Proyecto Fin de Carrera
R.I.B.C
RED INALÁMBRICA DE BAJO CONSUMO

Guillermo Cabrera
José Inda
Martín Pérez

Tutor: Pablo Mazzara

INSTITUTO DE INGENIERÍA ELÉCTRICA
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

1 de julio de 2009

Índice general

1. Descripción y objetivo del proyecto **11**

2. Análisis de protocolos de capa MAC y elección del protocolo usado **13**

2.1. Funciones 13

2.2. Parámetros de manejo de prendido y apagado de la radio 14

2.3. Protocolos de capa MAC para redes inalámbricas de sensores. . . 15

2.3.1. Protocolo Sensor MAC (S-MAC) 15

2.3.2. Protocolo Timeout MAC 18

2.3.3. Low Power Listening. 19

2.3.4. Protocolo SCP 20

2.4. Comparación y elección 22

3. Plataforma e implementación **25**

3.1. Sistema operativo TinyOS 25

3.2. MLA 27

3.2.1. Diseño 27

3.2.2. Descripción 28

3.2.3. Interfaces con Capas Superiores 28

3.2.4. Componentes 29

3.3. Protocolo SCP implementado en MLA 30

4. Verificación del funcionamiento. **33**

4.1. Itroducción 33

4.2. Compilación del protocolo 33

4.3. Medida de tiempos de encendido de la radio con osciloscopio . . . 34
4.4. Primeras observaciones . . . 34
4.5. Solución a los problemas . . . 37
4.5.1. Tiempo de chequeo de canal. . . . 37
4.5.2. Radio encendida por dos períodos . . . 39
4.5.3. Radio encendida por varios períodos . . . 41
4.5.4. Pérdida de Sincronismo. . . . 41
4.5.4.1. Ajustes del preámbulo y del largo del período . . 42

5. Detalles del código y funcionamiento de la implementación 43
5.1. Introducción . . . 43
5.2. Chequeo periódico del canal . . . 43
5.2.1. Desde el disparo de `LplAlarm` hasta el chequeo de canal propiamente dicho. . . . 45
5.2.2. Chequeo de canal . . . 49
5.2.3. Luego del chequeo y apagado de la radio . . . 49
5.2.4. Duración del chequeo de canal . . . 52
5.3. Sincronismo . . . 53
5.4. Booteo . . . 54

6. Consumo 57
6.1. Introducción. . . . 57
6.2. Medida . . . 57
6.3. Niveles de consumo del mote . . . 57
6.4. Consumo en función de los tiempos y de los niveles de consumo de los motes . . . 60
6.5. Resultados . . . 63

7. Capas superiores 65
7.1. Introducción . . . 65
7.2. Recolección y Diseminación . . . 65
7.2.1. Introducción . . . 65
7.2.2. Recolección . . . 66

7.2.3. Diseminación 66
7.3. Identificación de Paquetes 67
7.3.1. Paquete de Medida 67
7.3.2. Paquete de control 67
7.3.3. Paquete de Topología 68
7.4. Time Stamp 68
7.4.1. Introducción y planteo del problema 68
7.4.2. Propuesta 69
7.5. Comunicación PC-Mote 69
7.5.1. Introducción 69
7.5.2. Puerto Serial 70
7.5.3. Interfaz TCP/IP 70
7.5.4. MIG 71
7.6. Manual de Usuario 71
7.6.1. Interfaz de Usuario. 71
7.6.1.1. Introduccion 71
7.6.1.2. Instructivo de uso 72

8. Conclusiones **77**
8.1. Conclusiones sobre SCP 77
8.2. Conclusiones sobre MLA 77
8.3. Conclusiones sobre SCP-MLA 78
8.4. Concluciones sobre el Consumo 79
8.5. Concluciones sobre Capa de Red-Capa MAC 79

Bibliografía **81**

A. Filosofia de NESC **83**

B. El chip cc2420 **85**
B.1. Descripción de capas 85
B.2. Capa CsmaC (CSMA/CA) 87
B.2.1. Radio Backoff 87
B.2.2. Comprobación de canal libre (CCA, Clear Channel Assessment) 88

Introducción

Una red inalámbrica de sensores es un conjunto de dispositivos con sensores que forman una red de datos inalámbrica para transmitir sus medidas. De esta forma, las medidas pueden ser recogidas por un nodo central, que puede llevar un registro de las medidas obtenidas por los sensores del entorno, o puede tomar acciones si las medidas recogidas cumplen determinadas condiciones de alarma.

Las redes inalámbricas de sensores se hacen necesarias cuando se quiere tener control de un determinado parámetro en un lugar remoto o extenso, donde no haya tendido eléctrico alguno, como por ejemplo, en la detección localizada de incendios en un bosque. La idea es que los dispositivos sean autónomos y se puedan comunicar entre sí para hacer llegar la información de las medidas a un lugar donde hayan más recursos disponibles para poder procesarla.

Actualmente esta tecnología se encuentra en pleno desarrollo, a los dispositivos se les suele llamar *motes* (que significa mota en inglés) por su pretensión de ser en el futuro tan pequeños como una partícula de polvo.

Entre las posibles aplicaciones de esta tecnología, se pueden mencionar:

- Detección de incendios, terremotos, inundaciones u otro desastre natural: En este caso los motes estarían monitoreando determinados parámetros y transmitirían una alarma en caso de detectar una anomalía. Desde un nodo central se puede detectar a tiempo el desastre y saber con suficiente precisión el lugar del suceso.

- Monitoreo de variables ambientales en una plantación agrícola: Los motes pueden estar midiendo constantemente temperatura, humedad, luz, etc. y transmitir las medidas a un nodo central que lleve un registro de todas las medidas. Posteriormente el productor puede observar el registro y tomar desiciones.

- Monitorización de un hábitat: Para estudiar los movimientos de una determinada población animal con sensores que detecten la presencia de individuos (detectores de movimiento por ejemplo).

- Detección de movimiento enemigo en campo de batalla: Se podría arrojar con aviones, a territorio hostil, motes con detectores de movimiento, para que detecten y transmitan la presencia del enemigo.

- Control de tráfico: Una red inalámbrica de sensores podría monitorear el movimiento del tráfico para prevenir embotellamientos o investigar su comportamiento.

- Domótica y edificios "inteligentes": Se han propuesto muchas formas de usar esta tecnología en el campo de la domótica y los edificios "inteligentes".

- Control médico: Para que los pacientes que necesiten monitorear algún parámetro del organismo puedan moverse libremente en una determinada área.

Por su definición, los motes consisten de, por lo menos, un transmisor-receptor de radio conectado a algún sensor. Existen varias compañías que fabrican motes para redes inalámbricas. Estos motes son pequeños dispositivos con todo lo necesario para facilitar el montaje de una red inalámbrica de sensores con ellos. Por lo general, estos motes disponibles comercialmente están constituidos por:

- Sensores: Los motes ya vienen con algunos sensores de fábrica, y al mismo tiempo, tienen la posibilidad de conectar cualquier otro sensor externo. Si se quiere tomar las medidas con buena precisión, o el sensor necesita estar a una determinada distancia del transmisor de radio (un par de metros por ejemplo), es necesario acoplar un sensor externo.

- Transmisor-receptor de radio.

- Microcontrolador: Donde se prevé que se encuentre toda la inteligencia del mote. Las empresas que fabrican motes por lo general venden los mismos con el microcontrolador desprogramado y con una manera de programarlos, para que el cliente programe los motes de acuerdo a la aplicación específica.

- Suministro de energía: En algunos modelos esto es un espacio para colocar pilas, en otros modelos el suministro de energía son celdas solares, etc.

- Puerto serial de comunicaciones: El puerto serial de comunicaciones tiene por lo menos dos aplicaciones importantes. Una es ser el medio para programar el microcontrolador de los motes, y otra, permitir conectar uno de los motes a una máquina con más recursos para que pueda oficiar de nodo central.

Entre las compañías que fabrican motes y ofrecen servicios de redes inalámbricas sen encuentran:

- CROSSBOW: Especializada en el mundo de los sensores, desarrolla plataformas hardware y software que dan soluciones para montar redes de sensores inalámbricas. Entre sus productos se encuentran las plataformas Mica, Mica2, Micaz, Mica2dot, telos y telosb.
- SENTILLA: Antiguamente llamada MOTEIV, formada por Joseph Polastre, antiguo doctorando de un grupo de trabajo de la Universidad de Berekeley. Ha desarrollado la plataforma Tmote Sky y Tmote Invent.
- SHOCKFISH: Empresa suiza que desarrolla TinyNode. A partir de este tipo de mote en Laussane se ha llevado un proyecto que implementa una red de sensores en todo el campus de la "Ecole Polytechnique Fédérale de Lausanne".

Para el desarrollo de las redes inalámbricas de sensores, se han propuesto algunos sistemas operativos para facilitar la programación de los microcontroladores en los motes [16]. Dos de ellos son *TinyOS* [5] y *MANTIS Operative System* (MOS) [4]. TinyOS prioriza el ahorro de consumo de energía y memoria, mientras que MOS simplifica la programación de las aplicaciones basadas en él, ahorrando menos energía y memoria. Como se anota en [16], no es evidente que las redes de sensores necesiten un sistema operativo especial para programar los motes. La mayoría de los sistemas embebidos, o están programados sin ningún sistema operativo, o contienen versiones especiales de sistemas operativos originalmente pensados para sistemas más grandes (Windows, Linux, etc). Las redes de sensores tienen características únicas que originó el desarrollo de estos sistemas operativos.

También existen algunos estándares aplicables a redes inalámbricas de sensores. El estándar 802.15.4 de la IEEE [6] norma las capas inferiores en una la red de datos inalámbrica de bajo consumo, mientras que ZigBee complementa el estándar anterior y regula las capas superiores. Estos estándares están pensados más bien para la domótica y no son del todo adecuados para monitoreo de variables ambientales. Por otro lado, una aplicación específica puede ser autocontenida y no tiene por qué seguir estos estándares. Por lo tanto, el uso de los mismos no es universal, y se están estudiando protocolos que mejoran el desempeño respecto de los estándares.

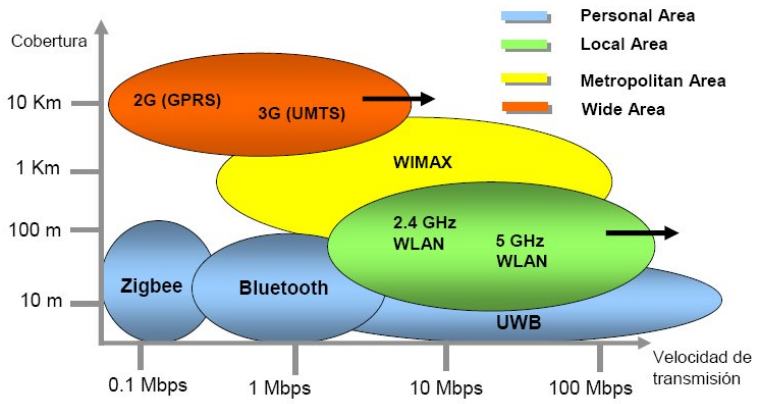

Figura 1: Comparación de normas para redes inalámbricas y ubicación de ZigBee (para redes de sensores) en la comparación.

Capítulo 1

Descripción y objetivo del proyecto

Este proyecto tuvo como objetivo el diseño de una red inalámbrica de sensores para uso agrícola, con énfasis en afinar el diseño respecto al bajo consumo.

En la Facultad de Ingeniería existían antecedentes de desarrollo en este campo, en especial, los proyectos de fin de carrera SI-AGRO [14] y SIAGRO2 [9] donde se establecieron exitosamente redes inalábricas de sensores para aplicaciones agrícolas.

Si bien el bajo consumo estaba entre los objetivos de los proyectos mensionados, en este proyecto se plantea como la prioridad, porque el mayor desafío en este tipo de redes es encontrar un mecanismo de comunicación confiable y de bajo consumo.

El hardware utilizado fueron los motes TmoteSky, que cuentan con un microcontrolador MSP430 de Texas Instruments, puerto USB y son alimentados con 2 pilas AA. El objetivo es que las pilas AA que alimentan los motes duren un año (suponiendo pilas alcalinas).

El software se basó en el sistema operativo TinyOS, que es de código abierto y la Facultad de Ingeniería ya tiene experiencia en dicho sistema.

Figura 1.1: Tmote Sky

En una red de sensores el bajo consumo se logra apagando la radio la mayor parte del tiempo posible y prenderla solo lo necesario, ya que es la radio el elemento que más consume en un mote.

Por lo tanto, la parte principal del diseño que determina el consumo de un mote es el protocolo de control de acceso al medio (protocolo MAC por sus siglas en inglés), y por este motivo el proyecto se centró en el diseño de dicho protocolo. El trabajo supuso estudiar diferentes protocolos MAC, implementar el más adecuado en TinyOS, grabarlo en los motes y verificar su funcionamiento.

