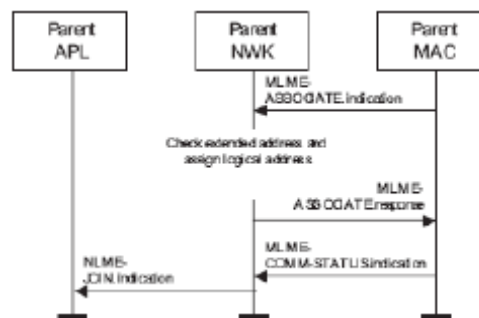


Figura 13: Incorporación a una red por Asociación (Padre)

- Incorporación a una red Directamente

En este caso el proceso de incorporación a una red es mucho más sencillo. Ya que la comunicación es directa entre padre e hijo, sin utilizar intermediarios. El hijo, una vez encontrada una red en la que un dispositivo Controlador o Router se encuentra próximo. Se le envía la petición de unión a la red. Si el dispositivo padre dispone de memoria física suficiente para almacenar la nueva dirección de este dispositivo. Genera una nueva dirección lógica de red, se la envía al nuevo dispositivo hijo y la almacena en su tabla en encaminamiento.

2.1.4 Capa MAC

2.1.4.1 Conceptos básicos

Las metas de un protocolo MAC para aplicaciones en WSN son:

- Operación en baja potencia.
- Evasión de colisiones efectiva.
- Implementación simple, Código pequeño en el tamaño de la RAM.
- Utilización del canal eficiente a bajas y altas tasas de datos.

- Reconfigurables para protocolos de red.
- Tolerantes a los cambios de RF /condiciones de la red.
- Escalable a un número grande de sensores.

Teniendo en cuenta el tipo de dispositivo inalámbrico que se utiliza en el proyecto, TelosB, el protocolo de control de acceso al medio no va a ser el incluido en IEEE 802.15.4, sino el protocolo B-MAC (Berkeley Media Access Control), creado por la Universidad de Berkeley para dichos dispositivos inalámbricos.

B-MAC nos ofrece un protocolo MAC reconfigurable para redes de sensores. Este es simple tanto en el diseño e implementación. Tiene un pequeño núcleo que excluye funcionalidades de capas superiores, lo que le permite soportar una amplia variedad de escenarios en la red.

2.1.4.2 B-MAC

B-MAC es un protocolo de acceso al medio con detección de portadora para RSI que provee una interfaz flexible para obtener operaciones de baja potencia. Para lograr operar en baja potencia B-MAC emplea un esquema de muestreo de preámbulo adaptativo para reducir el ciclo de trabajo y minimizar estados de escucha ociosos. B-MAC soporta reconfiguración y provee interfaces bidireccionales para optimizar el desempeño de throughput, latencia o conservación de la energía en los servicios. B-MAC utiliza evaluación de canal (Clear Channel Assessment, CCA) y retracción de paquetes para el proceso de acceso al medio, ACK de capa de enlace para la confiabilidad y escucha de baja potencia (LPL) para la comunicación.

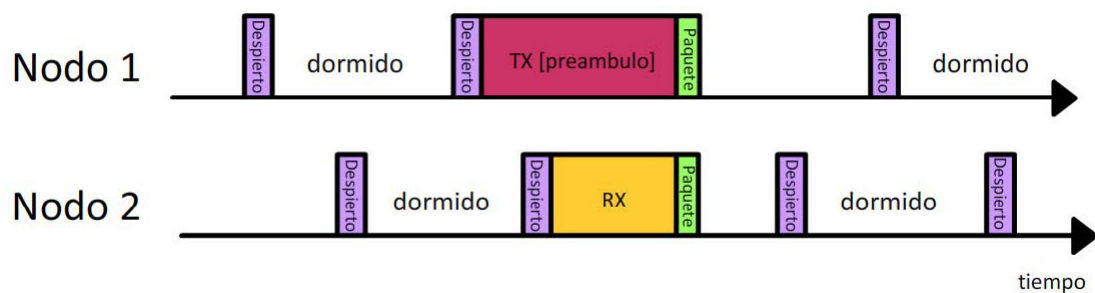


Figura 14: Uso de trama Preamble

B-MAC es solamente un protocolo de enlace con servicios de red como organización, sincronización y ruteo construidos sobre su implementación. Aunque B-MAC no provee mecanismos para el soporte de terminal oculta o fragmentación de mensajes, obliga una política particular de baja potencia.

2.1.4.3 Evaluación de canal libre (CCA, Clear Channel Assessment)

Para la evasión de colisiones efectiva, un protocolo MAC debe ser capaz de determinar si el canal está desocupado, determinado como Clear Channel Assessment (CCA). B-MAC emplea un software de control automático de ganancia para estimar el

nivel de ruido. La fuerza de la señal se muestrea cuando el canal se considera que está libre, es decir, inmediatamente después de una transmisión. Las muestras son almacenadas en una cola FIFO. El valor medio de la cola es agregado a un promediador de movimientos ponderados con decaimiento α . Una vez que se establece el nivel de ruido estimado, la solicitud de transmisión de un paquete inicia el testeado de la potencia de la señal recibida desde el radio. B-MAC busca muestras fuera de rango en la señal recibida, tal que la energía del canal esté significativamente por debajo del nivel de ruido. Si existen muestras fuera de rango durante el muestreo, B-MAC considera que el canal está libre ya que ningún paquete válido estaría por debajo del nivel de ruido. Si son tomadas 5 muestras y no se encuentran ninguna fuera de rango, el canal se considera ocupado.

La mayoría de los mecanismos básicos permiten servicios que habilitan o no la opción CCA usando la interfaz de MacControl (Figura 1). Al deshabilitar CCA se puede implementar un protocolo de programación sobre B-MAC. Por el contrario si se habilita CCA, B-MAC utiliza una vuelta al canal inicial cuando se envía un paquete. B-MAC no establece un tiempo de retracción, en vez de ello señala un evento al servicio que envía el paquete mediante la interface de MacBackoff. El servicio puede regresar en un tiempo de retracción inicial o ignorar el evento. Si es ignorado se utiliza un tiempo de retracción aleatorio muy pequeño. Después de la retracción inicial se corre el algoritmo CCA que busca muestras fuera de rango. Si el canal no está libre, un evento solicita al servicio un tiempo de retracción por congestión. Si no se da ese tiempo de retracción, nuevamente se utiliza una retracción aleatoria pequeña. Habilitar o deshabilitar CCA y configurar el tiempo de retracción permite a los servicios cambiar la equidad y throughput disponible. B-MAC provee un soporte opcional de ACK. Si el ACK es habilitado, B-MAC inmediatamente transfiere código ACK después de recibir un paquete unicast.

2.1.4.4 Escucha de baja potencia (LPL, Low Power Listening)

Utiliza una técnica similar al muestreo de preámbulo en ALOHA pero adaptado a las diferentes características del radio. Cada vez que el nodo despierta, enciende el radio y checa la actividad del canal. Si se detecta actividad, el nodo permanece despierto durante el tiempo necesario para recibir un paquete. Después de la recepción el nodo regresa a dormir. Si no se recibe ningún paquete (falso positivo), un temporizador obliga al nodo a dormirse. CCA es crítico para lograr una operación de baja potencia con este método. Debido a que la memoria RAM y ROM disponibles en redes de sensores son extremadamente limitadas, es necesario mantener un tamaño pequeño de la implementación, reduciendo la complejidad del protocolo. Gracias a que B-MAC no cuenta con un mecanismo RTS/CTS o los requerimientos de sincronización de SMAC, la implementación es más simple y pequeña como se muestra en la Tabla 1. B-MAC no dificulta la implementación eficiente de los protocolos de red, encima de B-MAC se pueden implementar esquemas de RTS/CTS y servicio de fragmentación de mensajes utilizando interfaces de control que tengan la funcionalidad equivalente.

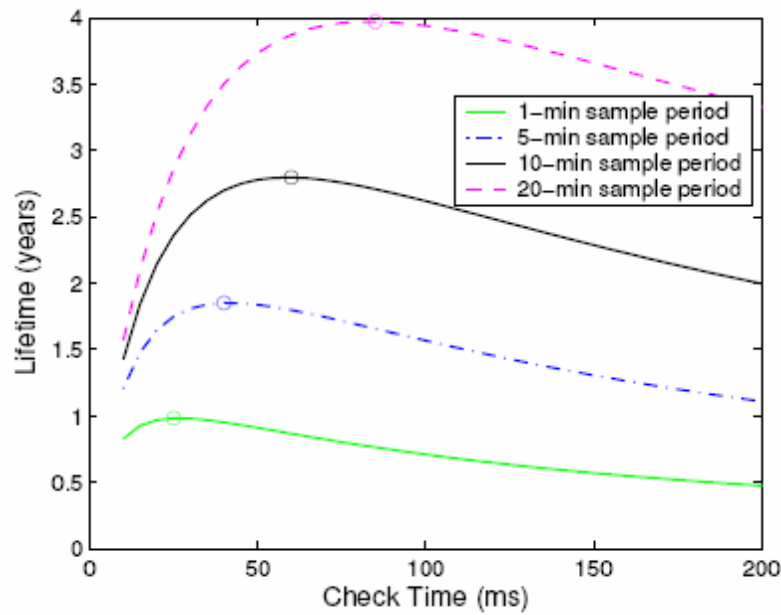


Figura 15: LPL

2.1.4.5 Ejemplos de funcionamiento del protocolo B-MAC

A continuación vamos a comprobar el funcionamiento del protocolo B-MAC ayudados por una serie de gráficas y tablas.

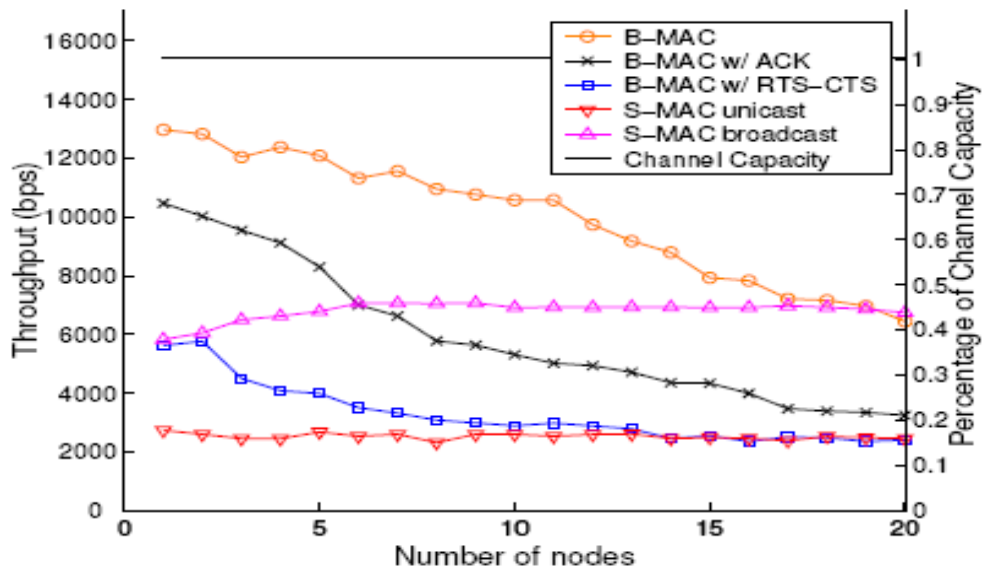


Figura 16: Utilización del canal

En la primera gráfica se muestra la utilización del canal con respecto al número de nodos de la red, comparando distintos protocolos MAC. Se puede observar como B-MAC en una de sus variantes es el que mayor tasa usa sin tener un alto nivel de utilización del canal

Length (bytes)	B-MAC	S-MAC
Preamble	8	18
Synchronization	2	2
Header	5	9
Footer (CRC)	2	2
Data Length	29	29
Total	46	60

Figura 17: Comparativa de la longitud de tramas

En la tabla superior se realiza una comparativa entre la longitud de las tramas de dos de los protocolos MAC mas utilizados, B-MAC y S-MAC; donde se observa un amplia diferencia en el tamaño de *Preamble*.

2.1.5 Especificación de los Servicios de Seguridad

2.1.5.1 Arquitectura de Seguridad

La seguridad en una red de dispositivos ZigBee se basa en claves de enlace y de red. En una comunicación por unicast entre pares de entidades APL la seguridad se basa en claves de 128 bits entre los dos dispositivos. Por otro lado, la comunicación existente cuando es por broadcast, también las claves para la seguridad se establecen de 128 bits entre todos los dispositivos de la red.

Un dispositivo adquiere la clave de enlace mediante el transporte de clave, establecimiento de clave o dada en la preinstalación desde el fabricante. Por otro lado, para el establecimiento de la clave de red hay dos maneras: el transporte de clave y la preinstalación. Como se ha mostrado en apartados anteriores el establecimiento de clave se obteniendo previamente una clave de enlace basándose en una clave maestra. Esta clave maestra puede ser obtenida por el transporte de dicha clave o en fábrica.

La clave de red tiene que ser usada por las capas MAC, NWK y APL de ZigBee. Las claves maestras y las de enlace solo pueden ser usadas en la subcapa APS, de hecho, las claves maestras y de enlace deben estar disponibles solo en la capa APL.

2.1.5.2 Seguridad MAC

Cuando una trama en la capa MAC tiene que ser asegurada, ZigBee tiene que usar la capa de seguridad que se indica en la especificación 802.15.4. La capa MAC se encarga de su propio proceso de seguridad aunque sean las capas superiores las encargadas de determinar el nivel de seguridad a usar. La siguiente figura muestra un ejemplo de los campos de seguridad que tienen que ser incluidos en las tramas en las que se indica que tiene que existir seguridad a nivel de MAC.

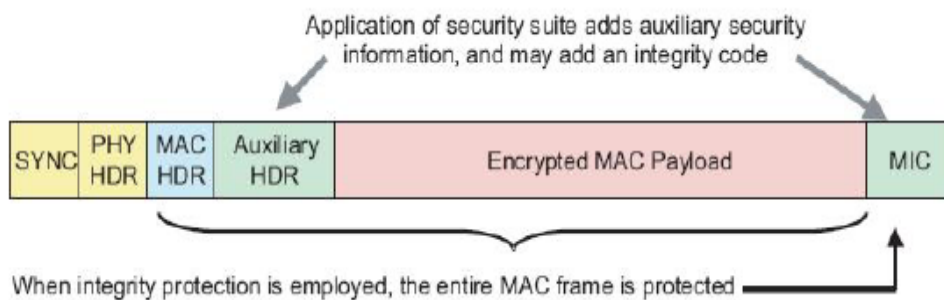


Figura 18: Seguridad en MAC

2.1.5.3 Seguridad NWK (Red)

Cuando una trama en la capa de red necesita ser asegurada, ZigBee debe usar ciertos mecanismos de protección de los datos. Al igual que la capa MAC, el mecanismo de protección de trama en la capa de red NWK usa de la encriptación *Advanced Encryption Standard*, es decir, AES. Sin embargo son las capas superiores las que deben indicar el nivel de seguridad que se tiene que aplicar.

Una responsabilidad de la capa de red (NWK) es enrutar los mensajes sobre enlace multi-hop. La capa de red tiene que enviar como broadcast sus peticiones de enrutado y recibir las respuestas. Se realiza de manera simultánea el enrutamiento de los mensajes de peticiones que se envían a los dispositivos cercanos y los que se reciben de ellos. Si la clave de enlace apropiada se indica, la capa de red usa esta clave de enlace para asegurar sus tramas de red. Si por el contrario no se indica, para poder asegurar los mensajes de la capa de red usa su propia clave de red para asegurar las tramas de red. Por tanto en el formato de la trama se indica de manera explícita la clave que se ha usado para protegerla. La siguiente figura muestra los campos que se deben incluir en una trama de red.

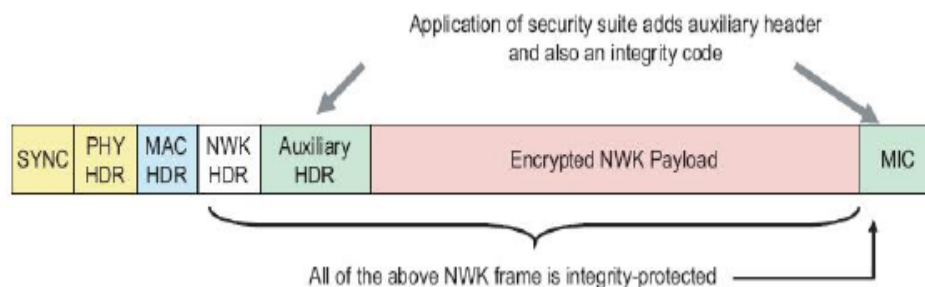


Figura 19: Seguridad en NWK

2.1.5.4 Seguridad en APL

Cuando una trama en la capa APL necesita ser asegurada, la subcapa APS es la encargada de gestionar dicha seguridad. La capa APS permite que la seguridad de trama

se base en las claves de enlace y de red (Link y Network Keys) como se ha visto en apartados anteriores. La siguiente figura muestra los campos para proporcionar seguridad en una trama del nivel APL.

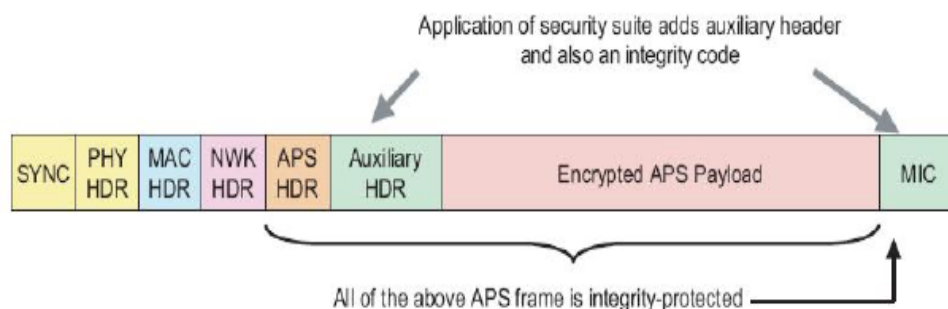


Figura 20: Seguridad en APL

2.1.5.5 Rol del Centro de Validación

Por temas de seguridad, ZigBee define el rol de Centro de Validación. Este elemento es un dispositivo validado por los dispositivos de la red para distribuir las claves para que gestione la configuración de aplicación de los dispositivos. Todos los miembros de la red deben reconocer solo a un centro de validación (Trust Center) y debe existir solo y solo un centro de validación por cada red segura.

Las funciones dadas por el Centro de Validación pueden ser subdivididas en tres roles: el gestor de la validación, el gestor de la red y el gestor de la configuración. Un dispositivo se encarga de validar el gestor de validación para identificar los dispositivos que toman el rol en dicha red y el gestor de configuración. El gestor de red se encarga de gestionar la clave de red, tanto para tenerla como para distribuirla. El gestor de configuración se encarga del enlace (binding) de dos aplicaciones y facilitar la seguridad entre estos dos dispositivos que gestiona, como por ejemplo distribuyendo las claves maestras o de enlace. Para simplificar el manejo de estos tres roles, se incluyen dentro de un único dispositivo, el centro de validación.

2.2 NesC

2.2.1 Introducción

NesC es un lenguaje de programación que se utiliza para crear aplicaciones que serán ejecutadas en sensores que ejecuten el sistema operativo de *TinyOs*, por tanto dicho lenguaje proporciona ciertas características necesarias para poder realizar aplicaciones de una forma más cómoda para el programador. Concretamente, se basa en una programación orientada a componentes, esto es, una aplicación se crea ensamblando componentes. Esta filosofía permite abstraer al programador de bastantes detalles de baja implementación presentes en el sistema operativo.

La idea que hay detrás de este tipo de programación, es que el propio sistema operativo, en conjunción con las casas que venden los dispositivos de sensores, proporcionan de forma intrínseca ciertos componentes ya implementados que ofrecen al programador un montón de funciones y utilidades para que el programador de este tipo de dispositivos pueda utilizar dichos componentes y centrarse solo en programar la funcionalidad que desea en el dispositivo sin necesidad de tener que preocuparse por todos estos aspectos.

Por otro lado, *NesC* combina ciertos aspectos de la orientación a objetos, basándose en una programación orientada a interfaces y de la orientación a eventos, ya que posee un manejo de eventos propio de este tipo de lenguajes como Visual Basic.

Todos los componentes que ofrece de forma intrínseca el sistema operativo se van a denominar a partir de ahora componentes primitivos y los componentes proporcionados por terceros, contribuciones, librerías, aplicaciones se van a denominar componentes complejos.

El sentido que se quería resaltar con lo dicho anteriormente de que es un lenguaje orientado a objetos, es que todos los componentes ya sea primitivos o complejos proporcionan unas interfaces, y si un programador quiere utilizar un componente, la forma de hacerlo es usar dichas interfaces. Pero recordemos qué son interfaces en el sentido OO, por lo que en un momento dado se podría cambiar un componente por otro siempre y cuando este otro proporcionase la misma interfaz y ese cambio no afectaría al código de la implementación de la aplicación.

De esta explicación se puede entresacar, que se van a tener dos partes diferentes como mínimo para un componente, la parte de implementación que estará programada hacia componentes y la parte de configuración que permitirá decidir que componentes son los que estoy utilizando para proporcionar dichas interfaces a mi componente, esto como veremos más adelante se denomina wiring. En cuanto a la orientación a eventos, decir que básicamente la forma de programar la aplicación no es del todo secuencial, sino que atiende a una programación basada en eventos, de manera que se programan las acciones que se desea realizar cuando se produzca un evento y este fragmento se ejecutará exactamente cada vez que se lleve a cabo dicho evento.

2.2.2 Estructura de un Componente

Un componente desde el punto de vista de programación está compuesto por varias secciones y el conjunto de todas ellas dan lugar a la creación de dicho componente.

Por tanto, primero para no perder el norte, vamos a empezar definiendo el convenio que es utilizado para organizar dichas secciones por *TinyOs*. En general, un componente posee tres grandes secciones que son: *Configuration*, *Implementation*, *Module*. Estas tres secciones han de estar obligatoriamente presentes en cualquier componente aunque puedan estar vacías.

El estándar de *TinyOs* determina, que las secciones de *Configuration e Implementation* han de ir en un fichero que recibirá el nombre del componente con la extensión *.nc* y la tercera sección de *Module* deberá de ir en otro fichero aparte que recibirá el nombre del componente concatenado con un M mayúscula (la M da el significado al fichero, es el significado de *Module*), este último fichero también poseerá extensión *.nc*.

Otra buena costumbre consiste en crear un fichero de *header* o cabecera con extensión *.h* que contenga todas las enumeraciones, registros o tipos de datos creados por el usuario de los que hace uso la aplicación, y cuando se realiza esto la forma de ligar dicho fichero con los otros dos es utilizando al principio de los otros fichero la directiva *includes header*. Aunque hay que decir que si nos fijamos mejor en este directiva se puede ver que no se incorpora la extensión *.h* en la misma.

Ahora que ya se conoce cuales son las secciones que va a contener cada fichero vamos a empezar a explicar cada una de ellas.

Implementación

Esta sección se va a encargar de definir las conexiones que hay entre los diferentes componentes que utiliza la aplicación, esto es debido a que si recordamos un poco, se ha comentado que la programación de un componente (que se llevará a cabo en la sección de *module*) se hace utilizando interfaces y dichas interfaces para poder utilizarlas las ha de proporcionar un componente, entonces básicamente es esta sección se definen cuales son los componentes que proporcionan las interfaces a nuestra aplicación (por lo general serán *componentes primitivos*).

Una vez que conocemos la finalidad de esta sección y llegados a este punto, vamos a insertar un concepto nuevo que es la diferencia que existe entre una aplicación que esta ya disponible para ser ejecutada en un sensor y un componente cualquiera. La diferencia es muy poca, y consiste en que una aplicación es un componente como cualquier cosa en este lenguaje que en su sección de implementación hace uso de un componente especial denominado *Main*.

Profundizando un poco más si necesitamos crear un componente para un sensor, por lo general tenemos que utilizar las interfaces que nos proporcionan otros *componentes primitivos o no* y en definitiva para cada una de estas interfaces que nosotros utilizemos en la creación de nuestro componente se han de definir obligatoriamente relaciones con las componentes que proporcionen dichas interfaces, al proceso de definir estas relaciones se le conoce como wiring.

Ahora veamos un poco como podemos definir estas relaciones, la sección de implementación constan de dos partes, la primera es una declaración de intenciones en la que se especifican todos los componentes que va a utilizar nuestra aplicación. Esta declaración se realiza mediante la palabra reservada *components* después de la cual van separados por una coma simple, todos los componentes.

La segunda parte de esta sección corresponde a la definición de las relaciones que hay entre las interfaces que utiliza nuestra aplicación y las interfaces que proporcionan los componentes; y la forma de definir que la interfaz que proporciona un componente es la que se corresponde a la que estoy utilizando en mi aplicación es la siguiente:

```
componete.interfaz -> miaplicación.interfaz
```

Código 1. Definición de interfaz que proporciona un componente

Pero ahora nos surge un problema y se da en el caso en que mi aplicación desee utilizar dos interfaces iguales, es decir que poseen el mismo nombre, pero que están proporcionadas por componentes diferentes, pues bien, para que el lenguaje pueda permitir esto se pueda utilizar alias para las interfaces (dichos alias se especificarán en la sección *module*) y la forma de indicar en esta sección que la interfaz que proporciona un componente corresponde a una de las interfaces que utiliza mi aplicación sería:

```
componete.interfaz -> miaplicación.Aliasinterfaz
```

Código 2. Definición de interfaz que proporciona un componente que utiliza mi aplicación

Ahora bien, si queremos realizar una aplicación ejecutable en un sensor, como se ha indicado anteriormente se ha de utilizar el componente especial *Main* dicho componente proporciona una interfaz especial denominada *StdControl* y por tanto mi aplicación ha de obligatoriamente proporcionar dicha interfaz, aunque la manera de proporcionarla la veremos cuando lleguemos a la correspondiente sección, vamos a ver ahora como tendríamos que hacer el *wiring* y así nos valdrá como primer ejemplo.

```
implementation {  
components Main, MiAplicacionM;  
Main.StdControl -> MiAplicacionM.StdControl;  
}
```

Código 3. Wiring

Ahora veámoslo desde otra perspectiva, imaginemos que la única información que poseo es la que aparece en el ejemplo de arriba, de este yo podría asegurar que mi aplicación proporciona la interfaz *StdControl* y que además no utiliza ninguna otra por que si no debería de haber realizado el *wiring* del componente que proporcione esa interfaz con la interfaz de mi aplicación, lo que implica también que dicho componente debería de estar declarado en la directiva *components*.

Otra cosa que se puede dar en esta sección es que si poseemos un componente A que utiliza la interfaz IC que proporciona un componente B y el componente B utiliza la interfaz IC que proporciona el componente A, esto lo podemos resolver de dos formas la primera es con lo que ya sabemos lo que quedaría:

```
implementation {  
  components A, B;  
  A.IC -> B.IC;  
  B-IC -> A.IC;  
}
```

Código 4. Asignación

Sin embargo por motivos de claridad y para que la sección de *wiring* se haga lo más legible y entendible posible, el lenguaje *NesC* ofrece otro operador de conexión par dar esta semántica, es el operador =. Con lo que quedaría de la siguiente forma:

```
implementation {  
  components A, B;  
  A.IC = B.IC;  
}
```

Código 5. Uso del operador “=”

Introduzcamos ahora otro concepto muy utilizado en el *wiring* y que es vital para la correcta comprensión del mismo, además proporciona bastante potencia al lenguaje, es el concepto de *interfaces paramétricas*. Para entender su significado se va a empezar por poner un caso real, pensemos en un componente que se utiliza para recibir paquetes por el medio físico del aire, dicho componente proporciona una interfaz denominada *ReceiveMsg* para la recepción, y dicha interfaz obliga a quien la utilice a programa un evento que se producirá cuando se reciba un paquete. Ahora bien, pensemos que tenemos dos tipos de paquetes, uno que es de nuestra aplicación y otro que es de una aplicación que hemos comprado a terceros y estas dos aplicaciones han de coexistir en el mismo medio físico (el aire) entonces ¿cómo informo yo al componente qué cuando quiero recibir un paquete?, me refiero a los paquetes que son de mi aplicación. Para ello se utilizan las *interfaces paramétricas*, yo me crearé un identificador para mi tipo de mensaje que sea diferente al identificador utilizado para el otro tipo e informaré en la sección de *wiring* que utilizo la interfaz pero para este tipo de paquetes.

El objetivo se encuentra , en que el lenguaje de programación que usa *interfaces paramétricas* permite tener varias instancias de una interfaz proporcionadas por un mismo componente. La forma de indicarlo sería la siguiente:


```
implementation {  
  components GenericComm, //Permite envio/recepcion de paquetes  
  MiAplicacion;  
  MiAplicacion.RecibeMsg -> GenericComm.RecibeMsg[MI_MENSAJE];  
}
```

Codigo 6. Instanciar varias veces una interfaz

Configuración

Esta sección del componente ha de estar obligatoriamente presente pero solo contendrá algo en el caso en el que se pretenda crear un componente, no mediante su implementación de código directa (en la sección *Module*), sino mediante la composición de otros componentes ya creados. Es un mecanismo que ofrece el lenguaje de programación para poder crear un nuevo componente mediante la combinación directa de otros.