Por otro lado se completó el resto de una aplicación donde se arma una red y desde una PC se recogen los datos sensados en todos los motes de la misma. El resto de la aplicación incluye el montaje de la red por parte de los motes, la comunicación entre el mote base (conectada a una PC) y la PC conectada al mismo, una aplicación en dicha PC que retransmite los datos a un puerto TCP, y una aplicación que lee los datos del puerto TCP y los procesa con una base de datos MySQL y una interfaz de usuario.

La aplicación final permite tener el mote base conectado a una PC conectada a internet que solo retransmite los datos a un puerto TCP, la interfaz de usuario en otra PC conectada a internet, y la base de datos en otra máquina también conectada a internet.

El resto de la documentación estará organizada de la siguiente forma:

- En el capítulo 2se introduce la capa MAC de la red de datos y su rol especial en redes de sensores. A su vez, se analizan y comparan un conjunto de protocolos y se concluye cuál es el óptimo.
- En el capítulo 3 se introduce una implementación particular que, entre otras cosas, implementa el protocolo que en el capítulo anterior se concluye como óptimo.
- En el capítulo 4 se detallan las primeras pruebas realizadas en laboratorio para verificar su funcionamiento, así como los cambios y ajustes que fue necesario realizar.
- En el capítulo 5 se hace un análisis más detallado de la implementación usada, intentando determinar por qué los resultados esperados en su momento no fueron obtenidos.
- En el capítulo 6 se hace un análisis del consumo de la implementación, comprobando si se obtienen los objetivos y estableciendo límites de funcionamiento.
- En el capítulo 7 se comenta lo trabajado en las capas superiores.
- En el capítulo 8 se analizan las conclusiones.

Capítulo 2

Análisis de protocolos de capa MAC y elección del protocolo usado

2.1. Funciones

En una red de sensores inalámbricos, la capa de control de acceso al medio cumple las mismas funciones que en cualquier otra red de datos inalámbrica, pero adicionalmente se incluyen nuevas funciones relacionadas con el bajo consumo.

La función principal de la capa MAC, en cualquier red de datos donde varios nodos comparten el mismo medio físico, es evitar la contención. Parte de la solución al problema es la multiplexión (FDMA TDMA CDMA, etc), aunque a ésta en general se la considera como parte de la modulación dentro de la capa física. De todos modos, muchas veces la multiplexión no es suficiente y en esos casos la solución debe estar en la capa MAC.

Adicionalmente, en una red inalámbrica la capa MAC también debe resolver los problemas de la estación expuesta y de la estación oculta.

Por otro lado, el protocolo de capa MAC, al proveerle servicio a las capas superiores, debe cumplir determinadas condiciones que dependen de la aplicación, como retardo, tasa de transferencia, equidad, calidad de servicio (QoS) o soportar múltiples servicios.

En una red de sensores, la potencia es un recurso crítico y la eficiencia energética es un punto muy importante en todo el diseño de la red. El receptor-transmisor de radio es el elemento que más consume energía en un mote. Por lo tanto, la manera de aumentar al máximo la eficiencia en una red de sensores es tener la radio apagada la mayor parte del tiempo, y prenderla solo lo estrictamente necesario. De esta manera se introduce el término de *ciclo de trabajo*, que es la relación de tiempo en que la radio permanece prendida, comparada con el tiempo total.

La función de prender y apagar la radio recae en la capa MAC, ya que típicamente es la que decide el momento en que se establece el envío de un mensaje. Por lo tanto, las implementaciones de capa MAC en una red inalámbrica de sensores resulta fundamental, y son muy diferentes a lo usado en otras redes, ya que intentan disminuir lo máximo posible el ciclo de trabajo de la radio.

Como el elemento principal de consumo de un mote es la radio, se puede suponer para simplificar que los motes presentan dos niveles principales de consumo, uno cuando la radio está prendida, y el otro cuando la radio está apagada. Las medidas realizadas en laboratorio mostraron que esta suposición es completamente razonable.

2.2. Parámetros de manejo de prendido y apagado de la radio

En un protocolo de capa MAC en una red inalámbrica de sensores, los parámetros de tiempo que determinan el prendido y apagado de la radio resultan fundamental en el consumo del mote. Los parámetros introducidos aquí se usarán en el resto de la sección.

En la figura 2.1 se muestra un esquema general de consumo de un mote en función del tiempo, donde se muestran los dos niveles de consumo, y se muestran los parámetros de tiempo que determinarán el consumo total. Estos parámetros se introducirán a continuación. Casi todos los protocolos de capa MAC vistos siguen aproximadamente este esquema. En 2.3, donde se comparan diferentes protocolos en particular, se verá cómo se relacionan estos parámetros en dichos protocolos.

Casi todos los protocolos se basan en mantener la radio apagada durante largos períodos y prenderla periódicamente durante un corto período por si algún mote vecino le quiere transmitir un mensaje. Al prender la radio, tan pronto como el mote determine que ningún vecino quiere transmitirle mensaje, apaga la radio nuevamente. Se introducen entonces t_{Sleep} el tiempo en que la radio se mantiene apagada, t_{Listen} el tiempo en que la radio se mantiene prendia para reconocer si un vecino quiere transmitir un mensaje, y T el período $t_{Sleep} + t_{Listen}$.

Por lo gneral, el tiempo necesario para transmitir un mensaje es mayor o igual a t_{Listen}, sea t_{TX} el tiempo en que la radio se mantiene prendida para transmitir un mensaje.

En principio parecería razonable que el tiempo requerido para recibir un mensaje debería ser casi igual al tiempo de envío. Como se ve en 2.3, en algunos protocolos el tiempo de recepción de mensaje es bastane menor a t_{TX} como se muestra en la figura 2.1, sea entonces t_{RX} el tiempo en que la radio se mantiene prendida cuando se recibe un mensaje.

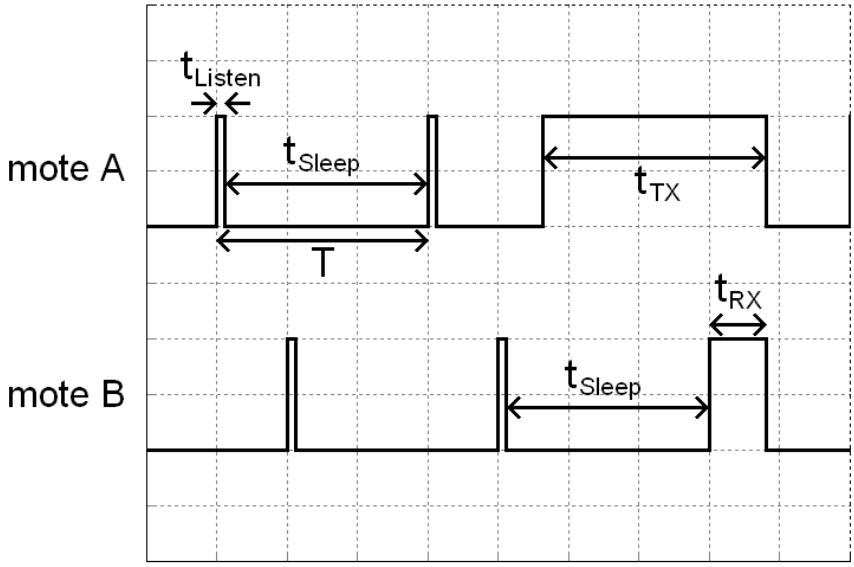

Figura 2.1: Esquema de tiempos de prendido y apagado de la radio en un protocolo genérico.

2.3. Protocolos de capa MAC para redes inalámbricas de sensores.

En esta sección se introducen las implementaciones y conceptos más importantes y difundidas de protocolos de capa MAC para redes inalámbricas de sensores. No es exhaustivo pudiéndose encontrar otros conceptos e implementaciones menos difundidos.

2.3.1. Protocolo Sensor MAC (S-MAC)

Sensor MAC [19] es uno de los primeros protocolos de capa MAC desarrollado para redes de sensores.

La idea principal del protocolo S-MAC es la sincronización. En este protocolo, el prendido de la radio se hace sincronizadamente con los nodos vecinos, asegurando que cuando un nodo desea enviar un mensaje, lo haga cuando sus vecinos tengan la radio prendida. En las figuras 2.2 y 2.3 se muestra esquemáticamente su funcionamiento.

En el contexto de este protocolo se definen un *período activo* (active period) y un *período de sueño* (sleep period) como se muestra en la figura 2.3. Dentro del período activo, hay un período de intercambio de mensajes de sincronismo, y

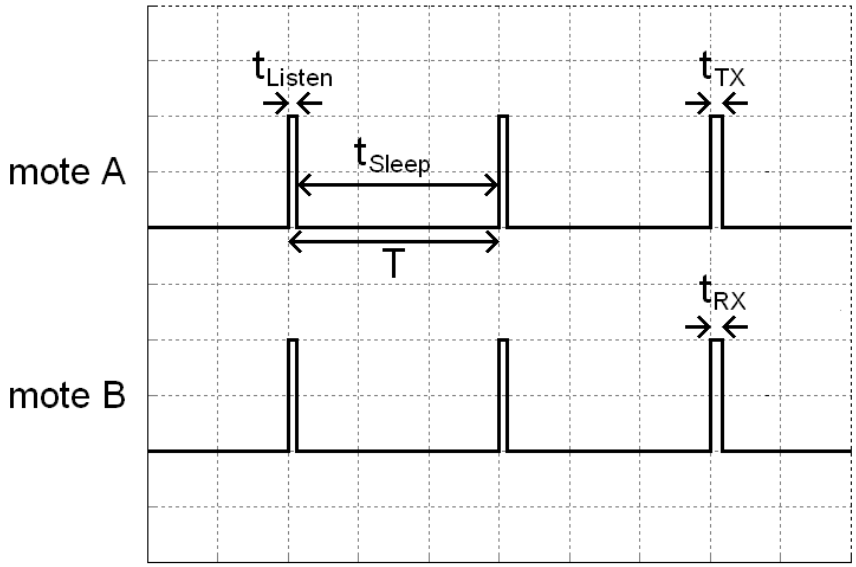

Figura 2.2: Esquema de tiempos en el protocolo S-MAC.

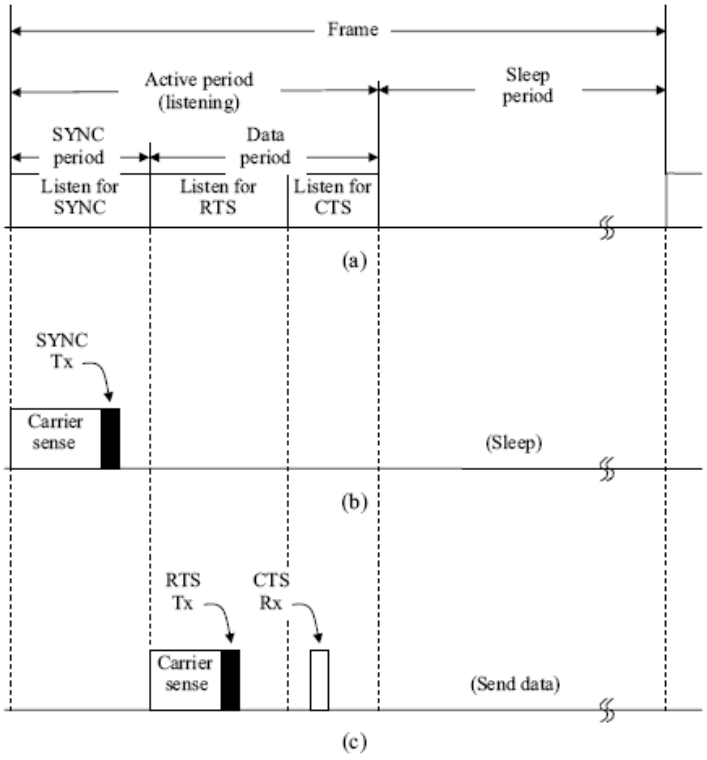

Figura 2.3: Esquema de funcionamiento de S-MAC. (a) Recepción. (b) Transmisión de mensaje de sincronización. (c) Transmisión de datos. (Extraído de [16])

un período de intercambio de mensajes RTS y CTS. Cuando hay transmisión de datos, se intercambian mensajes de RTS y CTS, y luego se usa el período de inactivo para la transmisión de datos.

La deriva de los relojes es pequeña disminuyendo el riesgo de desincronización. El período de sueño está determinado principalmene por la latencia requerida.