La estructura de esta sección es la misma que la de la sección de *Implementación* que acabamos de ver, además posee la misma semántica, un ejemplo podría ser:

```
Configuration MiAplicacion {  
  implementation {  
    components GenericComm, //Permite envio/recepcion de paquetes  
    TimerC; // Para realizar temporizaciones  
    TimerC.StdControl-> GenericComm.StdControl;  
  }  
}
```

Codigo 7. Ejemplo de componente *Configuration*

Module

Esta sección es la que por lo general, es más extensa y es en la que realmente se programa el comportamiento que se desea realizar en la aplicación. Esta a su vez, esta dividida en tres subsecciones : *Uses, Provide, Implementation*.

Antes de empezar con el significado de cada una de estas subsecciones vamos a situarlas dentro del fichero, para que se pueda apreciar la estructura del mismo:

```
module MiAplicacionM {  
  provides {... }  
  uses {... }  
}  
implementation {...}
```

Codigo 8. Estructura de componente *Module*

La primera subsección, *provides*, proporciona al lenguaje de programación, es decir, cuáles son las interfaces que va a proporcionar nuestro componente, en el caso de que nuestro componente sea una aplicación, como ya se vio anteriormente se ha de proporcionar como mínimo la interfaz *StdControl* (será explicada más

adelante). Pero ahora bien, cuando nosotros informamos de que vamos a proporcionar una interfaz, estamos diciendo desde un punto de vista orientado a objetos, que vamos a implementar dicha interfaz, por tanto, se deberán de implementar los métodos que obligue a implementar dicha interfaz.

Dentro de la sección *provide* la forma de anunciar que se va a proporcionar una interfaz es la siguiente;

```
provides {  
  interface interfazqueproporciono;  
}
```

Codigo 9. Ejemplo de cómo proporcionar una interfaz

La subsección *uses* informa al lenguaje de programación que voy a hacer uso de una interfaz, por tanto si hago uso de una interfaz podré realizar llamadas a los métodos de dicha interfaz.

Sin embargo, cuando nosotros informamos que vamos a utilizar una interfaz, se derivan ciertas acciones que tenemos que realizar para el correcto funcionamiento de nuestra aplicación. Primero, si utilizamos una interfaz, obligatoriamente en la sección de *implementation* debe de haber un *wiring* que conecte dicha interfaz con un componente que la proporcione. Segundo y último, el utilizar una interfaz conlleva implícitamente el tener que implementar los eventos que se puedan producir por el hecho de haber utilizado la interfaz.

La forma de indicar que vamos a utilizar una interfaz es la siguiente:

```
uses {  
  interface interfacequeutilizo;  
}
```

Codigo 10. Indicación de uso de interfaz

Veamos ahora la última subsección, quizás la más extensa, es la subsección *implementation*, esta contendrá todas las métodos necesarios para proporcionar el comportamiento desea a nuestro componente o aplicación. La estructura de la misma, es similar a la de un programa en lenguaje C pero con sutiles diferencias, aunque más bien que diferencia son añadidos para poder adaptar la programación al estilo de la programación orientada a eventos. Esta subsección ha de contener como mínimo

- Las variables globales que va a utilizar nuestra aplicación
- Las funciones que tenga que implementar debido a las interfaces que estoy proporcionando
- Los eventos que tenga que implementar debido a las interfaces de las que estoy realizando uso.

Así que, llegados a este punto vamos a profundizar un poquito más acerca de los diferentes tipos de datos que se nos proporcionan, los diferentes tipos de métodos, y algunas palabras reservadas nuevas.

2.2.3 Tipos de datos

Los tipos de datos que se pueden utilizar en *NesC* son todos los que proporciona *C estándar* más unos cuantos más, que en realidad no aportan potencia de cálculo pero son muy útiles para la construcción de paquetes ya que proporcionan al usuario información acerca del número de bits que ocupan y esto es importante a la hora de transmitir información vía radio.

Los tipos adicionales son:

- *uint16_t* que viene a ser un entero sin signo que ocupa 16 bits
- *uint8_t* que viene a ser un entero sin signo que ocupa 8 bits.
- *result_t*, se utiliza para devolver si una función se ha ejecutado con éxito o no, viene a ser como un booleano pero con los valores *SUCCESS* y *FAIL*
- *bool*, es un valor booleano que puede valer *TRUE* o *FALSE*

Como observación, comentar que si es posible la utilización de memoria dinámica pero no es muy recomendable a no ser que sea absolutamente necesaria. Para poder utilizarla existe un componente especial denominado *MemAlloc* que realiza una administración de la memoria dinámica.

Como después veremos existen diferentes tipos de funciones o métodos, uno de estos tipos son las denominadas funciones asíncronas, estas funciones no tienen por que realizarse de forma inmediata y por tanto cuando una de estas funciones utilice una variable global se ha de indicar de forma explícita para poder realizar una exclusión mutua de la memoria y no perder ningún valor. La forma de indicarlo es anteponer la palabra reservada *norace* delante de la declaración de la variable:

```
norace uint16_t temperatura;
```

Código 11. Definición de funciones asíncronas

2.2.4 Tipos de Funciones

En *NesC* las funciones pueden ser de tipos muy variados, primero existen las funciones clásicas con su misma semántica que en *C* y la forma de invocarlas es la misma que en *C*.

Ahora bien, existen otros tipos de funciones añadidos, estos son: *task*, *event*, *command* así que vamos a explicar sus diferencias y la forma de invocarlos. Las funciones *command* o comandos son básicamente funciones, que al igual que las clásicas se ejecutan de forma sincrónica, es decir que cuando son llamadas se produce su ejecución inmediatamente. La forma de llamar a una de estas funciones es:

```
call interfaz.nombreFuncion.
```

Código 12. Llamada a funciones

Las funciones *task* o tareas son funciones que se ejecutan concurrentemente en la aplicación, utilizan la misma filosofía que los *hilos* o *threads*, básicamente es una función normal que se invoca de la siguiente forma: *post interfaz.nombreTarea* y que inmediatamente después de su invocación, continua la ejecución del programa invocador.

Las funciones *event* o eventos son funciones que son llamadas cuando se levanta una señal en el sistema, básicamente poseen la misma filosofía que la programación orientada a eventos, de manera que cuando el componente recibe un evento se realizará la invocación de dicha función aunque existe un método para poder invocar manualmente este tipo de funciones, es : *signal interfaz.nombreEvento* , además en el caso de tratarse de interfaces parametrizadas poder utilizar la variante *signal*

```
interfaz.nombreEvento [parámetro]
```

Codigo 13. Llamada a interfaces parametrizadas

Ahora bien, estos tres tipos de funciones añadidos, por lo general poseen las características arriba mencionadas pero en su declaración se puede anteponer la palabra reservada *async* para indicar que poseen una ejecución asíncrona, este tipo de ejecución por lo general viene dado cuando la ejecución de uno de estos tipos de función viene determinada por el levantamiento de una señal hardware. Por ejemplo, que la temperatura ya se encuentre preparada para ser leída. En el caso de llamar a una de estas funciones asíncronas si se intenta acceder a una variable global, esta se ha de haber declarado con *norace* para indicar si exclusión mutua.

2.2.5 Otras funcionalidades de NesC

Debido a que se permite cierto tipo de programación recurrente mediante la invocación de tareas o *task* el lenguaje de programación permite construir una agrupación de sentencias que conformen una transacción, al estilo de bases de datos, para asegurar así que mientras que se este realizando dicha transacción, las variables implicadas no van a ser modificadas por otro hilo de ejecución. Esto se consigue mediante la palabra reservada *atomic* y la forma de usarlo es la siguiente:

```
atomic{  
... //Sentencias;  
}
```

Codigo 13. Uso de la sentencia *atomic*

Debido a que ciertas interfaces parametrizadas, no pueden ser llamadas dos veces con el mismo parámetro, existe una función *built-in* que proporciona un valor único para una constante, es decir si llamo a la función muchas veces con el mismo parámetro esta asegura que cada vez va a devolver un valor distinto. Dicha función es

```
uint_16 unique(string parámetro);
```

Codigo 14.

2.3 TinyOS

2.3.1 Introducción

En el presente documento se presentarán las características del sistema operativo “TinyOS”, desarrollado en la Universidad de Berkeley como sistema base para la construcción de aplicaciones en sistemas embebidos, específicamente redes de sensores inalámbricas (RSI).

El objetivo y alcance de este documento es discutir características que TinyOS posee, abarcando tópicos que van desde responder preguntas tales como ¿qué es? o ¿qué se necesita para su funcionamiento?, para pasar luego a ver la filosofía de su implementación y funcionamiento, su lenguaje de programación, el estudio de los modelos de ejecución y de “componentes”, hasta presentar a modo de resumen una lista de las componentes existentes y que pueden utilizarse para una implementación.

Durante este trabajo se mencionará bastante el término “componente”. La metodología de diseño en programación para el trabajo sobre este tipo de sistemas, es conocida como “programación orientada a componentes”, y se encuentra referido a la forma en que se conciben las aplicaciones, las cuales se diseñan de forma modular, definiendo entradas y salidas a ésta. Además siguiendo la idea de separar para lograr un mejor entendimiento y facilidad en el diseño, es que se definen dos tipos de componentes: los módulos, y de configuración.

2.3.2 Conociendo a TinyOS.

¿Qué es TinyOS?

TinyOS es un sistema operativo para trabajar con redes de sensores, desarrollado en la Universidad de Berkeley. TinyOS puede ser visto como un conjunto de programas avanzados, el cual cuenta con un amplio uso por parte de comunidades de desarrollo, dada sus características de ser un proyecto de código abierto (Open Source). Este “conjunto de programas” contiene numerosos algoritmos, que nos permitirán desde generar enrutamientos, como también aplicaciones pre-construidas para sensores. Además soporta diferentes plataformas de nodos de sensores, arquitecturas bases para el desarrollo de aplicaciones.

Como veremos más adelante, el lenguaje en el que se encuentra programado TinyOS es un meta-lenguaje que deriva de C, cuyo nombre es NesC. Además existen variadas herramientas que ayudan el estudio y desarrollo de aplicaciones para las redes de sensores, que van desde aplicaciones para la obtención y manejo de datos, hasta sistemas completos de simulación. Sin duda esta respuesta puede ampliarse bastante, y es objetivo de este documento presentar todas las características que TinyOS posee, y por ende lo definen.

¿Qué es necesario para su funcionamiento?

Aunque puede trabajar en otras plataformas, las plataformas apoyadas son Linux (RedHat 9), Windows 2000, y Windows XP. Es importante mencionar que el proyecto se inicio sobre BSD (Unix), por lo cual para el funcionamiento en Windows se hace necesario la instalación de *Cygwin*, un simulador de plataformas unix, siendo necesario tener básicos conocimientos acerca de este entorno. Además muchas de las herramientas disponibles se encuentran desarrolladas para plataformas JAVA, por tanto también es necesaria su instalación. La instalación de éstas en sistema Windows se realiza de forma transparente, puesto que existe un paquete ejecutable, el cual además realiza toda la configuración del sistema.

La programación de dispositivos se puede realizar a través de distintos puertos, dependiendo del módulo con el que se cuente. Estos pueden ser serial, paralelo o ethernet. Es entonces necesario contar con al menos uno de estos puertos en el PC.

2.3.3 Introducción a TinyOS.

El diseño de TinyOS está basado en responder a las características y necesidades de las redes de sensores, tales como reducido tamaño de memoria, bajo consumo de energía, operaciones de concurrencia intensiva, diversidad en diseños y usos, y finalmente operaciones robustas para facilitar el desarrollo confiable de aplicaciones. Además se encuentra optimizado en términos de uso de memoria y eficiencia de energía.

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 interrumpen, o atender las interrupciones de un conversor análogo-digital). Además pueden interrumpir las tareas que se están ejecutando.
- **Tareas:** Las tareas son pensadas para hacer una cantidad mayor de procesamiento y no son críticas en tiempo. 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.

Ahora bien, tal como se mencionaba en la sección 4.2.1, TinyOS se encuentra programado en NesC, un lenguaje diseñado para reflejar las ideas propias del enfoque

de componentes, incorporando además un modelo de programación que soporta concurrencia, manejo de comunicaciones y fácil interacción con el medio (manejo de hardware). TinyOS y NesC se encuentran profundamente relacionados, es por eso que a lo largo de este documento presentaremos un pequeño resumen con algunas de las características de este lenguaje.

2.3.4 TinyOS.

TinyOS posee un modelo de programación característico de los sistemas embebidos: el modelo de componentes. Tal como se ha mencionado TinyOS está escrito en NesC, lenguaje que surge para facilitar el desarrollo de este tipo de aplicaciones. A continuación se presentan las ideas de este modelo.

- **Arquitectura basada en componentes:** TinyOS se encuentra construido sobre un conjunto de componentes de sistema, las cuales proveen la base para la creación de aplicaciones. En la figura 1 se muestra el modelo de una componente. Conectar estas componentes, usando un conjunto de especificaciones detalladas, es lo que finalmente definirá una aplicación. Este modelo es esencial en sistemas embebidos para incrementar la confiabilidad, sin sacrificar desempeño.
- **Concurrencia en tareas y en eventos:** Las dos fuentes de concurrencia en TinyOS son las tareas y los eventos. Las componentes entregan tareas al planificador, siendo el retorno de éste de forma inmediata, aplazando el cómputo hasta que el planificador ejecute la tarea. Las componentes pueden realizar tareas siempre y cuando los requerimientos de tiempo no sean críticos. Para asegurar que el tiempo de espera no sea muy largo, es que se recomienda programar tareas cortas, y en caso de necesitar procesamientos mayores, se recomienda dividirlo en múltiples tareas. Las tareas se ejecutan en su totalidad, y no tiene prioridad sobre otras tareas o eventos. Así también los eventos hasta completarse, pero estos sí pueden interrumpir otros eventos o tareas, con el objetivo de cumplir de la mejor forma los requerimientos de tiempo real.
- **Todas las operaciones de larga duración deben ser divididas en dos estados:** la solicitud de la operación y la ejecución de ésta. Específicamente si un comando solicita la ejecución de una operación, éste debiese retornar inmediatamente mientras que la ejecución queda en mano del planificador, el cual deberá señalar a través de un evento, el éxito de la operación.
- **Una ventaja secundaria de elegir este modelo de programación,** es que propaga las abstracciones del hardware en el software. Tal como el hardware responde a cambios de estado en sus pines de entrada/salida, nuestras componentes responden a eventos y a los comandos en sus interfaces.

Ahora se tiene claro que las aplicaciones TinyOS son construidas por componentes. Un componente provee y usa interfaces. Estas interfaces son el único punto de acceso a la componente. Además una componente estará compuesta de un espacio de memoria y un conjunto de tareas.

A continuación se detallan los cuatro elementos que conforman una componente:

- Manejador de comandos.
- Manejador de eventos
- Un *frame* de tamaño fijo y estáticamente asignado, en el cual se representa el estado interno de la componente.
- Un bloque con tareas simples.

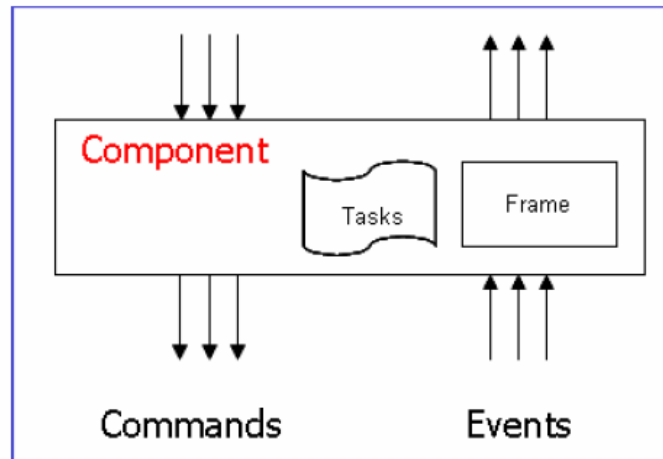


Figura 21: Modelo de componentes de TinyOS y su interacción.

Los comandos son peticiones hechas a componentes de capas inferiores. Estos generalmente son solicitados para ejecutar alguna operación. Existen dos posibles casos de uso de los comandos. El primero es para operaciones bifase, donde los comandos retornan inmediatamente, no generando bloqueos por la espera de la ejecución, es decir, una vez que realizan la petición, el planificador será el encargado de ejecutar lo solicitado, generando un evento que indique el fin de la operación. El otro es el caso de realizar una operación no bifase, donde ésta se realizará completamente, y por tanto no habrá evento retornado.

Los manejadores de eventos son invocados por eventos de componentes de capas inferiores, o por interrupciones cuando se está directamente conectado al hardware. Similar a los comandos, el *frame* será modificado y las tareas agregadas. Los eventos son capaces de interrumpir tareas, no así viceversa.

Las tareas se ejecutan hasta terminar y pueden ser sólo interrumpidas por eventos (podríamos decir que eventos tienen mayor prioridad). Las tareas son entregadas a un planificador (*task scheduler*) que en este caso está implementado con método FIFO. Debido a esta implementación, las tareas se ejecutan secuencialmente y no deben ser excesivamente largas. Alternativamente al planificador de tareas FIFO, puede ser reemplazado por planificadores basados en prioridades (priority-based) ó por planificadores basados en plazos (*deadline-based*), los cuales pueden ser implementados sobre TinyOS.

Tipos de Componentes.

En general las componentes se clasifican en una de estas categorías:

- *Abstracciones de Hardware*: Mapean el hardware físico en el modelo de componentes de TinyOS. Por ejemplo, en el módulo que se observa en la figura 2, la componente de radio RFM, es representativa de esta clase, ya que exporta comandos para manipular los pines de entrada-salida conectados al RFM y entrega eventos, informando a otras componentes acerca del bit de transmisión o recepción. Su *frame* contiene información acerca del estado actual de la componente (si se encuentra transmitiendo o recibiendo, la tasa de transferencia de bits, etc). El RFM detecta las interrupciones del hardware que se transforman en el bit de evento de RX o en el bit TX, dependiendo del modo de operación.
- *Hardware Sintético*: Los componentes de hardware sintéticos simulan el comportamiento del hardware avanzado. Un buen ejemplo de tal componente es el Radio Byte (véase el figura 2). Intercambia datos de entrada o salida del módulo RFM y señala cuando un byte ha sido completado. Las tareas internas realizan la codificación y decodificación de los datos. Conceptualmente, esta componente es una máquina de estado que podría ser directamente modelada en el hardware. Desde el punto de vista de los niveles superiores, esta componente provee una interfaz, funcionalmente muy similar a la componente de abstracción de hardware UART: proporcionan los mismos comandos y señalan los mismos eventos, se ocupan de datos del mismo tamaño, e internamente realizan tareas similares (buscando un bit o un símbolo de inicio, realizando una codificación simple, etc.).
- *Componente de alto nivel*: Realizan el control, enrutamientos y toda la transferencia de datos. Un representante de esta clase es el módulo de mensajes (“*messaging module*”), presentado en la figura 22. Éste realiza la función de llenar el buffer de los paquetes antes de la transmisión y envía mensajes recibidos a su lugar apropiado. Además, las componentes que realizan cálculos sobre los datos o su agregación, entran en esta categoría.

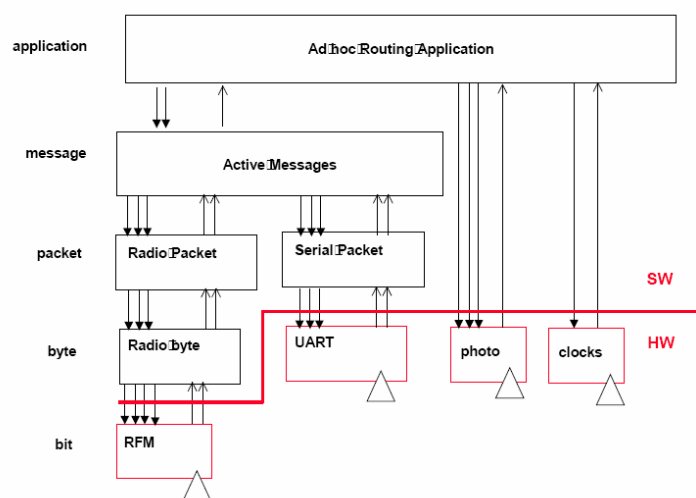


Figura 22: Grafo de componentes para un sistema MultiHop

Otra de las características de TinyOS es que el modelo de componentes permite la fácil migración a otro hardware, dada la abstracción de hardware que se logra con el modelo de manejo de eventos. Además, la migración es particularmente importante en las redes de sensores, ya que constantemente aparecen nuevas tecnologías, donde de seguro los diseñadores de sistemas querrán explorar el compromiso entre la integración a nivel físico, los requerimientos de energía, y el costo del sistema, requerimientos claves para la optimización.

Ejemplo de una componente.

Un componente típico que incluye un frame, eventos, comandos y tareas es mostrada en la figura 23.

Gráficamente, la componente se presenta como un conjunto de tareas, un bloque de estado (*frame* del componente), un conjunto de comandos (triángulos al revés), un conjunto de manejadores (triángulos), flechas para abajo para los comandos que utiliza, y flechas para arriba (prepicadas) para los comandos que señala.

En la figura 24 se muestra el código de esta componente. Como se puede observar el módulo principal se divide en dos partes: una que contiene las interfaces provistas y la otra que contiene las que utiliza. Es deber del programador hacer coincidir los formatos de eventos y comandos entre las distintas interfaces. De todas formas errores son verificados en tiempo de compilación.

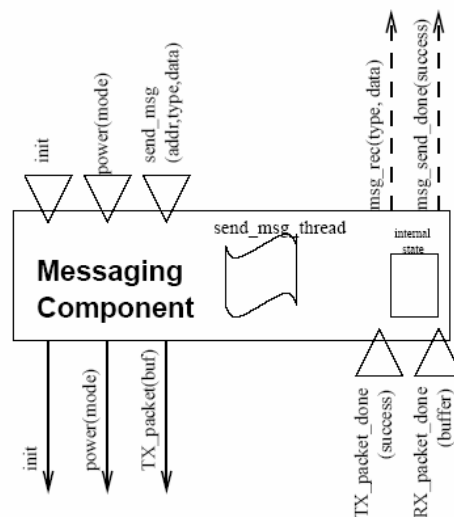


Figura 23: Representación gráfica de la componente “Active Messages” (AM)

```
module AMStandard
{
  provides {
    interface StdControl as Control;
    interface CommControl;

    // The interface are as parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    // How many packets were received in the past second
    command uint16_t activity();
  }

  uses {
    // signaled after every send completion for components which wish to
    // retry failed sends
    event result_t sendDone();

    interface StdControl as UARTControl;
    interface BareSendMsg as UARTSend;
    interface ReceiveMsg as UARTReceive;

    interface StdControl as RadioControl;
    interface BareSendMsg as RadioSend;
    interface ReceiveMsg as RadioReceive;
    interface Leds;
    //interface Timer as ActivityTimer;
  }
}
```

Figura 24: Archivo que describe la componente.

Implementación de un componente.

En TinyOS las componentes se unen en tiempo de compilación, considerando las especificaciones dadas en las componentes de configuración, es decir, se unen respetando las relaciones declaradas por las interfaces. Para facilitar la composición, cada interfaz está descrita en el inicio de cada archivo de componentes. Éste entrega una lista de los comandos que acepta y los eventos que manipula, como también el conjunto de eventos que señala y los comandos que usa. Podemos visualizar el modelo como una caja negra, donde solo se ven entradas y salidas. La completa descripción de estas interfaces superiores e inferiores, son usadas en el tiempo de compilación para generar automáticamente archivos de cabecera.

Todo esto es manejado por el compilador de lenguaje NesC. Un claro ejemplo en que el compilador debe verificar múltiples uniones, es que si un evento simple debe ser manejado por varias componentes, en tiempo de compilación el código será automáticamente generado para enviar el evento a tantos lugares como sea necesario.

Ejemplo de una aplicación completa.

En la figura 25 se muestra una vista simplificada de una aplicación implementada con TinyOS. Cada nodo representa una componente, y las flechas representan las uniones de las interfaces. Cada flecha es rotulada con el correspondiente nombre de la interfaz.

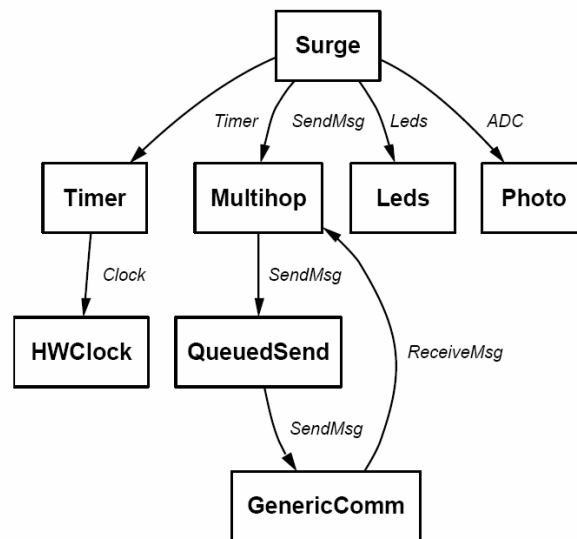


Figura 25: Aplicación surge

La aplicación se llama Surge, y tiene como función sensar periódicamente una variable, que en este caso es luminosidad, para luego ser enviada a través de la red inalámbrica. El objetivo de esta aplicación es entregar el valor medido a la base. Para eso cada nodo es capaz de identificar a un nodo padre, el cual le permitirá llegar a la estación base.

La aplicación solo debe incorporar las partes que necesita de TinyOS, como son el código de inicio de sistema (MAIN), el timer (Timer), un sensor (Photo), acceso a los leds (Leds), y enrutamiento de mensajes Multihop (Multihop). Además explícitamente se especifican las dependencias de ambiente en función de las interfaces. La aplicación requiere un timer (interfaz Timer), un sensor (interfaz ADC), leds (interfaz Leds) y finalmente comunicación (interfaz Send).

Modelo de Comunicación.

Una pregunta a resolver es ¿cómo se manejarán los mensajes en la red? Sin duda implementar socket en protocolo TCP/IP no es una buena idea, dado que se necesitaría excesiva memoria para almacenar los paquetes que van llegando además de los bloqueos que se generan mientras no se lea el mensaje. También es necesario evitar el excesivo tráfico sobre la red, por lo cual mensajes de recibo, secuencias de retransmisión, etc. son inaceptables.

La solución propuesta es trabajar con la metodología de “Active Messages”. Para su funcionamiento cada mensaje debe contener el nombre de un manejador de eventos. Para esto el que envía debe declarar un buffer en el frame, para así colocar el nombre del manejador, luego solicitar el envío y esperar que llegue la respuesta de realizado. El receptor entonces se encarga de invocar el correspondiente manejador de eventos.

De esta forma no se generan bloqueos o esperas en el receptor, el comportamiento es similar a que ocurriese un evento, y el almacenamiento es simple.

2.4 XubunTOS

2.4.1 Introducción

Sistema operativo estilo UNIX, que cuenta con el añadido de llevar incorporado e instalado el sistema operativo para redes de sensores inalámbricos Tinyos. Este sistema nace a causa de las dificultades que solía presentar la instalación y la actualización de Tinyos, tanto en sistemas operativos UNIX como Windows. Se trata de la unión de Xubuntu 7.04 (Feisty fawn) con la versión de Tinyos 2.x más estable, la 2.0.2. , además de la última versión de Tinyos 1.x.



Figura 26: Logotipo del sistema

Dicho sistema operativo viene presentado en formato live CD, con la opción de instalación, con lo que su uso se hace muy sencillo, tanto a los que no tengan la intención de almacenarlo en su PC como a aquellos que quieran hacerlo, ya que está diseñado para usuarios que poseen recursos limitados del sistema y buscan un entorno de escritorio altamente eficiente (Xfce).



Figura 27: Escritorio de XubunTOS

2.4.2 Características principales

Las principales características del XubunTOS son las siguientes:

- posesión de programas de desarrollo de código que reconocen el lenguaje NesC,
- posibilidad de actualización tanto del sistema operativo Xubuntu como de Tinyos,
- definición de variables de consola típicas de Tinyos, como \$TOSROOT, que hacen que la navegación por los distintos directorios sea mas fácil,
- posibilidad de establecer dispositivo de arranque USB con la ayuda del live CD

2.4.3 Limitaciones

Las principales limitaciones de XubunTOS son las siguientes:

- aunque el sistema viene con la última versión de Tinyos, si este quisiera ser actualizado, se cambiarían de forma automática los permisos del sistema, por lo que habría que volver a establecerlos ejecutando `tos-fix-permissions`
- el uso del live CD puede dar algunos problemas, eso si, ajenos al software, sobre todo en cuanto a rapidez.