Para mantener la sincronización, los motes deben intercambiar mensajes de sincronización indicando su *horario* (schedule) marcando el momento de inicio del siguiente ciclo activo-sueño. Este momento se indica de forma relativa, indicando el momento, contando el tiempo desde que se envía el mensaje. Los mensajes de sincronización se envían periódicamente. El período envío de mensajes de sincronización se llama *período de sincronización.*

Para establecer la sincronización, primero, un nodo escucha un tiempo fijo, por lo menos tan largo como el período de sincronización, para detectar un mensaje de otro nodo. Si el nodo no detecta ningún mensaje durante ese período, elige su propio horario y lo sigue. Además, inmediatamente transmite un mensaje de sincronización, el cual se vuelve a envier cada vez que transcurre un período de sincronización.

Si un nodo detecta un mensaje de sincronización antes de transmitir el suyo, obedece el horario recibido, y comienza a transmitir mensajes de sincronización a partir del siguiente período activo, y luego cada vez que transcurre un período de sincronización.

En redes multi-hop puede ocurrir que un nodo reciba dos horarios diferentes generados por nodos que no se detectan entre sí. Para este caso, los proponentes del protocolo plantean más de una solución con ventajas y desventajas. En cualquier caso, se incluye el *descubrimiento de vecinos* (neighbor discovery) que consiste en, cada tanto, escuchar durante todo el período de sincronización, para evitar que nodos vecinos elijan diferentes horarios y nunca se detecten.

Cuando no hay intercambio de mensajes, el período activo corresponde al tiempo t_{Listen}, y el período de inactivo se corresponde con t_{Sleep}. Cuando hay intercambio de mensajes, los tiempos t_{TX} y t_{RX} son prácticamente iguales e incluyen un t_{Listen}más el tiempo necesario para transmitir los mensajes. La transmisión de datos puede ser más corta que el período activo, por lo tanto, t_{TX} y t_{RX} son levemente mayores a t_{Listen} como se muestra en la figura 2.2, y para cargas livianas el ciclo de trabajo se puede considerar constante.

S-MAC también introduce un concepto original que es la *escucha adaptativa* (adaptive listening). Si un nodo detecta algún mensaje RTS o CTS para una comunicación que no lo involucra, apaga su radio, pero en vez de prenderla recién en el siguiente ciclo, la prende cuando se haya terminado la comunicación entre los nodos correspondientes (los mensajes RTS y CTS incluyen el largo de la comunicación). De esta forma, si el nodo en cuestión es el destino del siguiente salto, puede recibir la información inmediatamente después.

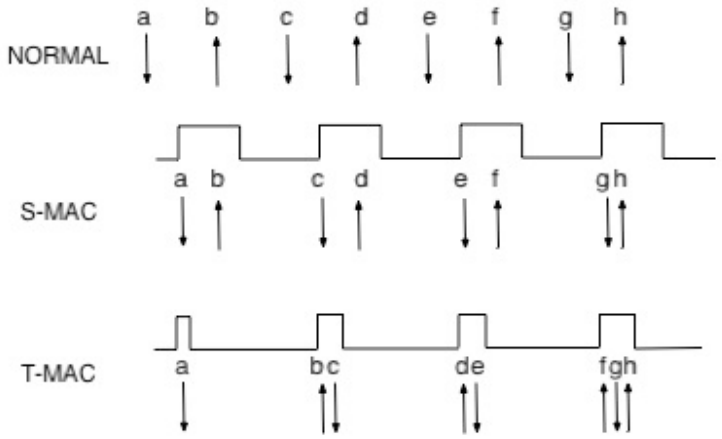

Figura 2.4: Comparativa SMAC y TMAC. (Extraído de [17])

2.3.2. Protocolo Timeout MAC

Timeout MAC [17], es un protocolo que intenta mejorar al anterior S-MAC. La idea es disminuir el ciclo de trabajo en casos en que la carga de mensajes a transmitir es variable haciendo el período activo *adaptativo* de a cuerdo a la carga de mensajes que se necesitan enviar. Los proponentes afirman que tiene una performance similar a S-MAC en casos de carga constante, mientras que mejora por un factor de 5 con carga variable.

En la Figura 2.4 se muestra una comparación ilustrativa entre T-MAC y S-MAC. La sincronización, RTS CTS y los sistemas de acuse de recibo son similares a S-MAC.

El protocolo T-MAC presenta el problema del *sueño temprano* en las redes multihop, que ocurre cuando hay contención y se explicará a continuación.

Se suponen cuatro nodos A, B, C y D en una red multihop según figura 2.5.

Se supone que A está enviando un mensaje a B, y C necesita enviar un mensaje a D pero está en el radio de comunicación de B. En este caso, C no puede enviar un RTS a D, de otra manera causaría una colisión con B. Mientras D esté fuera del rango de A y B, no detectará actividad y se va a dormir, incluso cuando C tiene un mensaje para él. El mismo problema ocurre otra vez en el siguiente ciclo de trabajo, siempre y cuando A o B obtengan el acceso al canal antes de C.

Los desarrolladores de T-MAC ofrecen algunas soluciones al problema, pero incluso con estas soluciones, la capacidad de envío/recepción máxima del protocolo se ve comprometida, cuando se la compara con esta misma capacidad sobre S-MAC.

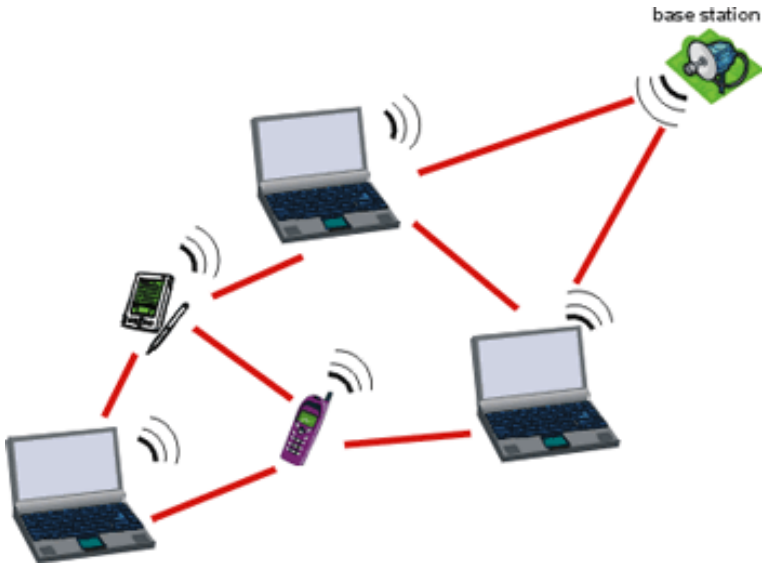

Figura 2.5: Una red multihop

2.3.3. Low Power Listening.

Una aproximación completamente distinta, basada en la suposición de que la mayor parte del consumo se debe a la escucha del canal, es usar lo que se llama *Low Power Listening* (LPL), desarrollado en [7] (WiseMAC) y en [11] (B-MAC) entre otros. En este caso los nodos no se prenden de manera sincronizada. En vez de eso, cada nodo revisa el canal a intervalos regulares de tiempo, pero de manera desincronizada, para ver si tiene mensajes pendientes para recibir, semejante a la figura 2.1. Cuando un nodo debe enviar un mensaje, envía primero un preámbulo al menos tan largo como el intervalo de revisión. Cuando el receptor revise el canal, va a detectar el preámbulo y mantendrá la radio prendida para recibir el mensaje. Si un nodo revisa el canal y no detecta un mensaje pendiente para el mismo, apaga su radio inmediatamente. De esta forma, el tiempo t_{Listen} se reduce al mínimo, porque solo se prende lo necesario para detectar actividad en el canal, a diferencia de los protocolos anteriores en que incluyen varios períodos de control.

Además, en principio los nodos lo primero que revisan es si hay actividad en el canal, y para esto no es necesario prender la radio completamente. Si no detectan actividad se apagan inmediatamente. Por lo tanto, cuando no hay mensajes para enviar, el consumo es mínimo porque la energía requerida para revisar el canal es muy poca.

Si se detecta actividad, significa que hay un nodo emisor que está transmitiendo un preámbulo, o que se está estableciendo una comunicación. El segundo paso si se detecta actividad en el canal, es verificar si hay un mensaje pendiente para el nodo. Si no hay mensaje pendiente se apaga la radio, y si la hay, se establece la comunicación.

El tiempo t_{TX} incluye el preámbulo y resulta bastante mayor al de los protocolos anteriores. Por otro lado, cuando se empieza a enviar el preámbulo el receptor continúa dormido, por lo tanto t_{RX} resulta casi siempre bastante menor a t_{TX}.

En la implementación más simple de este protocolo, el preámbulo que envía el emisor antes del mensaje es de largo fijo (t_{TX} constante). El receptor cuando detecta que hay mensaje pendiente espera a que termine el preámbulo y recién entonces se inicia la comunicación.

El largo de los preámbulos establece un compromiso para determinar el ciclo de trabajo a usar.

Una primera mejora es que el receptor no espere a que se termine el preámbulo del emisor, sino que envíe un reconocimiento para que el emisor detenga el preámbulo e inicie inmediatamente la comunicación. Esta mejora es posible con radios que permitan transmisión y recepción simultánea.

Otra mejora, presentada en [15], llamada *Adaptive LPL*, es aumentar el ciclo de trabajo cuando hay actividad (T variable). Esto permite optimizar el consumo en redes con actividad no constante como en el caso de redes que informan sobre eventos inusuales (detección de incendios forestales, detección de ejército enemigo en campo de batalla).

Este protocolo tiene la ventaja de que el consumo de energía es muy bajo si no hay datos para enviar ya que t_{Sleep}es mínimo. Pero por otro lado, se tienen preámbulos largos que pueden resultar en pérdida innecesaria de energía. También poseen la desventaja de que es difíciles de optimizar en redes donde los datos no se generan periódicamente. Además en recepción consumen poco, pero en transmisión consumen mucho. Una posible solución a este problema es el protocolo SCP.

2.3.4. Protocolo SCP

En el trabajo [18] se propone una variante de LPL llamado *Schedualed Channel Polling* (SCP).

Análogamente a LPL, este protocolo se basa en que cada mote realiza periódicamente una escucha de canal para ver si otro mote vecino le quiere transmitir algún mensaje. El prendido de la radio para realizar dicha escucha de canal es mínimo porque solo se detecta si hay actividad o no en el canal. Si un mote detecta actividad en el canal, interpreta que un mote vecino quiere transmitir un mensaje y deja la radio prendida para recibir dicho mensaje. Un mote que quiera transmitir un mensaje, debe primero transmitir un tono de preámbulo para que los vecinos lo detecten, y luego debe transmitir el mensaje.

La diferencia con LPL es que los motes hacen la escucha de canal de forma sincronizada. Por lo tanto el mote que quiere enviar un mensaje ya sabe cuando sus

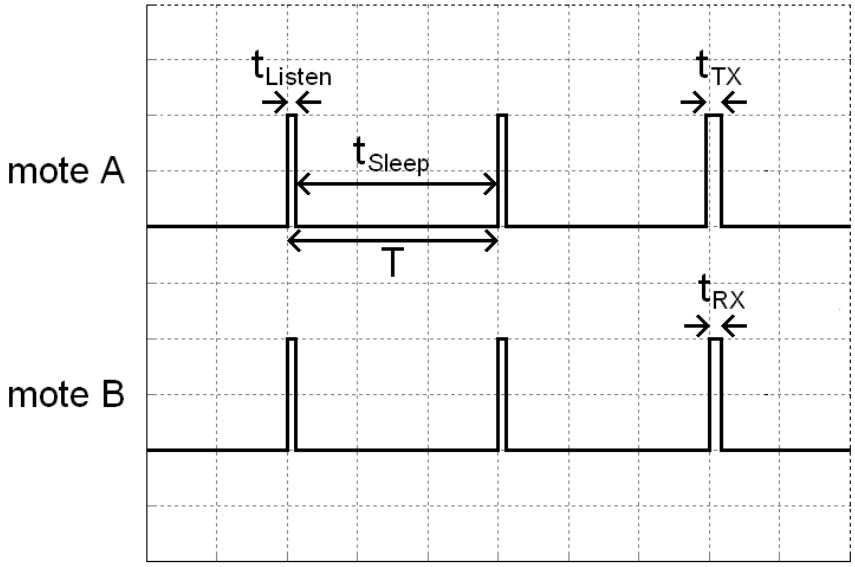

Figura 2.6: Esquema de tiempos en el protocolo Schedualed Channel Polling.

vecinos van a hacer la escucha, y comienza a enviar el tono de preámbulo un tiempo corto antes. De esta forma el preámbulo se reduce significativamente respecto del protocolo LPL.

El mecanismo de sincronización es el mismo al del protocolo S-MAC.

En la figura 2.6 se muestran los parámetros de tiempos para este protocolo y en la figura 2.7 se muestra una comparación con LPL.

En este protocolo t_{Listen} es mínimo como en LPL pero t_{TX} se reduce significativamente respecto a LPL permitiendo además aumentar T (aumentando a la vez t_{Sleep}) a costa de aumentar la complejidad y agregar carga de sincronización.

Además, los proponentes del protocolo implementan Adaptive LPL explicado más arriba y lo que ellos llaman *Two-Phase Contention* y *Overhearing Avoidance Based on Headers*.

La idea de Two-Phase Contention es disminuir la carga del mecanismo de contención, aprovechando que enviar datos es un proceso de dos etapas, ya que primero se envía el tono de preámbulo, y después se envían los datos. El mecanismo consiste en introducir dos ventanas de contención, una anterior al envío del tono, y otra anterior al envío de los datos. De esta forma, como se explica en [18] se disminuyen a la vez el tiempo total de contención y la probabilidad de colisión. Hay que tener en cuenta que el mecanismo tolera sin problemas colisión en el envío de los tonos.

Overhearing Avoidance Based on Headers consiste en que los receptores examinen

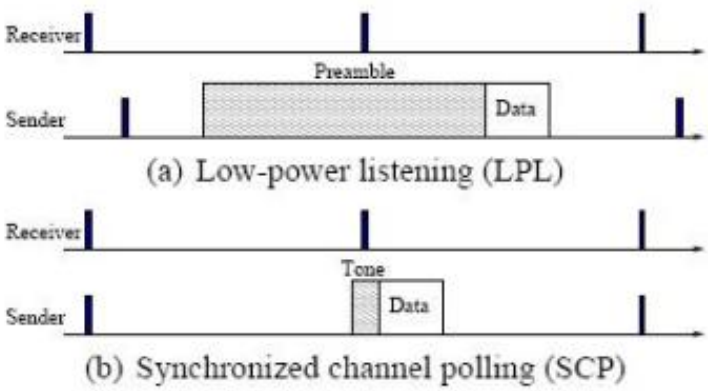

Figura 2.7: Schedualed Channel Polling. (Extraído de [18])

los encabezados del mensaje que reciben para decidir si el mensaje es para ellos. Si un nodo encuentra que el mensaje es para otro destinatario, apaga la radio antes de recibir el resto del mensaje con los datos.

Los proponentes del protocolo SCP aseguran que consiguen bajar el ciclo de trabajo a un orden de magnitud menor que en los protocolos anteriores.

Este protocolo presenta la desventaja de un alto costo de mantenimiento de la sincronización programada y potencialmente el requisito de mantener múltiples calendarios.

2.4. Comparación y elección

Como ya se mencionó, el factor crítico es el consumo. El consumo en todos los casos depende del ciclo de trabajo, es decir, del tiempo en que la radio permanece prendida en comparación con el tiempo total. En todos los casos el ciclo de trabajo es mucho menor que 1.

En los protocolos S-MAC las radios de nodos vecinos se prenden sincronizadamente para iniciar el período activo. El período activo debe ser lo suficientemente largo como para que quepan varias ventanas de contención, y puede ser más largo que el tiempo necesario para transmitir los mensajes de datos. Con poca carga, el ciclo de trabajo se puede considerar constante todo el tiempo, y está determinado principalmente por la latencia requerida.

El protocolo T-MAC es más eficiente que S-MAC porque un nodo se apagará antes que el resto si determina que en ese ciclo no va a tener ninguna comunicación, permitiendo que el ciclo de trabajo disminuya cuando no haya actividad. T-MAC no tiene bien resuelto el problema visto del sueño temprano en casos de contención.

El protocolo Low Power Listening tiene una política distinta a los anteriores. La idea es disminuir al máximo el consumo de los dispositivos que no envíen ni reciban mensajes, ya que el mismo revisa el canal y si no detecta actividad se apagará inmediatamente, reduciendo al mínimo el ciclo de trabajo. Además, el consumo se reduce aun más porque solo se realiza una preescucha para detectar actividad en el canal ya que no se necesita prender la radio completamente para esto. En las implementaciones existentes, la revisión del canal es unas 10 veces menos costosa que escuchar el período activo de S-MAC y T-MAC.

Por otro lado, los dispositivos que desean enviar un mensaje necesitan enviar preámbulos muy largos. Por lo tanto, el protocolo LPL reduce al mínimo el consumo cuando no hay actividad, a costa de consumir más cuando se transmite un mensaje. En la práctica, el ciclo de trabajo no baja del 1 %.

El protocolo SCP [18] combina las ventajas de los protocolos anteriores y sus proponentes aseguran que se logra un consumo varias veces menor. Cuando no hay actividad, el ciclo de trabajo es mínimo como en LPL con el único agregado de tener que mantener la sincronización, y cuando hay actividad los preámbulos se minimizan porque la sincronización determina que un dispositivo sabrá cuándo sus vecinos revisarán el canal.

Además se encontró una implementación de este protocolo en el trabajo MLA que se analiza en el capítulo siguiente.

Dadas las comparaciones entre los distintos protocolos de la capa MAC mencionadas anteriormente, la naturaleza de los datos con los cuales vamos a trabajar y teniendo en cuenta los objetivos del proyecto es que se decidió utilizar el protocolo SCP.

Capítulo 3

Plataforma e implementación

La plataforma hardware para los nodos de la red fue con motes Tmote Sky (figuras 1.1 y 3.1) que cuentan con que cuentan con un microcontrolador MSP430 de Texas Instruments, radio CC2420 de Chipcon, puerto USB y son alimentados con 2 pilas AA.

Para la implementación software se usó el trabajo *MAC Layer Architecture* (MLA) [12] que es una arquitectura que implementa varios protocolos para redes de sensores, entre ellos, el protocolo SCP elegido. El software está basado en el sistema operativo TinyOS.

En este capítulo se introducen los conceptos básicos de TinyOS, se hace una descripción de MLA y se introduce la implementación MLA del protocolo SCP elegido y descripto en el capítulo anterior.

3.1. Sistema operativo TinyOS

TinyOS se define como un *sistema operativo de código abierto* diseñado para redes inalámbricas de sensores [5]. Usa una *arquitectura basada en componentes* y un modelo de ejecución *event-driven* (conducido por eventos). Tiene una *biblioteca de componentes* con varias herramientas implementadas. Los componentes se pueden usar como son, o se pueden modificar de a cuerdo a la aplicación específica.

TinyOS no es un sistema operativo en el sentido comúnmente entendido en las computadoras de escritorio, como el "software base" sobre el cual se agrega cualquier número de aplicaciones específicas (programas) desarrolladas de manera independiente. Cada aplicación programanda sobre TinyOS asume completo control del dispositivo y solo se puede grabar una aplicación a la vez en cada mote. TinyOS se puede entender por lo tanto como una herramienta de desarrollo de software para motes de redes de sensores.

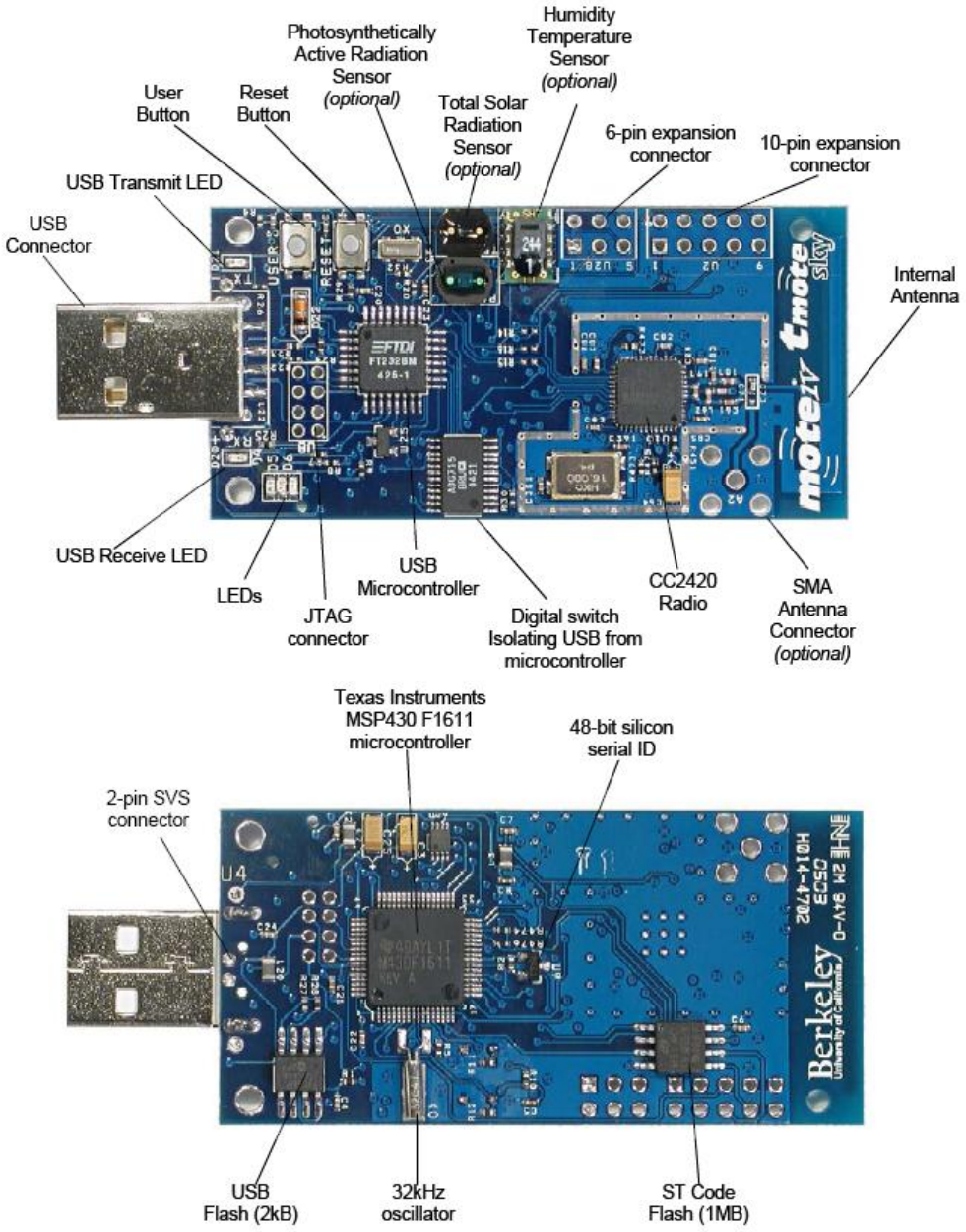

Figura 3.1: Tmote Sky con todos sus componentes (sin portapilas)

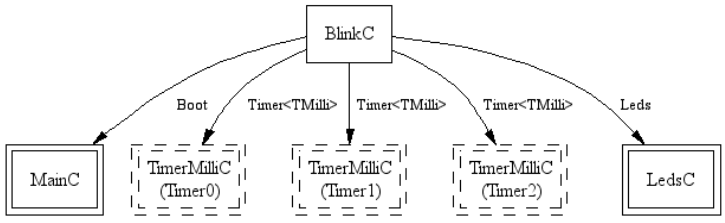

Figura 3.2: Diagrama ejemplo de una configuración

TinyOS tiene la ventaja de tener ya implementados varios mecanismos de manejo de radio, protocolos y drivers que simplifican la programación de una aplicación. Como desventaja presenta un conjunto de particularidades propias que hay que aprender a manejar antes de poder programar aplicaciones eficientemente.

Tiene un lenguaje de programación propio asociado, llamdo *nesC*, introducido sobretodo para implementar la arquitectura basada en componentes que usa la herramienta. En este sentido, se reconocen dos tipos de componentes: *módulos* y *configuraciones*. La diferencia es que los módulos tienen una descripción funcional y los componentes son el ensamblado de varios subcomponentes con sus interfaces.

3.2. MLA

MLA (MAC Layer Architecture) se define como una arquitectura basada en componentes para el desarrollo de protocolos MAC, con uso eficiente de la energía, para redes inalámbricas de sensores. MLA consiste en usar componentes reutilizables que implementen un conjunto común de características compartidas por los protocolos MAC, así como abstracciones que puedan encapsular la complejidad de plataformas de hardware. A través de MLA, los desarrolladores han implementado cinco protocolos en TinyOS 2.0.1, entre ellos SCP.

MLA se encuentra implementado, y promete una simplificación del trabajo y un correcto funcionamiento de todos los protocolos.

3.2.1. Diseño

Las políticas utilizadas para el diseño fueron que la arquitectura debía presentar una interfaz limpia a las capas superiores, exponiendo el menor número de detalles del hardware como fuera posible. En segundo lugar, el stack de la radio debía exportar las necesidades funcionales de bajo nivel utilizando un conjunto de interfaces independientes de la plataforma. En tercer lugar, a través de la funcionalidad común de los protocolos MAC, debían ser identificados y aplicados dentro de un conjunto optimizado de componentes reutilizables.