2.5 TelosB

Los sensores que se utilizarán a lo largo de este proyecto serán los Motes TelosB TPR2420 de Crossbow. Las características principales de estos sensores son las siguientes:



Figura 28: TelosB

- Protocolo IEEE 802.15.4.
- Tasa de transferencia de 250 kbps.
- Microcontrolador TI MSP430 con 10 KB de RAM.
- Antena integrada en la placa.
- Conexión y comunicación vía USB.
- Sensores de luz, humedad y temperatura integrados.
- Sistema abierto Tinyos 1.1.11 o superior.

La plataforma fue desarrollada y publicada a la comunidad de investigación por la Universidad de Berkeley. Esta plataforma desarrolla bajo consumo lo que permite una larga vida de la batería, ya que el sensor se encuentra en estado “Sleep” hasta que recibe una orden y despierta. El TPR2420 es compatible con el sistema operativo de distribución abierta TinyOS, utiliza dos pilas AA, si el sensor se conecta en el puerto USB del ordenador para ser programado y para comunicarse con él, el sensor coge la energía del PC, por lo que no necesita pilas cuando está conectado al ordenador.

El telosB tiene unos conectores de expansión y unos jumpers en la placa que pueden ser configurados para controlar sensores analógicos, periféricos digitales y displays LCD.

A continuación se muestra la tabla de especificaciones del dispositivo inalámbrico:

Specifications	TPR2420CA	Remarks
Module		
Processor Performance	16-bit RISC	
Program Flash Memory	48K bytes	
Measurement Serial Flash	1024K bytes	
RAM	10K bytes	
Configuration EEPROM	16K bytes	
Serial Communications	UART	0-3V transmission levels
Analog to Digital Converter	12 bit ADC	8 channels, 0-3V input
Digital to Analog Converter	12 bit DAC	2 ports
Other Interfaces	Digital I/O,I2C,SPI	
Current Draw	1.8 mA	Active mode
	5.1 μ A	Sleep mode
RF Transceiver		
Frequency band ¹	2400 MHz to 2483.5 MHz	ISM band
Transmit (TX) data rate	250 kbps	
RF power	-24 dBm to 0 dBm	
Receive Sensitivity	-90 dBm (min), -94 dBm (typ)	
Adjacent channel rejection	47 dB	+ 5 MHz channel spacing
	38 dB	- 5 MHz channel spacing
Outdoor Range	75 m to 100 m	Inverted-F antenna
Indoor Range	20 m to 30 m	Inverted-F antenna
Current Draw	23 mA	Receive mode
	21 μ A	Idle mode
	1 μ A	Sleep mode
Sensors		
Visible Light Sensor Range	320 nm to 730 nm	Hamamatsu S1087
Visible to IR Sensor Range	320 nm to 1100nm	Hamamatsu S1087-01
Humidity Sensor Range	0-100% RH	Sensirion SHT11
Resolution	0.03% RH	
Accuracy	\pm 3.5% RH	Absolute RH
Temperature Sensor Range	-40°C to 123.8°C	Sensirion SHT11
Resolution	0.01°C	
Accuracy	\pm 0.5°C	@25°C
Electromechanical		
Battery	2X AA batteries	Attached pack
User Interface	USB	v1.1 or higher
Size (in)	2.55 x 1.24 x 0.24	Excluding battery pack
(mm)	65 x 31 x 6	Excluding battery pack
Weight (oz)	0.8	Excluding batteries
(grams)	23	Excluding batteries

Capítulo 3

Desarrollo de la aplicación

El propósito de este Proyecto Final de Carrera es el desarrollo de un “Modelo de Funcionamiento Colaborativo” para Redes de Sensores Inalámbricas (WSN). En líneas generales, dicha aplicación permite controlar el tráfico en una WSN, teniendo en cuenta la potencia obtenida al analizar los mensajes que circulan por dicha red, el origen del mensaje y el destino final. Para poder crear la aplicación se ha usado NesC, un lenguaje de programación orientado a eventos para el desarrollo de aplicaciones para sensores inalámbricos. A continuación se describe el proceso de desarrollo de la aplicación.

El primer paso fue la instalación de la aplicación Cygwin para el modelo de sensores TelosB en el sistema operativo Windows. Como se ha explicado anteriormente, Cygwin permite tener en un entorno Windows una consola similar a la de sistemas operativos UNIX para poder navegar por los directorios que conforman el sistema operativo Tinyos (tanto la versión 1.x como la 2.x), los cuales han sido presentados anteriormente. La instalación fue mediante la ejecución de un archivo “setup”, que englobaba las dos aplicaciones anteriores. La instalación no ofrece ningún problema y una vez realizada se pasó a la primera toma de contacto con las aplicaciones.

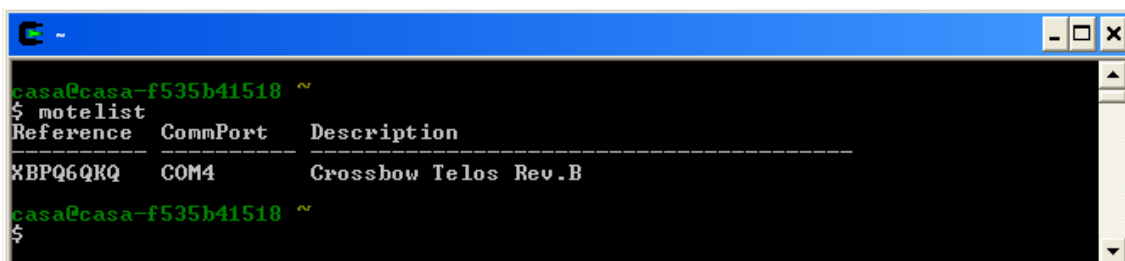
La primera toma de contacto fue la navegación por los directorios del sistema operativo Tinyos 1.x para ir accediendo a los distintos directorios del sistema y explorar la estructura del mismo. Se trata de un sistema distribuido en forma de árbol (nodo raíz “/”) similar a los sistemas UNIX, de ahí el uso necesario de Cygwin en sistemas operativos Windows. Una vez familiarizados con el entorno de trabajo, entró en juego el uso de los motes, algo totalmente nuevo. Se comenzó con la instalación de los drivers necesarios para el reconocimiento de los motes y que estos pudieran interactuar con el PC, tanto en el caso de transferencia de datos de la red inalámbrica al PC, sobre todo a la hora de visualizarlos mediante aplicaciones Java; como en el caso de hacerlo en sentido contrario, es decir, desde el PC hacia los dispositivos inalámbricos, sobre todo con el propósito de programar dichos dispositivos. Una vez instalados los drivers y obtenidos los dispositivos inalámbricos, se pasó a la conexión de estos con el PC mediante los puertos USB.

Con los dispositivos conectados al PC y ayudado por el manual de instalación para TelosB se logró programar los motes con uno de los programas de ejemplo que ofrece el sistema operativo Tinyos 1.x. Para poder programar los dispositivos se ha de arrancar el programa Cygwin para poder simular un shell estilo UNIX, navegar por los directorios que ofrece el sistema operativo mediante los comandos apropiados hasta

llegar a aquel que contenga la aplicación a instalar. Una vez allí se ejecuta una instrucción con la siguiente estructura:

```
>make telosb install,<id_disp> bsl,<num_puerto - 1>
```

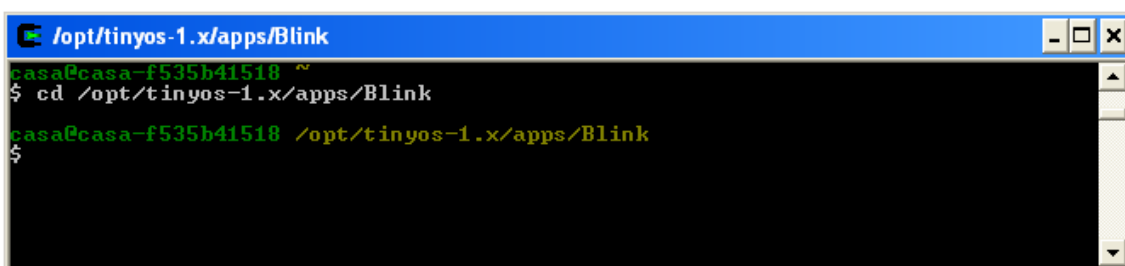
donde <id_disp> es el identificador de la red del dispositivo, y <num_puerto> es el número de puerto del PC al que se encuentra conectado. La información del número de puerto se puede obtener ejecutando el comando >motelist el cual mostrará la información de la siguiente forma:



```
casa@casa-f535b41518 ~
$ motelist
Reference  CommPort  Description
-----
XBPO6QKQ  COM4      Crossbow Telos Rev.B
casa@casa-f535b41518 ~
$
```

Figura 29: comando motelist

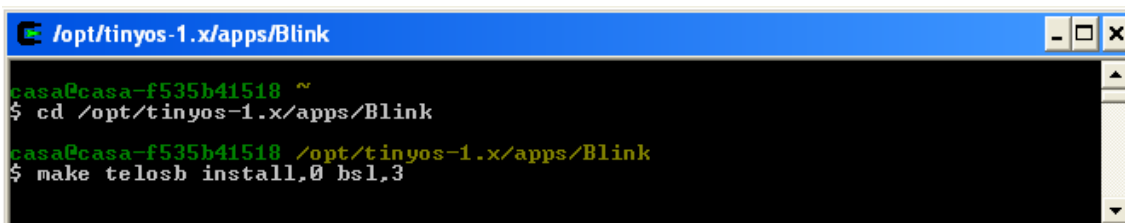
Como ejemplo se programa un dispositivo con la aplicación Blink (cuya funcionalidad se explica en el siguiente apartado), alojada en el directorio /opt/tinyos1.x/apps/Blink. Para acceder a dicho directorio se ejecuta el comando >cd y la ruta:



```
/opt/tinyos-1.x/apps/Blink
casa@casa-f535b41518 ~
$ cd /opt/tinyos-1.x/apps/Blink
casa@casa-f535b41518 /opt/tinyos-1.x/apps/Blink
$
```

Figura 30 : acceso al directorio

Una vez en el directorio seleccionado se ejecuta el comando con la similar al expuesto en líneas anteriores:



```
/opt/tinyos-1.x/apps/Blink
casa@casa-f535b41518 ~
$ cd /opt/tinyos-1.x/apps/Blink
casa@casa-f535b41518 /opt/tinyos-1.x/apps/Blink
$ make telosb install,0 bsl,3
```

Figura 31: comando make

Cuando se ejecuta el comando, dos de los 5 leds del mote parpadean indicando que está siendo programado, el de recepción y el de transmisión de datos del puerto USB. Cabe apuntar que el comando ejecutado anteriormente conlleva varias acciones, en un primer paso, se compila el código de la aplicación que se va a instalar, a continuación pasa a realizar una comprobación del estado del mote, hace un borrado total de la memoria y por último lo programa.

Una vez controlada la programación de los sensores, dentro del estudio de las aplicaciones que se ofrece en el sistema operativo, se estudia la estructura de una aplicación en un lenguaje de programación como es NesC y la forma de adquirir ciertas funcionalidades para los dispositivos inalámbricos. El estudio de las aplicaciones no se centró en un tipo específico, de forma que se comenta de forma genérica. A continuación se describe las aplicaciones más importantes.

-Blink: aplicación básica útil para el manejo de los temporizadores, ya que su única función es la de encender un led cada vez que se lance un temporizador.

-RfmToLeds: aplicación cuya función es la recepción de mensajes de tipo IntMsg y mostrar por medio de los leds del mote los tres bits menores recibidos. En combinación con CntToLeds y CntToLedsAndRfm sirve como ejemplo de la comunicación entre dispositivos.

-CntToLeds: aplicación basada en un contador controlado por un temporizador de 4Hz que mostraba la secuencia mediante los leds del dispositivo inalámbrico.

-CntToLedsAndRfm: aplicación similar a CntToLeds pero a la misma vez que maneja el contador, envía el valor en un paquete AM IntMsg.

-SenseToLeds: aplicación que lee de uno de los sensores del dispositivo inalámbrico y representa las muestras en sus leds.

-SenseToRfm: aplicación similar a la anterior con la diferencia que envía las muestras tomadas por los sensores en un mensaje AM broadcast.

-GenericBase: aplicación que sirve de puente entre la red de sensores inalámbricos y el PC. Si esta aplicación recibe un mensaje desde el PC este mensaje es automáticamente reenviado hacia los sensores; si por el contrario el mensaje que recibe procede de la red de sensores, este lo envía hacia el PC. Para controlar su funcionamiento hace uso de los leds.

HERRAMIENTAS DE ANÁLISIS

Para el análisis de las redes de sensores inalámbricos, a parte de las aplicaciones detalladas anteriormente, se han creado una serie de programas en lenguaje Java, cuya función básica es la monitorización del estado de la red: envío y recepción de paquetes, latencias y tiempos de transmisión, contenido de los paquetes, etc... Estas aplicaciones en combinación con las anteriores pueden servir de gran ayuda, aunque en ocasiones su funcionamiento no suele ser del todo correcto y su puesta en marcha algo compleja.

En primer lugar se comprobó el funcionamiento de TOSSIM, un entorno de simulación que permitía examinar la ejecución de una aplicación NesC sin necesidad de que ésta se encuentre instalada en un dispositivo inalámbrico. Profundizando un poco más, TOSSIM compila el código en el propio PC y una vez compilado se ejecuta el archivo ejecutable creado. Para hacer uso de dicha aplicación hay que situarse en el directorio de la aplicación y ejecutar el comando `>make pc`, este comando compilará el programa y almacenará el archivo ejecutable para ser simulado desde el propio PC. Para comenzar la simulación, desde el mismo directorio, se ejecuta el comando `>build/pc/main.exe <num_disp>`; donde `num_disp` es el número de nodos que deseas simular. La simulación pondrá en funcionamiento la aplicación, mostrando los eventos que vayan ocurriendo: mensajes recibidos y enviados, contenido, estado de los leds, etc. Además en TOSSIM puedes seleccionar las opciones que pueden ser mostradas a través de la variable del sistema `DBG`, activándola mediante la sentencia `>export DBG=<opciones>`, donde `opciones` es aquello que va a ser mostrado. A modo de ejemplo se muestra una captura de la simulación con TOSSIM de la aplicación `CntToLedsAndRfm`, comentada anteriormente, para un solo dispositivo y con las opciones `leds` y `AM` (estado de los leds y mensajes activos).

```
0: LEDS: Yellow off.
0: LEDS: Green off.
0: LEDS: Red off.
0: Sending message: ffff, 4
  ff ff 04 7d 08 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3b f3 00 00 01 00 00 00
0: LEDS: Yellow off.
0: LEDS: Green off.
0: LEDS: Red on.
0: Sending message: ffff, 4
  ff ff 04 7d 08 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ac e6 00 00 01 00 00 00
```

Junto con TOSSIM aparece otra herramienta de simulación más desarrollada y con una interfaz gráfica que permite visualizar el envío y recepción de mensajes, el estado de la red, etc además de poder introducir nuevos mensajes. Esta herramienta se denomina TinyViz. Para poder ejecutar TinyViz hay que situarse en el directorio donde se aloja la aplicación, situada dentro del directorio general `/opt/tinyos-1.x/` en la ruta `/tools/java/net/tinyos/sim`, seguidamente compilar con `>make`, establecer la variable de entorno `DBG` con el modo reservado `usr1` y por último ejecutar con el comando `>tinyviz -run build/pc/main.exe num_disp` donde `num_disp` es el número de dispositivos para el que desea realizar la simulación. A continuación se muestra una captura de la simulación de la aplicación `TestTinyViz`, proporcionada para el sistema para 30 dispositivos inalámbricos, la cual manda un paquete a un vecino de forma aleatoria.

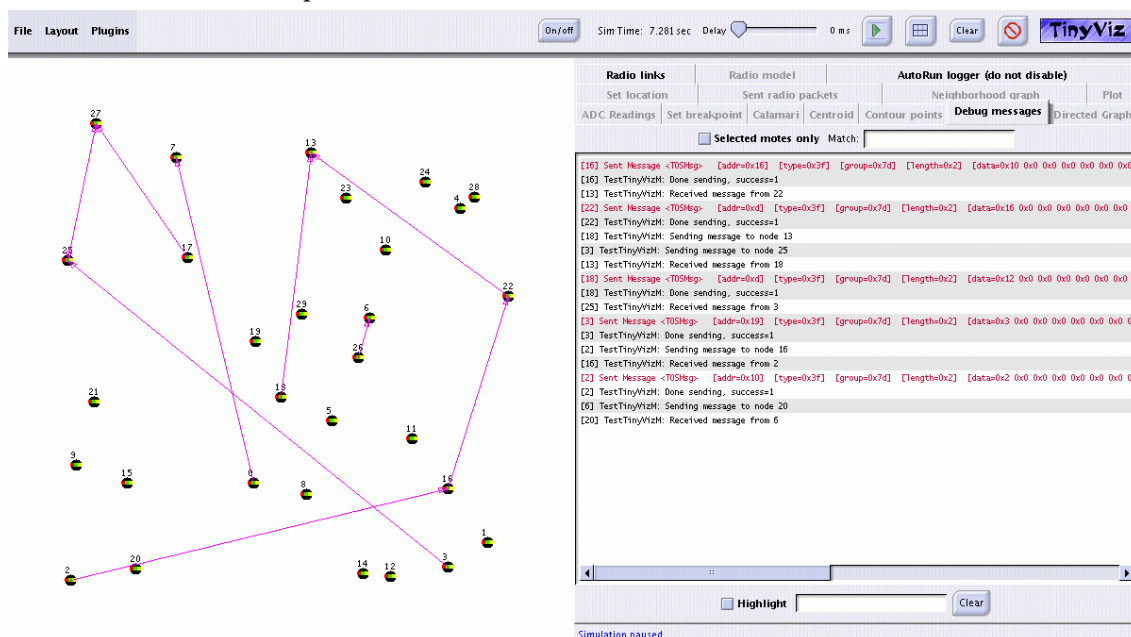


Figura 32: TinyViz

Como se observa en la zona izquierda se muestra la disposición de la red donde se observan los dispositivos con el estado de sus leds y los mensajes enviados y recibidos. En la parte derecha se tiene varias pestañas. Cada una de ellas es una opción que nos ofrece el programa para visualizar distintas características de la red. Destacan opciones como la visualización de los mensajes, como “Sent radio packets” o “Debug messages”, situación de los dispositivos inalámbricos como “Set location” o la opción “Radio links” que muestra el estado de los envíos y reopciones vía radio.

Como inciso, indicar que durante el estudio de esta aplicación surgieron varios problemas. El primero se relevó a la hora de compilar la aplicación donde aparecieron excepciones, debido a un problema del sistema con las variables de entorno que fue solucionado sin mayor problema gracias al foro de la web de Tinyos. Una vez se consiguió arrancar el programa, se percibe que no era capaz de mostrarnos ningún mensaje de los que debía mandar de forma aleatoria, así que se pensó que era problema de la aplicación por lo que se probó otra aplicación observando que tampoco producía resultado alguno. Viendo que la aplicación seguía sin funcionar se desarrolló una búsqueda intensiva en la red, pero sin solución válida, simplemente se comentaba que TinyViz solía dar muchos problemas y no era muy fiable. Finalmente tras continuar con la investigación y varias operaciones de configuración se llegó a que la aplicación mostraba los mensajes enviados vía radio.

De forma paralela al estudio de TinyViz se comenzó con otra herramienta programada en Java llamada *Listen*. Esta herramienta se encarga de escuchar en un puerto serie determinado para monitorizar todo lo que le llegue. Para ejecutarla se tiene que programar un dispositivo como base receptora de paquetes y conectarlo a uno de los puertos del PC. Una vez esté el dispositivo conectado al PC se accede al directorio */tools/java* dentro del directorio general */opt/tinyos-1.x/*, se compila con el comando *>make* y a continuación se pasa a la asignación el puerto por el que el programa debe escuchar. Para esto existen dos formas distintas: la primera estableciendo la variable de

entorno MOTECOM, o en la misma línea del comando de ejecución con el parámetro *comm*. A continuación se ejemplifican cada una de las opciones:

```
export MOTECOM=serial@COMx:(tasa del dispositivo)
```

```
export MOTECOM=serial@COM2:telos
```

```
java net.tinyos.tools.Listen -comm serial@COM2:telos
```

Una vez establecida la variable, como en el ejemplo 1, se ejecuta el programa con el comando `>java net.tinyos.tools.Listen`. En el caso de usar el ejemplo 2 se realizan las dos acciones en una, el establecimiento del puerto serie y la ejecución del programa. Ejecutado el programa muestra el contenido de los mensajes que se redireccionan por el puerto serie. A continuación se muestran unas capturas de la herramienta Listen sobre la aplicación Oscilloscope proporcionada por el sistema.

```
% java net.tinyos.tools.Listen
serial@COM1:telos: resynchronising
7e 00 0a 7d 1a 01 00 0a 00 01 00 46 03 8e 03 96 03 96 03 96 03 97 03
97 03 97 03 97 03 97 03
7e 00 0a 7d 1a 01 00 14 00 01 00 96 03 97 03 97 03 98 03 97 03 96 03
97 03 96 03 96 03 96 03
7e 00 0a 7d 1a 01 00 1e 00 01 00 98 03 98 03 96 03 97 03 97 03 98 03
96 03 97 03 97 03 97 03
```

Conociendo la estructura de los mensajes enviados se puede ver analizar el contenido de estos. Con los 5 primeros bytes se extrae la información de la cabecera del mensaje y a continuación aparece el campo payload donde se incluyen los datos. Posteriormente se realizaron otras pruebas con otras aplicaciones sin resultado satisfactorio en varios casos por lo que no sirvió de gran ayuda.

Para finalizar con el tema de las aplicaciones Java cabe comentar SerialForwarder. Esta aplicación no se estudió en profundidad ya que su uso no entraba dentro de lo que se preveía iba a ser el PFC.

PRIMEROS ASPECTOS DE LA APLICACIÓN

Tras finalizar con el estudio general del sistema operativo Tinyos 1.x se concreta el funcionamiento de la aplicación que forma parte de este PFC. La idea básica es implementar una aplicación colaborativa para distribuir paquetes a través de una WSN, en este caso los dispositivos son Motes Telosb. La aplicación está basada en el intercambio de paquetes entre dos o más dispositivos. En este intercambio se envían paquetes con un destino final a través de la red, cuando un nodo recibe un mensaje comprueba el RSSI del mensaje y lo actualiza para dicho enlace, una vez hecho esto comprueba el campo destino final del mensaje, para ver si es para él, si es para él se lo queda, sino comprueba en su tabla (una *pseudotabla* de enrutamiento) si conoce al destino final, en este caso se lo envía directamente a él, en caso contrario, lo envía por el enlace que mayor RSSI tenga.

El primer problema que se presentó vino a la hora de calcular la potencia de los mensajes. Tras diversas consideraciones y alternativas realizadas durante bastante tiempo probando varias opciones con Tinyos 1.x, no se obtuvo ninguna solución y todo

lo que se presentaban eran opciones ambiguas que no se correspondían con las expectativas presentadas, hasta que se encontró un documento que trataba sobre el cálculo de potencias de transmisión y contenía una pequeña aplicación para calcularla pero estaba pensado sobre el API de funciones de TinyOS 2.x. Llegados a este punto se barajaron dos posibilidades: transformar el documento para que fuera válido para TinyOS 1.x , o actualizar el sistema operativo y poder analizarla y estudiar su comportamiento. Finalmente se tomó la decisión de actualizar el sistema operativo y aprovechar las novedades que ofrece Tinyos 2.x. Inmediatamente se obtuvo la actualización descargándola a través de la web del sistema siguiendo las instrucciones paso por paso. Comenzaron a aparecer errores en la instalación de los distintos paquetes de los que está compuesta. Buscando nuevas alternativas apareció un novedoso sistema operativo que mezclaba una de las últimas versiones del sistema operativo Xubuntu tipo UNIX y la versión 2.0.2 del sistema operativo para redes de sensores inalámbricas Tinyos llamado XubunTOS, lo cual era precisamente lo que se estaba planteado ya que la mayor preocupación en ese momento era encontrar un sistema Tinyos 2.x, fuera cual fuera el sistema operativo que lo soportara. Venía presentado en formato live CD con opción a ser instalado. La instalación se realiza sin mayores problemas de compatibilidad.

Con el nuevo sistema operativo instalado se pasó a estudiar la distribución de los distintos directorios del sistema la cual no resultó muy diferente a la de Tinyos 1.x e inmediatamente después se probó la aplicación para ver si realmente era tan necesaria como se esperaba. En este momento se vio el primer cambio entre las dos versiones de Tinyos, aunque la verdadera diferencia venía por el sistema operativo ya que a la hora de programar los sensores, en el lugar de poner el numero de puerto donde está conectado el mote, se pone la ruta del puerto USB correspondiente, como se comprobará más adelante.

En la versión de TinyOS 2.x aparece un API o interfaz para la captura del RSSI que estaba formada por dos pequeños programas. El primero de ellos llamado “RssiBase” realizaba la función de base, es decir, recibe mensajes, calcula la potencia, y ayudado por una aplicación Java muestra dicha potencia por la pantalla del PC. El segundo programa, llamado “SendingMote”, se encarga de mandar mensajes de forma periódica controlada por un temporizador, para que sean recibidos por el mote y se le calcule la potencia.

Para comprobar la API se tiene que instalar cada uno de los programas anteriores en dos dispositivos distintos. A continuación se muestra el proceso paso a paso.

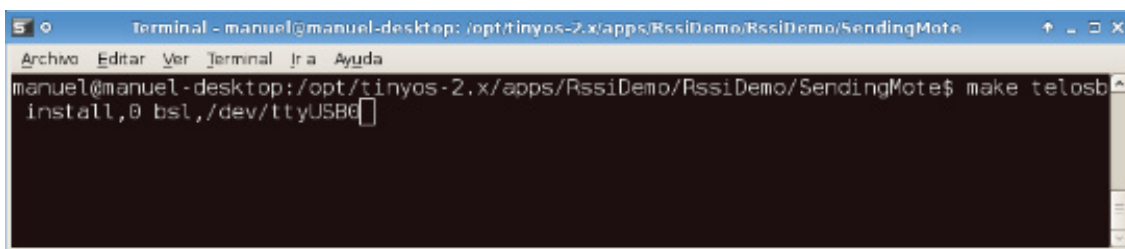


Figura 33: instalación SendingMote

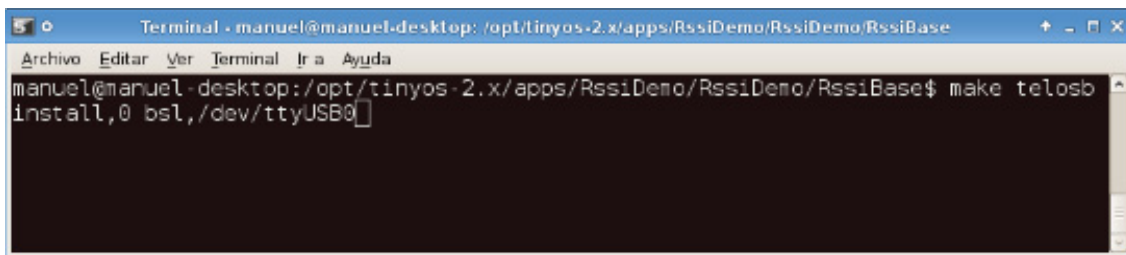


Figura 34: Instalación RssiBase

Una vez programados se vuelve al directorio principal de la aplicación y si compila el programa Java para la monitorización de la potencia. Una vez compilado se ejecuta con el siguiente comando, indicando el puerto al que se encuentra conectado con el parámetro *-comm*:

```
>make  
>java Rssi -comm serial@/dev/ttyUSB0:telos
```

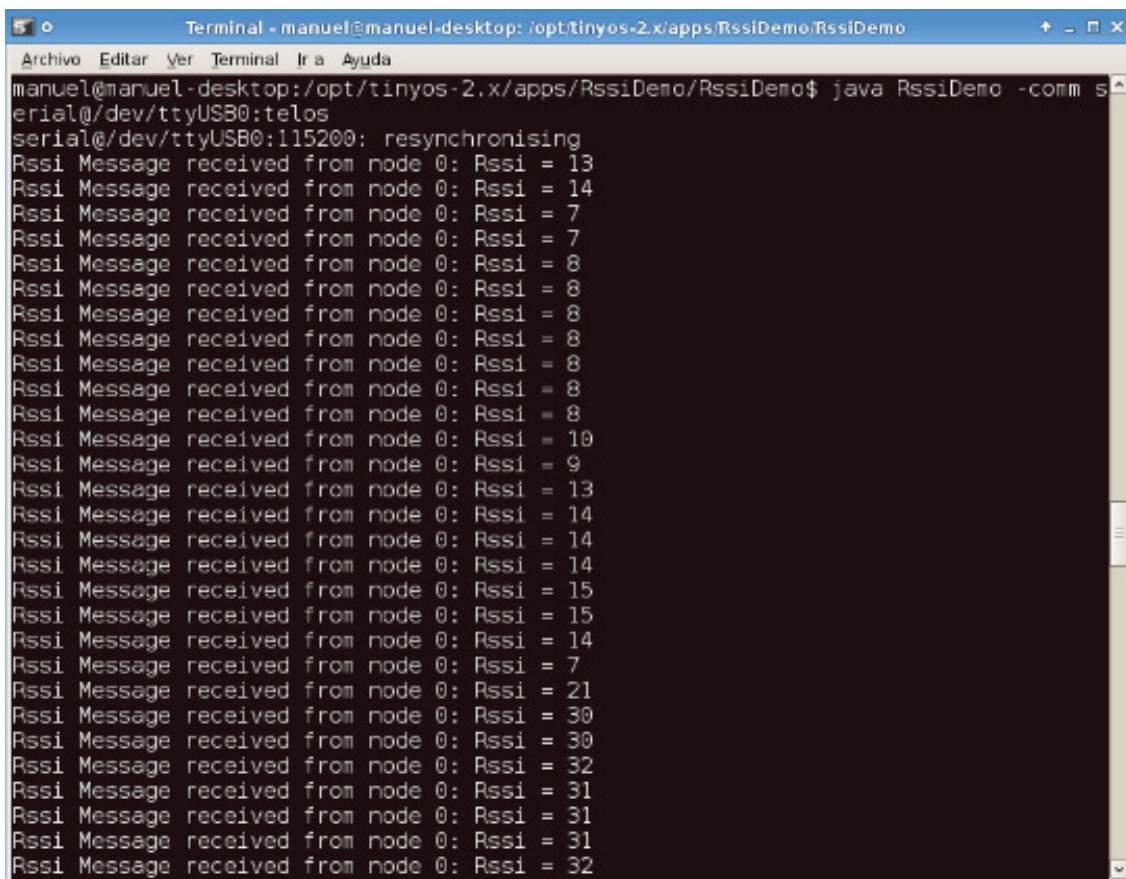


Figura 35: ejecución Rssi

Como se observa se muestran los valores de la potencia de cada uno de los mensajes recibidos junto con el id del mote.