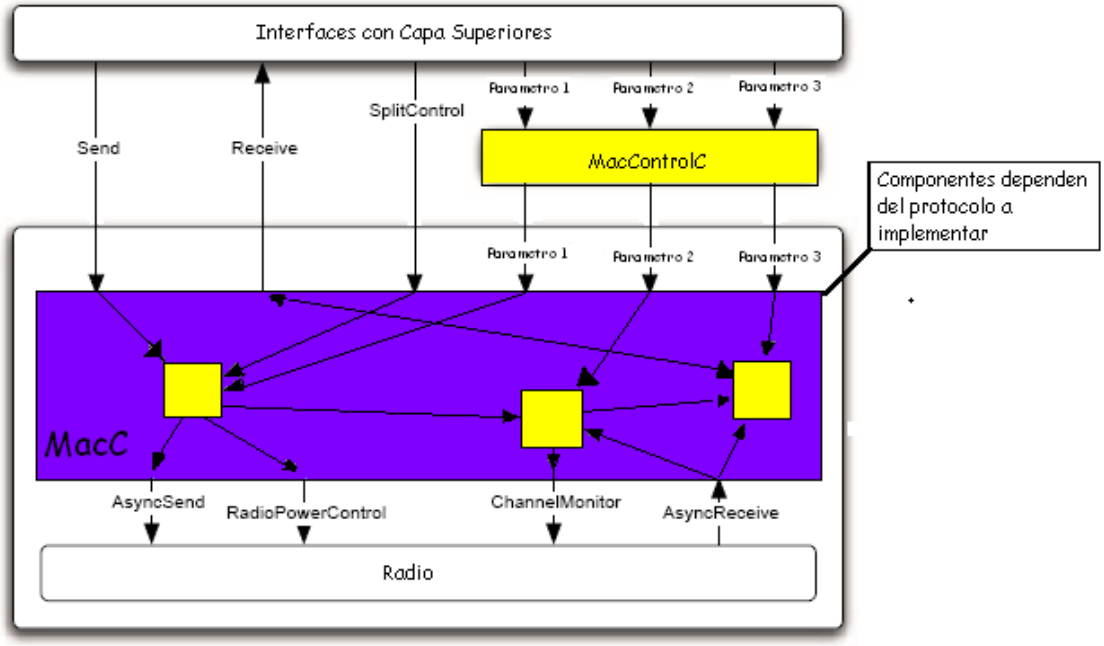

Figura 3.3: Interfaces MLA

3.2.2. Descripción

La arquitectura está constituida por dos tipos de componentes, los de alto nivel, independientes del hardware, y los de bajo nivel, dependientes del hardware. Los de alto nivel brindan flexibilidad ya que permiten que diferentes caracteristicas de los protocolos MAC sean integrados en una nueva implementación de un protocolo. Los de bajo nivel proporcionan abstraccion, ya que exportan las interfaces que apoyan al desarrollo de los componentes de alto nivel logrando independizarse de las características específicas de una determinada radio o microprocesador.

La Figura 3.3 muestra un panorama general de cómo estos componentes pueden ser utilizados para construir protocolos MAC más complejos dentro de MLA. Varios componentes se agrupan dentro de un conjunto más general MAC, usando un conjunto unificado de interfaces proporcionada por la radio, y exponiendo parcialmente un conjunto de interfaces unificadas para las capas superiores.

3.2.3. Interfaces con Capas Superiores

Las interfaces que MLA proporciona a las capas superiores deben estar desarrolladas de tal forma que el comportamiento de los protocolos de la capa MAC en tiempo de ejecucion sean lo más transparente posible para el usuario, y asimismo, al momento de realizar aplicaciones, los desarrolladores no necesitarán conocer

como estan compuestos internamente los protocolos.

Los desarrolladores de aplicaciones hacen el mayor esfuerzo en elaborar aplicaciones que traten las entradas y salidas de paquetes como una caja negra, además estos deben ser capaces de tratar a los Protocolos MAC como una sola entidad coherente, y por lo tanto, ser capaz de insertar, reemplazar o quitar un protocolo MAC con muy poco esfuerzo. Se logran estos dos objetivos al exponer todas las interfaces de las aplicaciones de la capa MAC a través de dos componentes distintos, como se muestra en la Figura 3.3.

Cada protocolo MAC esta compuesto por componentes reusables de MLA y por componenentes especificos del protocolo, los cuales forman la MacC. Con el fin de que su funcionamiento sea lo más transparente posible para el usuario, el componente MacC utiliza un conjunto fijo de interfaces de bajo nivel, y se genera correspondencia con la capa de aplicacion con interfaces de manejo de entradas y salidas de paquetes (*Send/Receive*) y interfaces de control de potencia *(SplitControl)*. Capas superiores llaman a *start()* y *stop()*, comandos de la interfaz *SplitControl* con el fin de activar/desactivar el funcionamiento del protocolo MAC. Al llamar *stop()* se apaga todo el protocolo MAC y pone la radio en un estado inactivo de energía. Al llamar *start()* cambia el estado del protocolo MAC a ON, alternando entre los distintos estados de energía de la radio en función de la especificación del protocolo. Las interfaces *Send* y *Receive* se utilizan para transmitir los paquetes entre los componentes del nivel superior y los subyacentes stacks de la radio. Como MacC está expuesta a niveles superiores como un único componente, con un conjunto fijo de interfaces, los desarrolladores pueden intercambiar diferentes protocolos MAC de una forma muy sencilla.

Algunos protocolos MAC necesitan exportar un pequeño número de interfaces específicas de la capa MAC por ejemplo, para permitir a las aplicaciones el control del largo de los intervalos de sueño del protocolo B-MAC o fijar el largo de la trama de un protocolo basado en TDMA. Estas interfaces están reunidas en el componente MacControlC, este proporciona a los desarrolladores de la capa de aplicacion un único lugar para el control de las interfaces de un protocolo especifico. Al igual que MacC, cada una de las implementaciones de los protocolos MAC ofrece diferentes definiciones de MacControlC. Aunque el conjunto de interfaces exportadas por cada MacControlC varía de protocolo a protocolo su composición interior nunca es expuesto a la aplicación.

3.2.4. Componentes

Aunque MLA presenta una interfaz simple para las capas superiores, diferentes protocolos MAC a menudo presentan una compleja variedad de características. La implementación de estas características desde cero para cada protocolo puede ser difícil y requerir mucho tiempo. MLA reduce este esfuerzo por identificar las

	Channel Polling	Scheduled Contention	TDMA
Channel Poller	x	x	
LPL Listener	x	x	
Preamble Sender	x		
Time Synchronization		x	x
Slot Handlers			x
Low Level Dispatcher			x
AsyncIOAdapter	x	x	x
Alarm	x	x	x
Local Time		x	x
Radio Core	x	x	x

Cuadro 3.1: Implementaciones en hadware por MLA. Imagen obtenida de [18]

características que muchos protocolos MAC comparten, y encapsulándolo dentro de un conjunto optimizado, de componentes reutilizables. La Tabla 3.1 enumera los diferentes componentes tanto los que son independiente del hardware como los que dependen de estos, que se encuentran definidos en el MLA. SCP combina las caracteristicas marcadas en la tabla 3.1 sobre Channel Polling y Scheduled Contention.

3.3. Protocolo SCP implementado en MLA

El funcionamiento general del protocolo SCP se describe en el capítulo anterior. En síntesis, es una variante de Low Power Listening en el que la verificación de canal se hace de manera sincronizada para disminuir los tiempos de preámbulo.

La implementación MLA de este protocolo usa los mismos componentes que B-MAC, pero introduce nuevos elementos para el funcionamiento de la sincronización.

Para lograr la sincronización, se hace piggibacking agregando a cada mensaje transmitido, una estampa de tiempo indicando el momento del próximo chequeo de canal. De esta forma el mote recpetor ajusta su reloj interno para hacer coincidir el momento de realización del chequeo de canal con el del mote transmisor del mensaje.

Por otro lado, cada mote tiene, además del reloj que indica el momento de realizar el chequeo de canal, otro reloj indicando el momento para empezar transmitir el preámbulo en caso de enviar un mensaje. Este reloj está ajustado con un pequeño adelanto respecto al primero, para asegurar que se esté enviando preámbulo cuando el receptor haga su chequeo de canal.

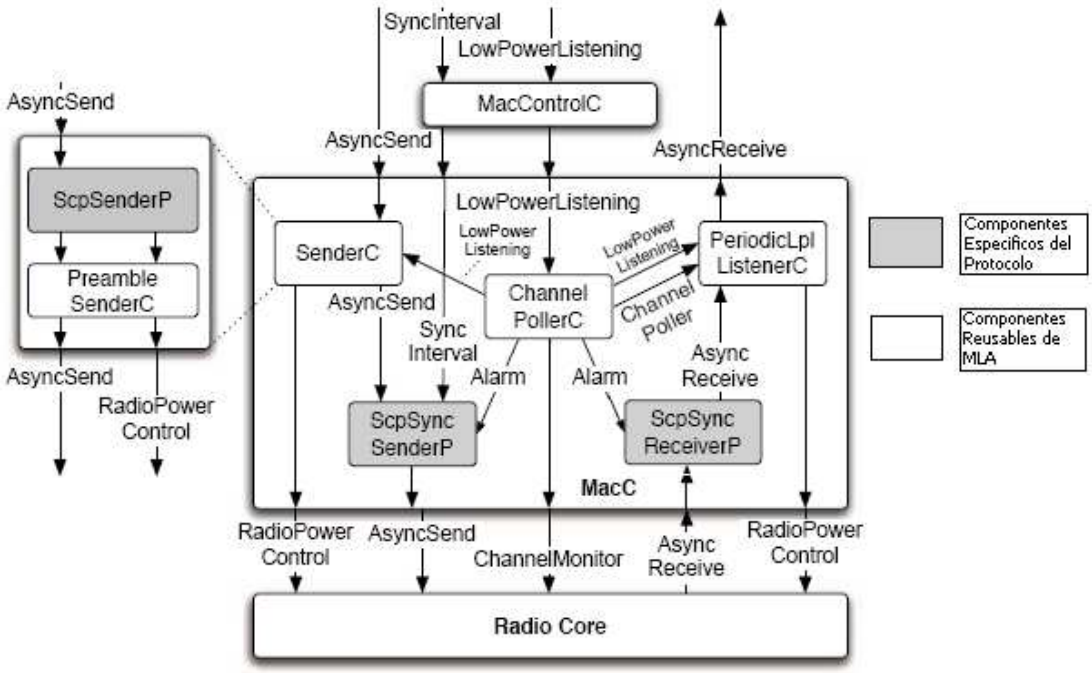

Figura 3.4: Composición del Protocolo SCP

Como se muestra en la Figura 3.4, el componente *ScpSyncSenderP* añade a cada paquete saliente un contador de 2-byte que representa el tiempo restante en la alarma interna de *ChannelPollerC* (es decir, cuánto tiempo falta para que el nodo realice su próximo chequeo de canal). *ScpSyncReceiverP* lee estas estampas de tiempo de los paquetes y ajusta la alarma local del *ChannelPollerC* en consecuencia. Este ajuste asegura que todos los nodos despierten para el chequeo CCA simultáneamente . El *ScpSyncSenderP* periódicamente envía un paquete de sincronización, si no se envió ningún paquete de datos en un intervalo de tiempo dado. La implementación podrá customizar este intervalo utilizando la interfaz *SyncInterval* exportados a través del componente *MacControlC*.

SCP también introduce un componente *ScpSenderP* para el buffer de paquetes salientes. En contraste con *BmacSenderP* y *XmacSenderP*, que comenzará el envío de preámbulos tan pronto como sea de rápido enviar un paquete, *ScpSenderP* espera hasta justo antes de su próximo chequeo CCA (es decir, justo antes de lo que tiene previsto que despierten los otros nodos). Este cambio permite a los nodos de envío y recepción sincronizar la actividad de su radio, y asi lograr utilizar un preámbulo corto (9 ms en radios CC2420). *ScpSenderP* también lleva a cabo un entubamiento del paquete, optimización que permite que la aplicación envíe múltiples paquetes dentro de un intervalo de LPL. Otra diferencia fundamental es que *ScpSenderP* construye su preámbulo desde paquetes de preámbulo explícitos que contengan 1s y 0s alternadamente, en lugar de reutilizar el buffer de paquete

de datos directamente. Los paquetes de preámbulo llegan tan rápidamente que *ScpSyncReceiverP* no tiene la opción de hacer cualquier transformación en ellos, ya que este rebasara su manejador de interrupción. Este cambio permite a *ScpSyncReceiverP* identificar de inmediato y desechar los paquetes de preámbulo con un mínimo de procesamiento.