De esta forma y observando el resultado de la ejecución de la aplicación, se sacaron varias conclusiones con respecto a la programación final de nuestra aplicación, una de ellas era el uso de la función que permite calcular la potencia de los mensajes recibidos, otra conclusión fue el uso de la estructura de dicha aplicación, tanto en el Coordinador como del nodo encargado del envío. También se observó cierta diferencia a la hora de desarrollar código, se eliminan ciertos eventos y funciones que en el sistema anterior eran de uso obligatorio, etc... pero es algo, en este caso, sin importancia ya que no se profundizó en el desarrollo de código en la versión 1.x, así que se partió de cero en este aspecto.

Con todos estos aspectos y tras varias decisiones tomadas sobre el papel, se comenzó con la creación del código. Los primeros pasos se dieron estudiando la estructura de las aplicaciones (explicadas anteriormente), ya que añadía nuevos elementos que no contienen lenguajes de programación parecidos como puede ser C. Cualquier aplicación NesC está compuesta por dos archivos, el archivo de configuración donde se realiza el “wiring”, para enlazar los componentes con las interfaces y poder ser usados en el otro archivo que es el module, donde se implementa realmente la aplicación.

AplicaciónColaborativaC.nc

Una vez preparados para la programación de la aplicación se empezó a desarrollar el código necesario para ello, se empezó teniendo en cuenta el propósito de este proyecto, y se analizó cual era el fin de este proyecto y cuál era la manera de realizarlo.

Esta aplicación debería recoger cualquier mensaje que circulara dentro de su alcance, capturarlo y analizarlo, para esto los mensajes se envían a la dirección broadcast (AM_BROADCAST_ADDR), el destino final del mensaje creando un campo en los datos, llamado Dest_final, del que se hablará más adelante y se recibe el mensaje utilizando el evento receive, el cual es proporcionado por la interfaz Receive:

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

En este caso msg es el mensaje recibido, payload un puntero al inicio de los datos dentro del mensaje, y len es el tamaño de la región de datos. Este evento salta cuando detecta un mensaje, el mensaje es capturado y puede ser tratado, nos interesaba conocer de este mensaje el destino final del mismo así como la potencia RSSI.

Para conocer el destino final se creó una nueva estructura de datos del tipo:

```
typedef nx_struct RssiMsg{  
    nx_int16_t Dest_final;  
} RssiMsg;
```

Donde Dest_final contendría cual sería la Id del nodo al cual estaba destinado el mensaje, para posteriormente comprobar si el mensaje es para mí o para otro nodo.

Cada vez que un mensaje es recibido se mira la Id de origen y se guarda en una tabla que será un array con la direcciones guardadas según la Id, es decir la Id número 3 se guardará en la posición 3 del array, así se conseguirá que cuando quiera saber si un nodo esta dentro de nuestro alcance para enviarle el mensaje directamente, simplemente comprobando si el elemento deseado está en la posición del array deseada lo se sabrá.

Para conocer el destino final se coge el mensaje msg y se saca de los datos y de los datos el campo destino final que se ha creado:

```
RssiMsg *destinofinal = (RssiMsg*) payload;  
Destino = destinofinal->Dest_final ;
```

Así, en Destino, se encontrará el contenido el campo Dest_final del mensaje, el Dest_final debe ser añadido anteriormente por el nodo que origina el mensaje, cada vez que se recibe un mensaje se comprobará el origen con la función, la cual la proporciona la interfaz AMPacket:

```
NOrigen = call AMPacket.source(msg);
```

Con esta llamada se tendrá en NOrigen la Id del nodo que envió el mensaje, luego se guarda en un array Tabla[], este dato, para posteriormente utilizarlo.

Cuando se recibe un mensaje, también se lee el valor de RSSI, el cual será útil cuando el destino del mensaje no sea conocido por el nodo, ya que el mensaje recibido en el caso de que no sea para este mote se enviará al nodo cuyo enlace tenga mayor RSSI, para que haya más posibilidades de éxito en el envío del mensaje. Para conocer el RSSI del mensaje se hace la siguiente llamada, esta función getRssi() es proporcionada por la interfaz CC2420Packet:

```
PotRssi = call CC2420Packet.getRssi(msg);
```

En PotRssi se tendrá la potencia de enlace por el cual ha sido enviado el mensaje msg.

La variable msg es el mensaje al que se le calcula la potencia. Llegados a este punto se va a explicar el método para obtener potencia de un mensaje recibido. Para llevarlo a cabo es necesario que el mensaje sea de tipo message_t ya que la obtención de la potencia mediante el uso de la función getRssi se basa en el estudio de los distintos campos de dicho tipo de mensaje. La función se encuentra implementada en el componente CC2420Packet. La llamada a dicha función provoca una nueva llamada, pero en este caso a la función getMetadata para poder acceder a los valores de ese campo, y a su vez especifica el campo al que quiere acceder, el rssi

```
getMetadata(message_t msg)->rssi
```

```
typedef nx_struct message_t {  
    nx_uint8_t header[sizeof(message_header_t)];  
    nx_uint8_t data[TOSH_DATA_LENGTH];  
    nx_uint8_t footer[sizeof(message_footer_t)];  
    nx_uint8_t metadata[sizeof(message_metadata_t)];  
} message_t;
```

```
typedef nx_struct cc2420_metadata_t {
```

```
nx_uint8_t tx_power;  
nx_uint8_t rssi;  
nx_uint8_t lqi;  
nx_bool crc;  
nx_bool ack;  
nx_uint16_t time;  
} cc2420_metadata_t;
```

Cada vez que se recibe un mensaje y se conoce su RSSI, este es actualizado, si ya se conoce, sino se guarda en un array en la posición de la Id del nodo, es decir, para el nodo 3, su RSSI, se guardará en la posición 3 del array RSSI[].

```
for(i=0;i<NumMotes;i++){ //actualizar RSSI  
    if(Origen[i]==NOrigen){  
        RSSI[i]=PotRssi;  
    }  
    else{  
        RSSI[pos]=PotRssi;  
        Origen[pos]=NOrigen;  
    }  
}
```

La siguiente tarea que se realizó fue ordenar el array de RSSI, para ello lo primero que se hace es una copia de dicho array para que este no sea modificado y poder siempre tenerlo en el orden en el que lo guardo, la copia se realiza:

```
for(i=0; i<NumMotes;i++){  
    CopiaOrigen[i]=RSSI[i];  
}
```

Una vez hecho esto, con la idea del algoritmo de la burbuja ordeno este array para tener en la posición 0 del array el valor de potencia mayor:

```
for (i=1; i<NumMotes; i++){  
    for (j=0 ; j<NumMotes - 1; j++){  
        if (CopiaOrigen[j] < CopiaOrigen[j+1]){  
            temp = CopiaOrigen[j];  
            CopiaOrigen[j] = CopiaOrigen[j+1];  
            CopiaOrigen[j+1] = temp;  
        }  
    }  
}
```

El fin de realizar esta tarea será, que cuando se reciba un mensaje se comprobará su destino final y se verá que no es para nosotros y que no se tiene en nuestra tabla como un nodo dentro de nuestro alcance, el mensaje será enviado al nodo cuyo RSSI, sea el que tenga mayor potencia, es decir, se encuentre en CopiaOrigen[0].

Llegados a este punto lo que hay que comprobar es si los datos son para mí, para ello, si la variable Destino es igual a mi Id, no se hace nada, ya que el mensaje ha llegado a su destino final.

Una vez que se determina que este nodo no es el receptor del mensaje, sino que es para otro destino éste se busca en la *pseudotabla* de enrutamiento, en caso de encuentro positivo, se conforma un nuevo mensaje cuyos datos llevará el destino obtenido como se comentó anteriormente y se envía directamente hacia ese destino, con la función:

```
command error_t send(am_addr_t addr, message_t *msg, uint8_t len)
```

Donde *addr* es la dirección de envío, *msg* el mensaje a enviar, y *len* el tamaño de la parte de datos del mensaje. El uso de esta función obliga a la implementación del evento *sendDone* que pertenece a la interfaz *AMsend* que es proporcionado por el componente *AMSenderC*. Tiene el siguiente formato:

```
event void sendDone(message_t *msg, error_t error)
```

la porción de código donde se realiza esto, es:

```
RssiMsg *Mensaje;  
Mensaje=(RssiMsg*) call Packet.getPayload(&Aux,NULL);  
Mensaje->Dest_final= Destino;  
if(Tabla[Destino]=Destino){//el destino del paquete está en mi tabla  
    call sendAux1.send(Destino,&Aux,sizeof(RssiMsg));  
}
```

En el caso de que no se conozca el destino, se buscará el enlace con mayor potencia y el mensaje será enviado por este, así cuando el nodo de este enlace lo reciba, realizará el mismo proceso, es decir, comprobará si es para él, en caso negativo, comprobará si está en su tabla, en ese caso lo enviará directamente, en caso contrario lo enviará por el enlace que mayor RSSI tenga. La porción de código donde se realiza lo anterior, es:

```
for (i=0;i<NumMotes+1;i++){  
    if(CopiaOrigen[0]==RSSI[i]){  
        call sendAux2.send(Origen[i],&Aux,sizeof(RssiMsg));  
    }  
}
```

Lo primero que hace esta aplicación es poner un timer que cada dos segundos envíen un mensaje broadcast (*AM_BROADCAST_ADDR*) cuyos datos contiene la Id del destino final del mensaje, el cual es puesto a modo de ejemplo por el programador, este caso se pone como destino final el Id 3, para realizar esto se hace una llamada a la función *getPayload* la cual es proporcionada por la interfaz *Packet*. Esta función tiene el siguiente formato:

```
command void *getPayload(message_t *msg, uint8_t len)
```

La variable *msg* va a ser el mensaje que se va a crear y *len* el tamaño de la zona de datos del mensaje a crear.

```
RssiMsg *rssiMsg;  
rssiMsg=(RssiMsg*) call Packet.getPayload(&msgAux,NULL);  
rssiMsg->Dest_final= 3;
```

Y cuando salta el timer se mandará el mensaje a todos los nodos. El evento `fired()` es proporcionado por la interfaz `Timer<TMilli>`:

```
event void TimerMain.fired() {  
    call sendRssi.send(AM_BROADCAST_ADDR, &msgAux, sizeof(RssiMsg);  
}
```

El motivo de mandar el mensaje broadcast es que el evento `receive` sólo salta cuando un mensaje va destinado hacia el nodo que lo implementa o va destinado a todos (`AM_BROADCAST_ADDR`).

Otras interfaces utilizadas en el desarrollo de la aplicación son:

- `Boot`, que se encarga de notificar a los componentes que TinyOS se ha iniciado, implementa el evento `event void booted()` que es el primero que se lanza.
- `SplitControl`, Usado para cambiar de estado de encendido a apagado los componentes proporcionados, implementa los eventos `command error_t start()` que inicia componentes y subcomponentes y `command error_t stop()`.
- `Packet`, sirve para proporcionar acceso a los datos del mensaje.
- `Leds`, que sirve para indicar la manipulación de los leds del mote.

AplicaciónColaborativaAppC.nc

La otra parte que conforma la aplicación encargada del envío de mensajes es la parte de configuración. En ella se especifican los componentes usados para darle función a las comandos y eventos que van incluidos en las interfaces. En nuestro caso, se usará los componentes `TimerMilliC`, `BaseStationC`, `LedsC`, `CC2420PacketC`, `ActiveMessageC`, `MainC`, `CC2420ReceiveP` y `AMSenderC`, a los que se debe añadir otro con el mismo nombre que el archivo `module` de nuestra aplicación, es decir “`AplicacionColaborativaC`”.

```
components AplicacionColaborativaC as App;  
components new AMSenderC(7);  
components new TimerMilliC();  
components BaseStationC;  
components CC2420PacketC;  
components LedsC;  
components ActiveMessageC, MainC;  
components CC2420ReceiveP;
```

Cada uno de los componentes anteriores proporcionará una o varias interfaces para darle funcionamiento a sus distintas funciones y códigos, este es el denominado “`wiring`”, una de las partes mas problemáticas a la hora de programar en NesC.

```
App.Boot -> MainC.Boot;  
App.AMPacket -> AMSenderC.AMPacket;  
App.Receive-> CC2420ReceiveP;  
App.sendRssi -> AMSenderC.AMSend;  
App.sendAux1 -> AMSenderC.AMSend;  
App.sendAux2 -> AMSenderC.AMSend;
```

```
App.TimerMain->TimerMilliC;  
App.RadioControl -> ActiveMessageC;  
App.CC2420Packet -> CC2420PacketC;  
App.Packet -> AMSenderC;  
App.Leds -> LedsC;App.Leds -> LedsC;
```

Cada una de las líneas del “wiring” indica que cada una de las interfaces de nuestro componente usa la interfaz que proporciona el componente al que apunta. El nombre de este archivo va a ser el mismo que el module pero seguido de la palabra App, es decir “AplicacionColaborativaAppC” y siempre con extensión .nc.

Funcionamiento global de la aplicación

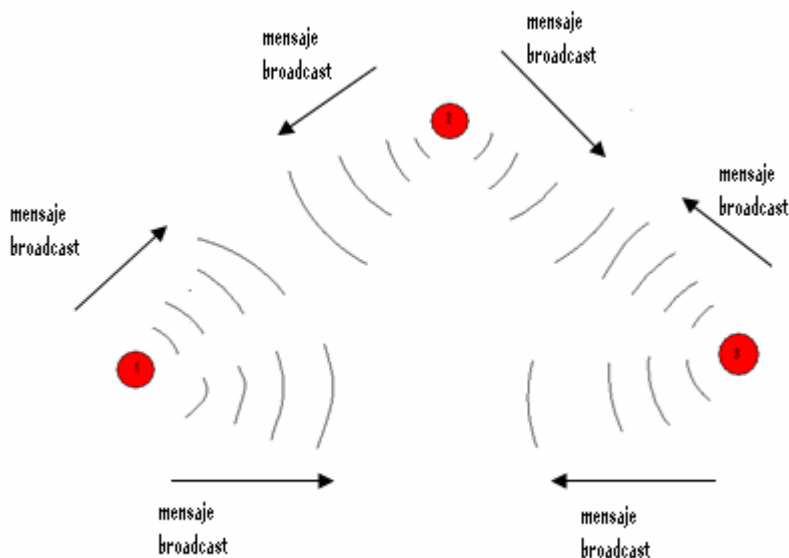


Figura 36: funcionamiento global de la aplicación

Esta aplicación lo primero que hace es mandar un mensaje broadcast para todos los nodos, insertando previamente en el campo de datos el destino final de la aplicación.

Cuando un nodo recibe el mensaje comprueba el campo de datos y extrae el contenido para saber el destino final del mensaje. Otra tarea que realiza cuando recibe un mensaje es guardar el origen del mensaje en una tabla para saber los nodos que tiene a su alcance y obtener el RSSI del mensaje para posteriormente estudiarlo si es necesario.

Una vez se tienen estos datos se dispone a conocer si el dato es para el nodo que recibe el mensaje haciendo una simple comprobación, en el caso de que sea el destinatario del mensaje ya no hace nada más. En caso contrario, comprueba si conoce al nodo al que va destinado el mensaje, si es así, se le envía el mensaje directamente a él, sino se conoce, busca el enlace que mayor potencia tenga y envía el mensaje por ese enlace, una vez este reciba el mensaje realizara la misma operación.

Compilación y puesta en marcha

Una vez terminada la codificación de las aplicaciones se pasa a la creación de las cabeceras. Las cabeceras contienen las distintas definiciones y enumeraciones de variables globales y de estructuras de datos que se usarán como mensajes. En nuestro caso van a contener una estructura para definir un tipo de mensajes, como muestra a continuación:

```
typedef nx_struct RssiMsg{  
    nx_int16_t Dest_final;  
} RssiMsg;
```

El archivo será llamado “RssiDemoMessages” y tendrá extensión .h. Las aplicaciones que hagan uso de cualquiera de las estructuras o variables definidas arriba deberán incluirla en sus archivos de la siguiente manera:

```
#include "RssiDemoMessages.h"
```

Para finalizar se compilarán ambas aplicaciones para depurar errores, se programarán en los dispositivos inalámbricos y se probará su funcionamiento. Los resultados de dicha prueba se observan en el apartado de resultados.

Capítulo 4

Evaluación del Funcionamiento de la Aplicación

A la hora de comprobar el funcionamiento de la aplicación, se va a usar un hardware que realiza el trabajo de sniffer para redes de sensores inalámbricas 802.15.4. El sniffer ha sido fabricado por *Chipcon*, está disponible para el sistema operativo Windows y se encarga de la monitorización de los paquetes que circulan por una red inalámbrica que se encuentre dentro de su alcance. Para hacer posible la monitorización de los mensajes se ha de instalar el software que acompaña al sniffer y configurarlo para el tipo de red utilizada.

Una vez conectado el sniffer al PC e inicializado el software, con los dispositivos inalámbricos programados y en funcionamiento, se inicia la captura.

En nuestro caso se capturará el intercambio de tramas entre los motes para ver si siguen el camino correcto teniendo en cuenta el destino final y el RSSI obtenido. La captura fue realizada en un habitáculo cerrado, por lo que ciertas mediciones pudieron estar enmascaradas por otros efectos radioeléctricos (aunque muy útiles para comprobar el adecuado funcionamiento de la aplicación), y consistió en varios escenarios.

El primero se pusieron todos los motes dentro del alcance del sniffer para ver las tramas capturadas, se colocaron 3 motes, los cuales se enviaban mensajes broadcast, pero con Destinos finales diferentes, así el mote con Id 1 los mandaba con Destino final 3, el Id 2 los mandaba con destino final 1, y el mote con Id 3 los mandaba con destino final 1, los resultados obtenidos fueron:

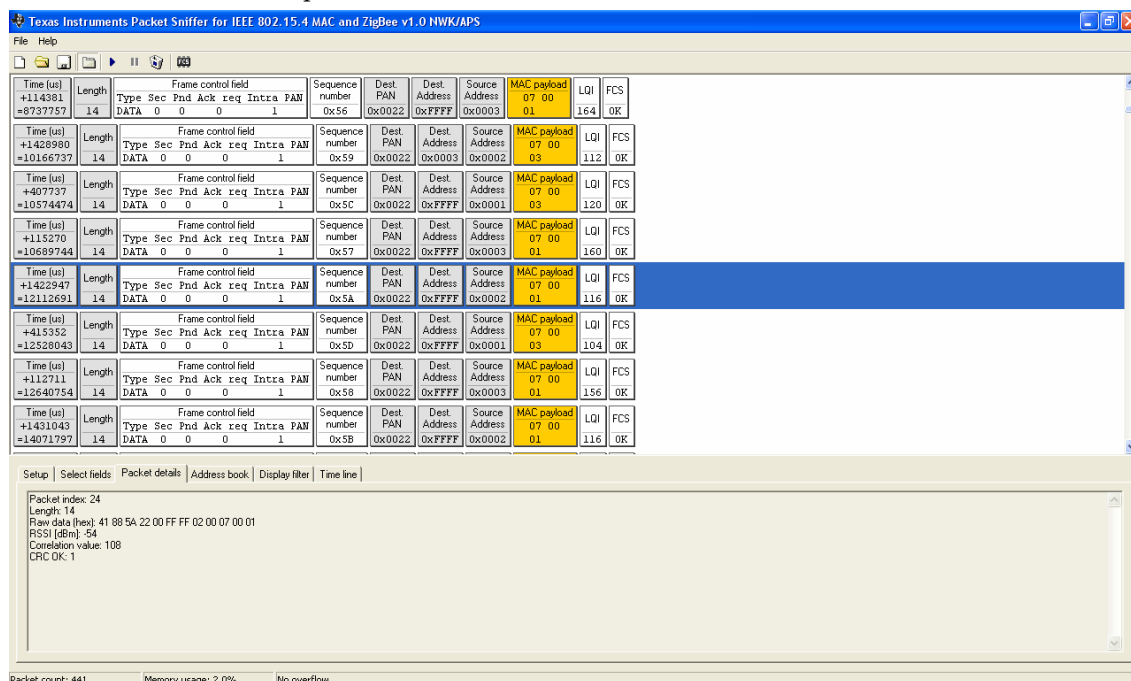


Figura 37: captura escenario 1

La imagen superior muestra un ejemplo de la captura de varias tramas por parte del sniffer. Como se ve, se muestran todos los campos de la trama y en la zona inferior, varias pestañas de opciones, donde seleccionar el tipo de campos a mostrar, lista de direcciones, frecuencia de actividad,...

Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload	LQI	FCS
+407737 +10574474	14	Type Sec Pnd Ack req Intra PAN DATA 0 0 0 1	0x5C	0x0022	0xFFFF	0x0001	07 00 03	120	OK

Figura38: Trama capturada

En la trama se observan los siguientes campos:

- **Time:** tiempo de transmisión
- **Length:** longitud de la trama en bytes
- **Frame Control Field:** campo de control de trama
- **Sequence Number:** número de secuencia de la trama (formato hexadecimal)
- **Dest. PAN:** red de destino
- **Dest. Address:** dirección de destino
- **Source Address:** dirección origen
- **MAC Payload:** contenido de los datos del paquete
- **LQI:** umbral de calidad del enlace
- **FCS:** control de errores (checksum) y verificación de datos

Para analizar el funcionamiento de la aplicación basta con ver la dirección origen y el MAC Payload y LQI, como se puede observar en la captura los nodos se intercambiaban mensajes correctamente, el segundo mensaje es uno que recibe el nodo con Id 2 y que tiene destino final 3 por lo que lo envía directamente hacia 3 ya que es un nodo conocido.

Un segundo escenario que se utilizó para comprobar el funcionamiento de nuestra aplicación colaborativa fue configurar los motes para que el mote con Id 1 le

mandara mensajes cada dos segundos a la red con destino final el mote 3. El mote con Id 3 queda configurado para que mande mensajes a la red cada dos segundos con destino final 1. Tanto el mote 1 como el 3 están configurados para que cuando envíen un mensaje parpadee el led rojo, y que cuando reciban un mensaje parpadee el led verde.

Una vez hecho esto, se ponen en funcionamiento y se observó que se intercambiaban mensajes sin problema ya que se encendía el led rojo y el verde, los cuales se configuraron para que se activaran cuando mandaban un mensaje y cuando lo recibiesen, asique se procedió a separarlos hasta una distancia mayor a su cobertura para que dejaran de verse el uno con el otro, cuando estaban separados el led rojo continuaba encendiéndose, ya que los motes seguían mandando mensajes a la red, pero estos se perdían ya que no eran capturados por nadie, al no verse los dispositivos, por lo que en los motes se dejaba de encender o se quedaba encendido permanentemente el led verde.

Para comprobar que el funcionamiento de la aplicación era el correcto, se situó el mote 2 con la aplicación instalada, pero cuyo propósito en esta prueba solo sería la de capturar los mensajes y reenviarlos hacia su destino final.

Cuando se encendió el mote 2 se observa que en el mote 1 y el mote 2 volvían a parpadear los leds verdes por lo que se pudo asegurar que la aplicación funcionaba correctamente, ya que el mote recibía los mensaje de 1 y de 3 y los encaminaba a 3 y 1, es decir, 1 y 3 se volvían a ver y a poder intercambiar mensajes gracias a la actuación de mote 2.

Capítulo 5

Conclusiones y líneas futuras

Se puede afirmar que todos los objetivos propuestos al comienzo del Proyecto Final de Carrera han sido cumplidos satisfactoriamente. Se ha conseguido definir el protocolo 802.15.4 y sus aplicaciones, conocer los dispositivos TelosB, conocer el lenguaje de programación NesC y desarrollar un modelo de funcionamiento colaborativo utilizando dispositivos TelosB.

La información incluida en esta memoria implica una guía de desarrollo de este tipo de aplicaciones, tanto en lo referente como al manejo y configuración de los dispositivos TelosB, como de la programación de aplicaciones en el lenguaje NesC.

Por último, la aplicación específica desarrollada satisface lo planteado, es decir, es capaz de efectuar un enrutamiento de la conexión cuando el origen y destino no poseen alcance directo, o simplemente es capaz de direccional por la mejor conexión en términos de RSSI. La modificación plantea se realiza en tiempo real de ejecución de la aplicación.

En cuanto a futuros trabajos relacionados con dichas redes de sensores inalámbricas, podemos mencionar el desarrollo de este proyecto para que los datos a enviar sean capturados por los sensores que tienen los dispositivos tales como luminosidad, humedad y temperatura. Es decir, se plantea integrar la aplicación desarrollada en un entorno real, en el que los datos a transmitir tenga interés práctico. La inclusión de esta aplicación permitiría, el incremento del tamaño de las redes de sensores así como optimizar su funcionamiento. El conocimiento de estos valores y el envío de ellos a través de la red puede tener gran utilidad para su uso en Domótica, proyecto ambientales o industriales.

En una segunda iteración, llevar la aplicación y el sistema a un entorno en el que la red de sensores esté compuesta por nodos móviles. La aplicación desarrollada se plantea como una herramienta útil para desarrollar este tipo de redes denominadas MWSN (*Mobile Wireless Sensor Network*). De igual forma, se plantea incluir la aplicación desarrollada en redes vehiculares, redes MANET, etc.

Bibliografía

- [1] Web sistema operativo Tinyos, URL
www.tinyos.net
- [2] Web lenguaje de programación nesC, URL
www.nesc.sourceforge.net
- [3] Web standard 802.15.4, URL
<http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf>
- [4] Web ZigBee, URL
www.zigbee.org/
- [5] Datasheet TelosB, URL pdf
www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf
- [6] Web oficial de Crossbow, URL
www.xbow.com/Products
- [7] Interfaces y componentes de Tinyos 2.x, URL
<http://www.tinyos.net/tinyos-2.x/doc/nesdoc/telosb/index.html>
- [8] Web de Cygwin, URL
www.cygwin.com
- [9] Sistema operativo XubunTOS, URL
<http://toilers.mines.edu/Public/XubunTOS>
- [10] Wikipedia, 802.15.4, URL
<http://toilers.mines.edu/Public/XubunTOS>
- [11] Wikipedia, Redes de Sensores Inalámbricas, URL
http://es.wikipedia.org/wiki/Red_de_sensores
- [12] Transformación de unidades, dBm a mW, URL
<http://www.aubraux.com/design/dbm-to-milli-watts-calculator.php>
- [13] Web potencia Rssi, URL
<http://www.redsmalladas.com/?p=227>
- [14] Manual Tinyos-1.x, URL
<http://www.tinyos.net/tinyos-1.x/doc/>
- [15] Manual Tinyos-2.x, URL
<http://www.tinyos.net/tinyos-2.x/doc/>
- [16] Tossim: a simulator for Tinyos network,
Philip Levis, Nelson Lee.

- [17] Cadena 2.0: NesC tutorial,
Todd Wallentine
- [18] NesC 1.1 language refence manual,
David Gay, Philip Levis, David Culler, Eric Brewer
- [19] TPR2400/2420 Quick start Guide,
Crossbow
- [20] Tinyos Programming
Philip Levis
- [21] ZigBee / IEEE 802.15.4 Summary
Sinem Coleri Ergen