Tener en cuenta que los paquetes de preámbulo no contienen datos útiles para la capas superiores: estos existen simplemente para ocupar el canal, mientras que el nodo receptor está a la escucha. A diferencia de otros protocolos, SCP contiene una pequeña porción de código dependiente de la radio. Este código enumera un puñado de constantes específicas de la radio, como la cantidad de tiempo que se tarda en despertar la radio. Es imposible eliminar completamente esta porción de dependencia de la radio, ya que SCP logra su alta eficiencia energética mediante un ajuste del largo de los preámbulos de acuerdo con estas propiedades especificas de la radio. Sin embargo, la porción especifica de la CC2420 del SCP contiene sólo 8 líneas que declaran constantes derivadas de datos de experimentación. El esfuerzo necesario para agregar soporte para nuevos radios debería ser mínimo. Mientras que la aplicación en MLA se implementa la mayoría de las características del protocolo SCP, aún no se ha llevado a cabo evitar su sobre escucha.

Capítulo 4

Verificación del funcionamiento.

4.1. Itroducción

La implementación MLA del protocolo SCP se compiló en los motes para probar su funcionamiento. En un principio se pensó que el funcionamiento era correcto, pero una vez que se empezó a hacer un análisis más detallado, observando el consumo de los motes con un osciloscopio, se vio que el funcionamiento no era el esperado y fue necesario hacer un conjunto de arreglos y ajustes.

En este capítulo se explican los problemas encontrados, su solución, y los ajustes hechos posteriormente.

4.2. Compilación del protocolo

Para probar la implementación MLA del protocolo SCP, se tomó un programa llamado TestSCP que venía junto con la implementación para tal efecto, se le introdujeron pequeñas modificaciones y se compiló en los motes.

A medida que se fueron probando diferentes características se encontraron algunos errores y problemas que se fueron arreglando. Esto motivó un estudio bastante detallado del código para solucionar los problemas. El estudio del código se ve en el próximo capítulo.

La funcionalidad del programa TestSCP original es muy básica. Ésta tiene dos modos de funcionamiento. En uno de los modos, el mote genera periódicamente un mensaje de datos y lo trasmite por la radio mediante SCP, mientras que en el otro modo el mote no genera mensajes de datos para enviar. En ambos modos, el mote puede enviar mensajes de sincronismo como lo requiere el protocolo. Además, en ambos modos el mote puede recibir cualquier mensaje enviado por otro mote cercano mediante SCP. El programa original tiene en el código una parte donde

se pretende transmitir por el puerto USB los mensajes recibidos por SCP, pero tiene un pequeño problema (bug) que se arregló. En ambos modos se prenden y apagan los leds mostrando cómo se transmiten los mensajes.

En una parte del código original se fijan tres parámetros que, al modificarlos, permiten estudiar el comportamiento del protodolo SCP. Estos parámetros son el período de sincronización (*Sync Interval*), el período de muestreo del canal (T), y el período de generación mensajes de los motes programados en modo de enviar mensajes (t_{msg}).

En primer lugar se comprobó que los leds se prendían como estaba previsto en el programa TestSCP mostrando aparentemente que los mensajes se transmitían correctamente.

Como forma de medir los tiempos en que la radio permanece prendida, se procedió a medir el consumo como se explica a contimuación.

4.3. Medida de tiempos de encendido de la radio con osciloscopio

Para comprobar el funcionamiento del protocolo, además de comprobarse que los mensajes se transmitieran correctamente, se midió el consumo de los motes con un osciloscopio para determinar los momentos de prendido y apagado de la radio y analizar el ciclo de trabajo. El procedimiento de medición se describe con más detalle en 6.2.

Las figuras 4.1 y 4.2 son dos impresiones las medidas obtenidas con el osciloscopio. Se observan claramente dos niveles de consumo, un pico de de 20 mA aproximadamente que corresponde a cuando la radio está prendida, rodeado de niveles de consumo mucho menores que corresponde a cuando la radio está apagada.

Haciendo esta medida simultaneamente en dos motes y utilizando ambos canales del osciloscopio, se estudió la sincronización y la transmisión de mensajes del protocolo SCP.

4.4. Primeras observaciones

Los resultados de las primeras observaciones no fueron las esperadas.

Repasando a modo de resumen, lo esperado debía incluir:

1. Las radios de todos los nodos debían prenderse al mismo tiempo o con un pequeño desfasaje.

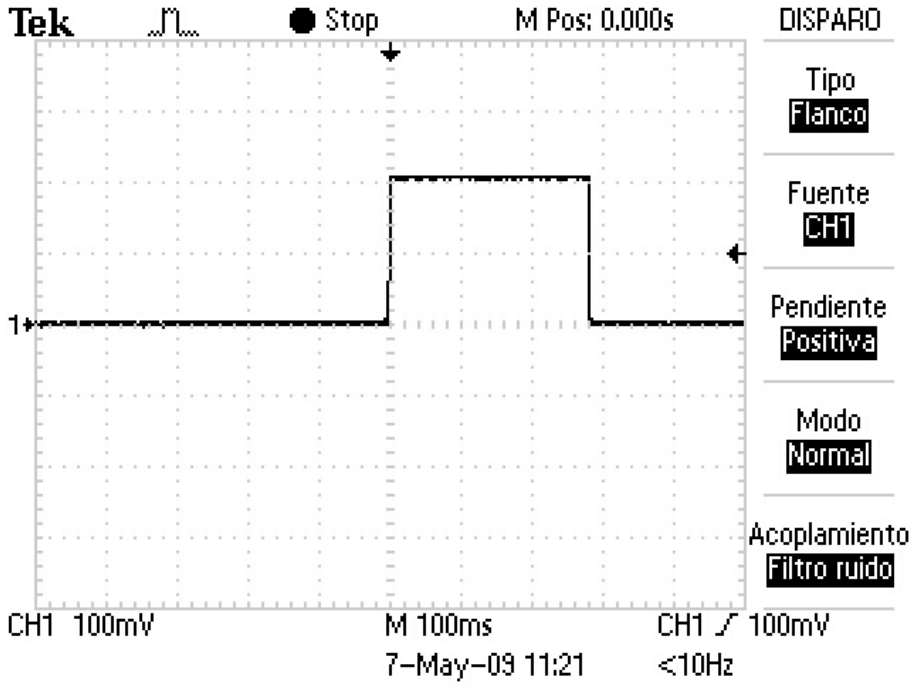

Figura 4.1: Impresión en osciloscopio del momento de la verificación del canal. Se observa un consumo debido debido al prendido de la radio de 280 milisegundos aproximadamente.
Escala eje horizontal: 100ms/div
Escala eje vertical: 100mV/div (correspondiente a 10mA/div)

2. Si no había intercambio de mensajes, las radios se debían prender durante un tiempo muy corto.

3. Si había intercambio de mensajes, tanto en el emisor como en el receptor, las radios debían permanecer prendidas durante el tiempo necesario para transmitir los mensajes y apagarse casi inmediatamente después.

Sin embargo, lo observado incluyó los siguientes problemas.

En primer lugar, cuando se transmitía un mensaje, la radio se prendía duarnte 100 milisegundos aproximadamente ($t_{TX} = 100ms$, figura 4.2) y cuando no recibia mensajes, la radio se mantenía prendida durante 300 milisegundos aproximadamente ($t_{Sleep} = 300ms$, figura 4.1). Esto significa que el tiempo en que la radio se mantenía prendida para hacer chequeo de canal, que debía ser muy corto, era mayor aun al tiempo necesario para transmitir un mensaje.

Por otro lado, erráticamente los motes dejaban su radio encendida durante varios períodos enteros, en vez de hacer el ciclo esperado de prendido y apagado de radio, como se muestra en la figura 4.3. Este problema causó confusión porque cuando no se manifestaba parecía no existir, por este motivo fue ignorado durante un

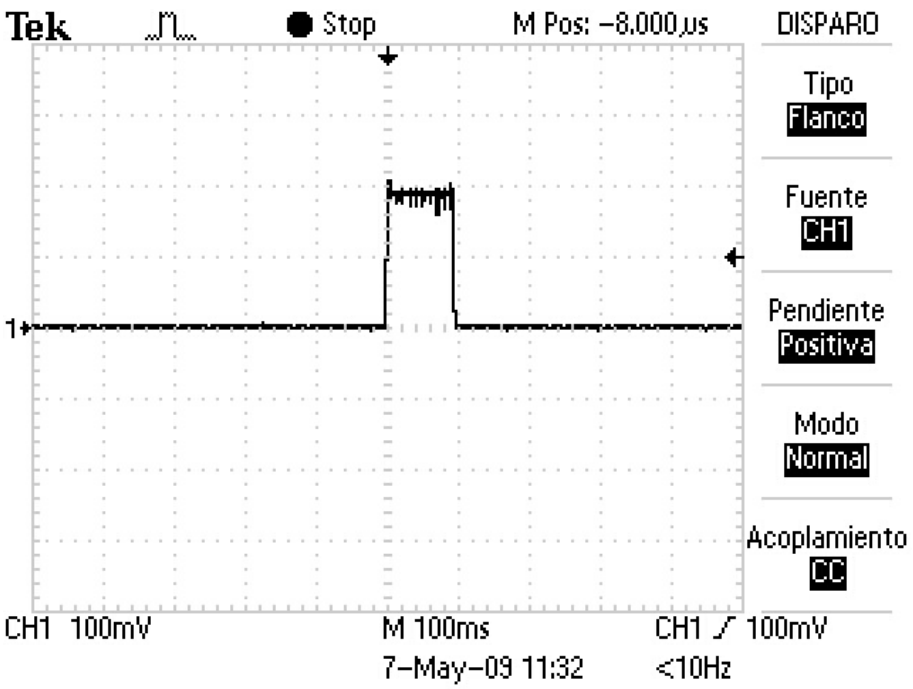

Figura 4.2: Impresión en el osciloscopio del momento de transmisión de un mensaje. Se observa un consumo debido al prendido de la radio de 100 milisegundos aproximadamente.
Escala eje horizontal: 100ms/div
Escala eje vertical: 100mV/div (correspondiente a 10mA/div)

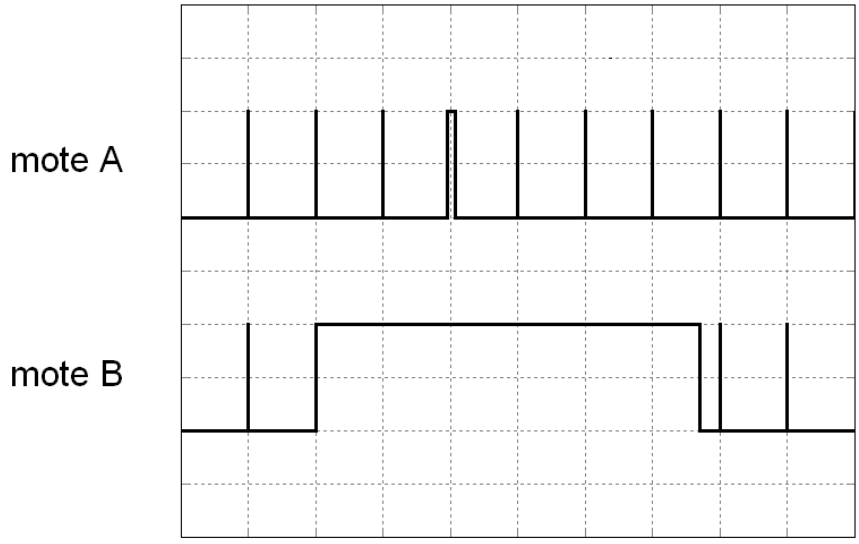

Figura 4.3: Esquema del comportamiento errático observado.

tiempo. El resto de los problemas fueron observados cuando este problema no se manifestaba, o se manifestaba en uno solo de los motes.

Cuando se recibía un mensaje, la radio se mantenía prendida durante dos períodos enteros. Por ejemplo, si se programaba el mote para hacer chequeo de canal cada 1 segundo, la radio del mote se mantenía prendida por lo menos 2 segundos cada vez que recibía un mensaje ($t_{RX} > 2 \cdot T$), como se ve en la figura 4.4.

Los motes se desincronizaban por completo y no se volvían a sincronizar. Cada vez que se producía una transmisión correcta de datos, el mote receptor del mensaje siempre se atrasaba respecto del emisor.

Por la propia naturaleza de los problemas, es claro que no fueron diagnosticados todos al mimo tiempo, ya que es imposible observarlos todos a la vez. La aparición de estos problemas motivó un estudio detallado del código implementado como se verá en el siguiente capítulo.

4.5. Solución a los problemas

4.5.1. Tiempo de chequeo de canal.

Como se describe en el capítulo siguiente, la duración del chequeo del canal está dada por una constante llamada `CcaCheckLength` que a nuestro entender estaba erroneamente multiplicada por 32. Esto determinaba que la radio estubiera

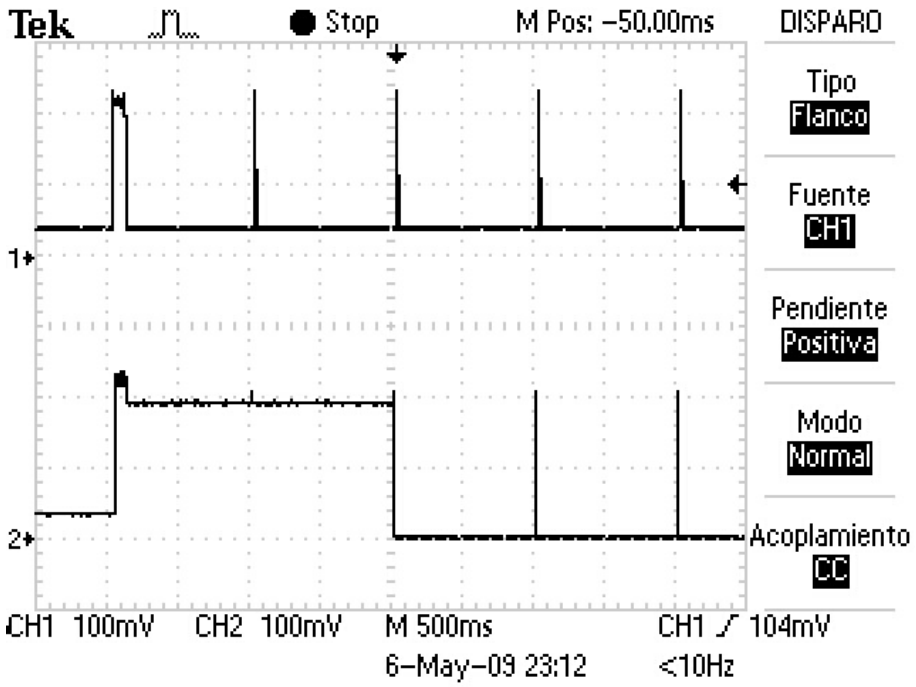

Figura 4.4: Impresión en el osciloscopio de la transmisión de un mensaje. Se observa un consumo de radio prendida de dos períodos de duración por parte del receptor.
Escala eje horizontal: 100ms/div
Escala eje vertical: 100mV/div (correspondiente a 10mA/div)

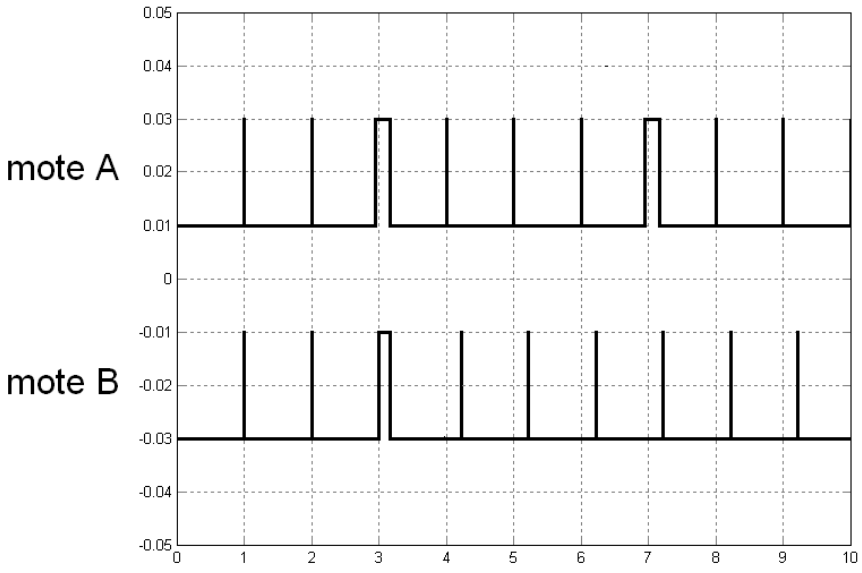

Figura 4.5: Esquema del comoprtamiento al momento de la desincronización.

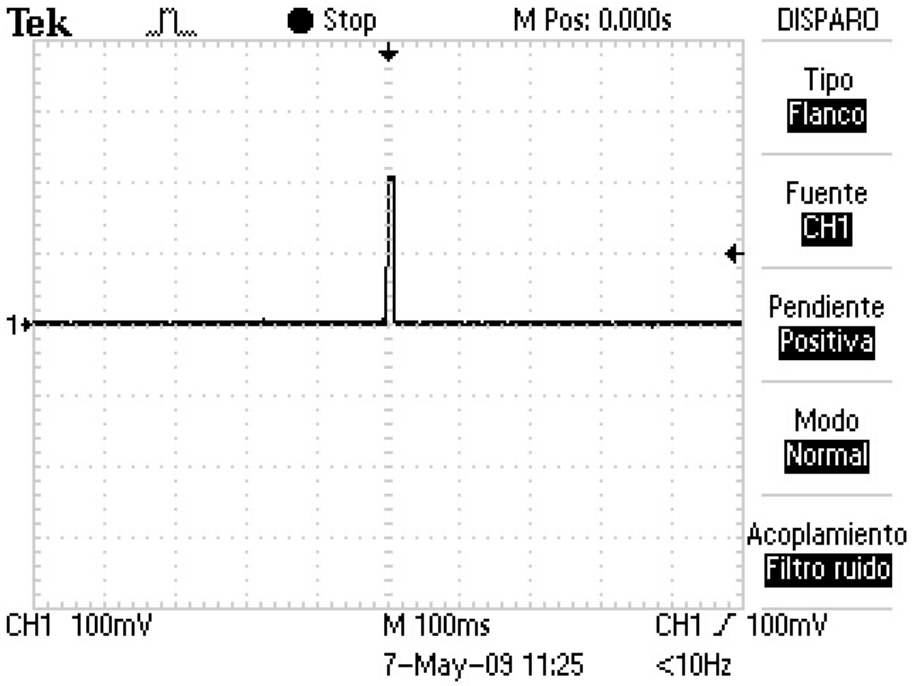

Figura 4.6: Impresión en osciloscopio del momento de la verificación del canal. Se observa un consumo debido debido al prendido de la radio de 10 milisegundos aproximadamente.
Escala eje horizontal: 100ms/div
Escala eje vertical: 100mV/div (correspondiente a 10mA/div)

prendida durante 300 milisegundos aproximadamante cada vez que se hacía una verificación del canal ($t_{Listen} = 300ms$). Al quitarle dicha multiplicación se obtuvo el comportamiento esperado lográndose que el tiempo de escucha del canal se redujera a 10 milisegundos ($t_{Listen} = 10ms$) como se muestra en las impresiones del osciloscopio mostrados en las figuras 4.6 y 4.7.

4.5.2. Radio encendida por dos períodos

Otro de los problemas fue que cuando un mote detectaba actividad en el canal, no apagaba la radio cuando terminaba de recibir mensajes, como era de esperar, si no que permanecía prendida dos períodos T completos, por lo que era $t_{RX} \approx 2 \cdot T$.

En este caso, lo que ocurría eran dos problemas. Por un lado no se encontró en el código ninguna órden de apagado de la radio al terminar de recibir mensajes, por lo tanto la radio permanecería prendida al menos hasta la siguiente verificación del canal.

Por otro lado, cada vez que se recibía un mensaje se incrementava una variable llamada `packetCount`, y al realizar la siguiente verificación de canal el hecho de

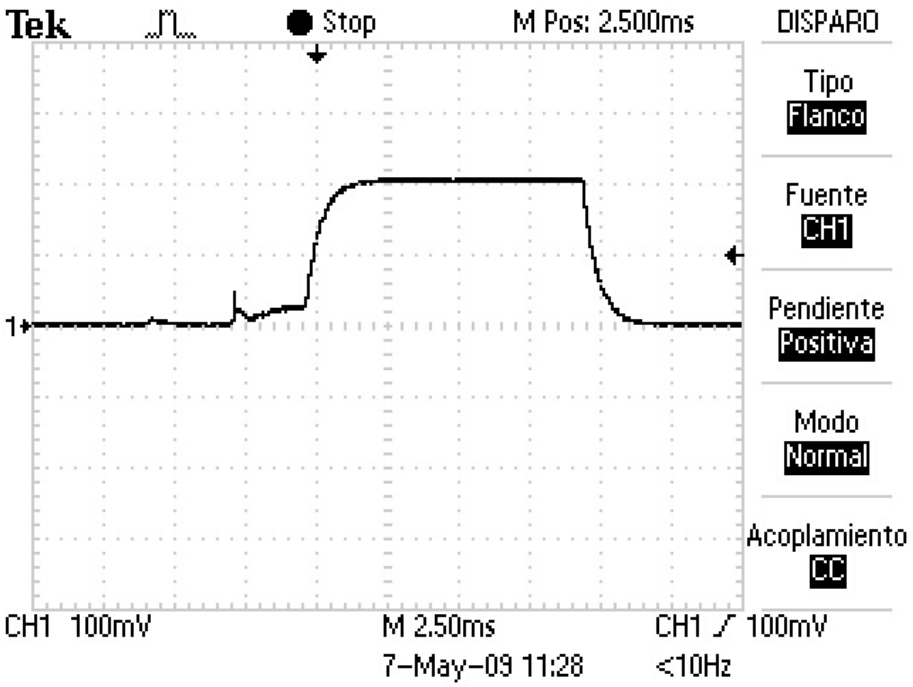

Figura 4.7: Impresión en osciloscopio del momento de la verificación del canal, como en la figura 4.6 pero con otra escala de tiempos
Escala eje horizontal: 2,5ms/div
Escala eje vertical: 100mV/div (correspondiente a 10mA/div)

que esa varable fuera mayor que cero implicaba que no se apagara la radio otra vez, por lo que continuaba la radio prendida hasta la siguiente verificación de canal. En este caso la variable se restablecía a cero por lo que en la siguiente verificación de canal la radio finalmente se apagaba. Los detalles de lo que ocurría están explicados en el siguiente capítulo.

Primero se quitó toda la parte de código relacionado con `packetCount` y se logró que el mote permaneciera con la radio prendida durante un ciclo en vez de dos, logrando por lo tanto que fuera $t_{RX} \approx T$.

Incluir que el mote apague la radio cuando termine de recibir mensajes requería una reescritura completa de parte del código, el mote debía ser capaz de determinar cuándo habían más mensajes pendientes para recibir y cuándo se había recibido el último mensaje.

De todos modos, asumiendo un caso particular en que los motes siempre recibieran un mensaje por vez, se agregó que el mote apague inmedatamente la radio luego de recibir un mensaje, logrando disminuir significativamente t_{RX}.

4.5.3. Radio encendida por varios períodos

Intentando diagnosticar este problema fue que se diagnosticó complemtamente el problema anterior, y se realizó la solución parcial descrita.

Pensando que el problema debía estar en las verificaciones que hacía el mote luego de terminado el tiempo dado por `CcaCheckLength`, se hizo un estudio minucioso de estas verificaciones no encontrándose nada que se pudiera relacionar con el problema.

Luego se realizo un debug del código, que cosistió en trabajar con una implementación básica del protocolo e ir agragando complejidad, mientras íbamos comentando parte del código, buscando que la radio se apagara indefectiblemente cuando esto debía ser asi, según lo analizado en el código.

Finalmente, y de manera casual, se descubrió que cubriendo uno de los motes con una mano, el problema se solucionaba. Se determinó que estábamos haciendo las pruebas con los motes demasiado cerca y que se producía interferencia causando que `EnergyIndicator.isReceiving` diera falsos positivos.

Posteriormente se observó que, separando los motes un par de metros, o cubriendo uno con la mano, este problema dejaba de manifestarse por completo.

4.5.4. Pérdida de Sincronismo.

En el osciloscopio se observó que, cada vez que se intercambiaba un mensaje con éxito, el receptor del mensaje se atrasaba con respecto al emisor, causando

que no se lograra nunca más intercambiar mensajes, si ninguno de los problemas anteriores se manifestaba.

Como se detalla en la sección 5.3, cada mote tiene un reloj que indica cuándo realizar el próximo chequeo de canal, y otro reloj que indica cuándo comenzar a transmitir preámbulo en caso de enviar un mensaje. Cada vez que un mote envía un mensaje, agrega una estampa de tiempo indicando cuándo va a realizar el siguiente chequeo de canal. A su vez, cuando el mote receptor recibe el mensaje, ajusta los dos relojes de acuerdo a dicha información.

La solución a este problema fue modificar la interpretación que el receptor hace de la estampa de tiempo. En vez de reiniciar su reloj con la información recibida sin ningún cambio, se agregó que hiciera una resta para que reiniciara sus relojes con un valor menor.

4.5.4.1. Ajustes del preámbulo y del largo del período

También se realizaron ajustes sobre el preámbulo dado que esto afectaba la sincronización.

Se vio que el preámbulo comenzaba a envierse unos 10ms antes del tiempo estimado de chequeo de canal, y luego duraba unos 90ms, cuando lo más razonable es que el largo del preámbulo sea el doble del adelanto sobre el comienzo del chequeo de canal, más la duración del chequeo de canal. Por ejemplo, si el chequeo de canal dura 10ms ($t_{Listen} = 10ms$) y el preámbulo se comienza a transmitir 20ms antes del comienzo del chequeo de canal, lo razonable es que el preámbulo dure 50ms.

Al modificar el preámbulo se observó que esto afecta la sincornización, teniendo que modificarse la resta que hacen sobre la estampa de tiempo, los motes que reciben mensajes, para ajustar sus mensjes.

Teniendo todo esto en cuenta, se mantuvo el largo del preámbulo en 90ms pero se lo adelantó para que comenzara unos 40ms antes del chequeo de canal. La resta que hacen los receptores sobre la estampa de tiempo quedó fijada en 50ms para el reloj que determina los chequeos de canal y 90ms para el reloj que determina el comienzo del prámbulo.

Con estos valores, el tiempo de transmisión se mantubo en unos 100ms ($t_{TX} = 100ms$) y el tiempo de recepción es de unos 60ms ($t_{RX} = 60ms$).

Con estos parámetros para el preámbulo se puede tomar un período de escucha T largo. Finalmente se fijó en 10 segundos.

Capítulo 5

Detalles del código y funcionamiento de la implementación

5.1. Introducción

Los problemas discutidos en el capítulo anterior motivaron un estudio con más detalle del funcionamiento y el código NesC de la implementación. El componente principal del código de la implementación MLA es una *configuración* denominada `MacC`. Este componente es básicamente el mismo que se describe en el capítulo 3.3 pero con algunas diferencias debidas sobre todo al funcionamiento particular de TinyOS.

Un diagrama de `MacC` generado a partir del propio código NesC se muestra en las figuras 5.1 y 5.2. En el diagrama se ven los mismos componentes vistos en 3.3 y algunos otros más.

En resto de las secciones se mostrará en detalle algunos puntos particulares del diseño que permitieron entender y arreglar los problemas vistos en el capítulo anterior. Sobre cada punto se mostrarán fragmentos del código NesC y se harán comentarios del mismo.

5.2. Chequeo periódico del canal

Como varios de los problemas observados en el capítulo anterior estaban relacionados con el apagado de la radio, se analizó cómo se controlaba la misma.

Se determinó que, inmediatamente después de prender la radio, el mote entraba en una rutina de chequeo de canal cuya duración estaba determinada por la constante

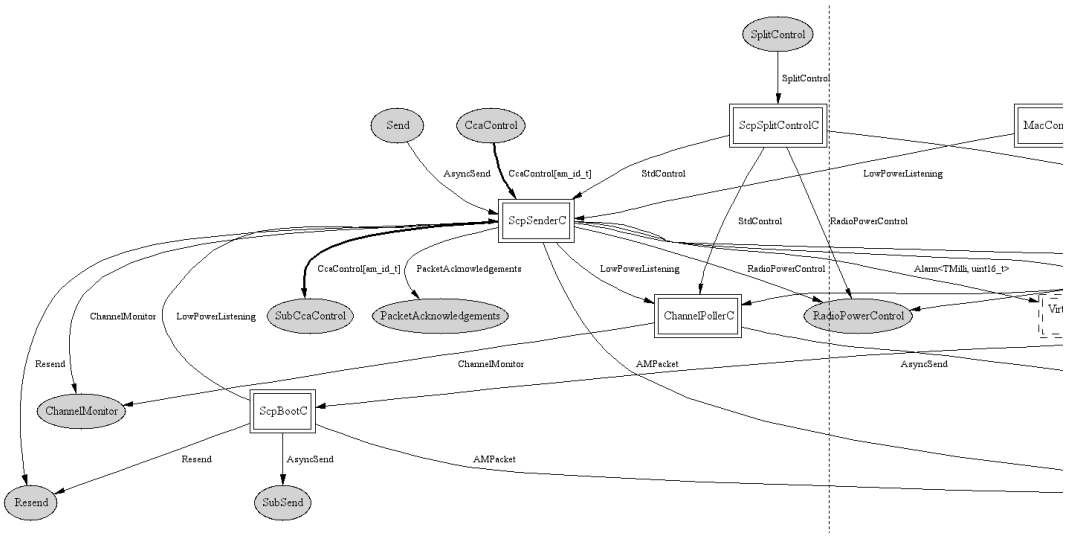

Figura 5.1: Diagrama del componente MacC del código MLA de SCP - parte izquierda. Los rectángulos indican subcomponentes y las elipses indican interfaces al exterior.

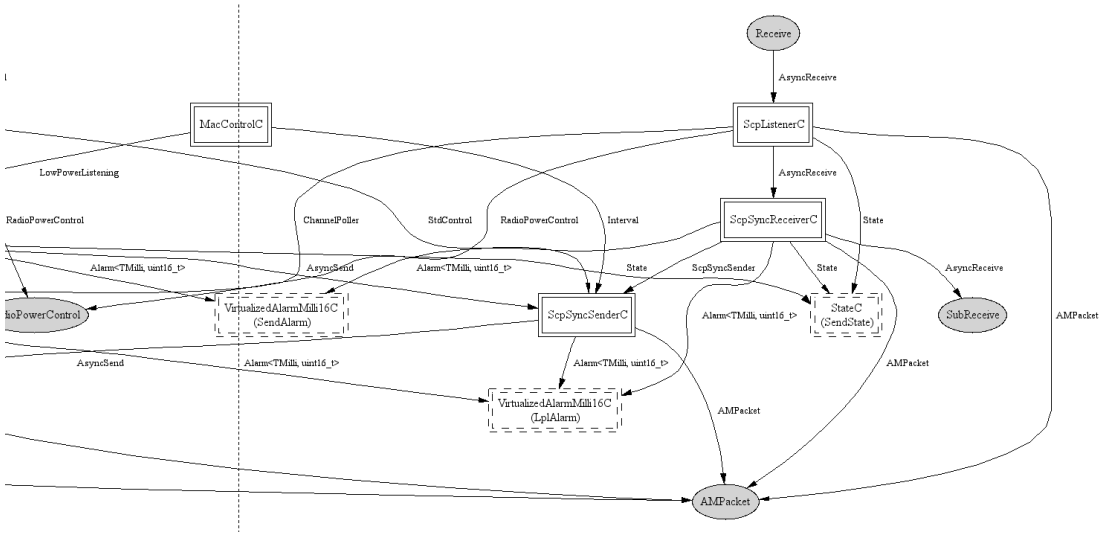

Figura 5.2: Diagrama del componente MacC del código MLA de SCP - parte derecha. Los rectángulos indican subcomponentes y las elipses indican interfaces al exterior.

`CcaCheckLength`, y durante el cual, se llamaba repetidas veces a dos interfaces de TinyOS llamadas, `PacketIndicator.isReceiving` y `EnergyIndicator.isReceiving`. La primera indicaba si la radio estaba recibiendo un mensaje reconocible, y la segunda indicaba si se detectaba determinado nivel de potencia en el canal. En el caso que indicara que estaba recibiendo un mensaje, se consideraba que el canal estaba ocupado y se abandonaba la rutina de chequeo de canal. En cambio la otra interfaz indicaba si se detectaba un nivel de potencia en el canal. En este caso para considerar que el canal estaba ocupado, se esperaba a que esta interfaz fuera varias veces positivo.

Luego de abandonado la rutina de chequeo de canal, el mote ejecutaba otras líneas de comando buscando establecer si se debía apagar la radio o no. En particular, había un contador llamado `PacketCount` que determinaba que al recibir un mensaje, la radio permanecía prendida durante dos períodos completos.

El chequeo del canal es periódico y el inicio de cada chequeo lo indica el componente `LplAlarm` (figura 5.2, parte inferior centro). Primero se hará un seguimiento de la secuencia de eventos que genera el disparo de esta alarma, esta secuencia se dividirá en tres etapas. En la tercer etapa se analiza de qué depende que se apague o no la radio. Luego se analizará cómo está determinado el tiempo que dura un chequeo de canal.

5.2.1. Desde el disparo de `LplAlarm` hasta el chequeo de canal propiamente dicho.

La señal que envía `LplAlarm` la reciben todos los componentes conectados a él que son `ChannelPollerC`, `ScpCyncSenderP` y `ScpSyncReceiverC` (figuras 5.1 y 5.2).

`ScpCyncSenderC` y `ScpSyncReceiverC` no hacen nada con esa señal por que usan la interfaz por la que reciben esa señal para otra cosa como se verá más adelante.

El componente que procesa la señal de disparo de `LplAlarm` es `ChannelPollerC`:

```
ChannelPoller:
async event void Alarm.fired()
{
// If the channel poller is active
if(running_)
{
call Alarm.start(ms_);
// Restart the timer
call State.forceState(S_CHECKING);
// Move into the busy state
```

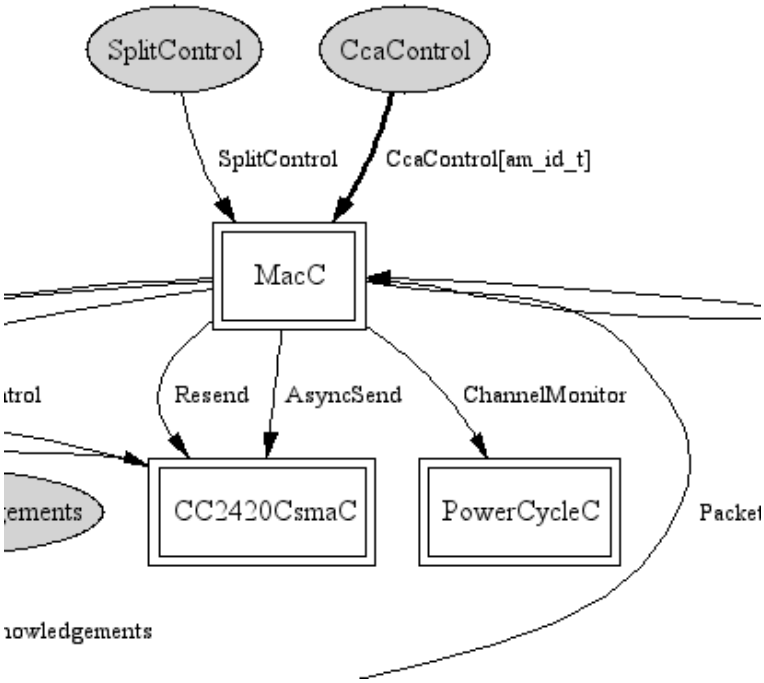

Figura 5.3: Diagrama que muestra la conexión entre `MacC` y `PowerCycleC` a través de la interfaz `ChannelMonitor`.

```
call ChannelMonitor.check();
 // Do a CCA check
}
}
```

`ChannelPollerC` básicamente ordena el inicio del chequeo a través de la interfaz `ChannelMonitor`: El inicio del chequeo de canal está sujeto a la condición de que la variable `running_` sea `TRUE`. Esta variable vale `TRUE` durante el funcionamiento normal del protocolo.

Además, se reinicia la alarma para la siguiente vez, y se guarda una variable de estado para que valga '`S_CHECKING`'. Esta variable de estado indica que se inicia un chequeo de canal.

La interfaz `ChannelMonitor` por donde se transmite la orden de inicio del chequeo, sale para afuera de `MacC` (figura 5.1, extremo izquierdo) y está conectada al componente `PowerCycleC` como se muestra en la figura 5.3.

`PowerCycleC` recibe el comando mencionado y es el componente que hace todo el chequeo de canal:

```
PowerCycle:
async command void ChannelMonitor.check() {
  if(call SplitControlState.getState() == S_TURNING_OFF) {
    signal ChannelMonitor.error();
  }
  else if(call SplitControlState.getState() == S_OFF) {
    call SplitControlState.forceState(S_TURNING_ON);
  }
  call RadioPowerControl.start();
}
event void RadioPowerControl.startDone(error_t error) {

  if(finishSplitControlRequests()) {
    return;

  } else if(isDutyCycling()) {
    post getCca();
  }
}
task void getCca() {
  uint8_t detects = 0;
  local_time16_t startAt, endAt, length, now;
  uint16_t count = 0;
  length.mticks = 0;
  length.sticks = ccaCheckLength;
  startAt = call Time.getNow();
  endAt = call Time.add(&startAt, &length);
  while(TRUE) {
    count++;
    if(count % 32 == 0) {
      now = call Time.getNow();
      if(!call Time.lessThan(&now, &endAt)) {
        break;
      }
    }

    if(call PacketIndicator.isReceiving()) {
      detects = MIN_SAMPLES_BEFORE_DETECT + 1;
      break;
    }
     if(call EnergyIndicator.isReceiving()) {
       detects++;
       if(detects > MIN_SAMPLES_BEFORE_DETECT) {
```

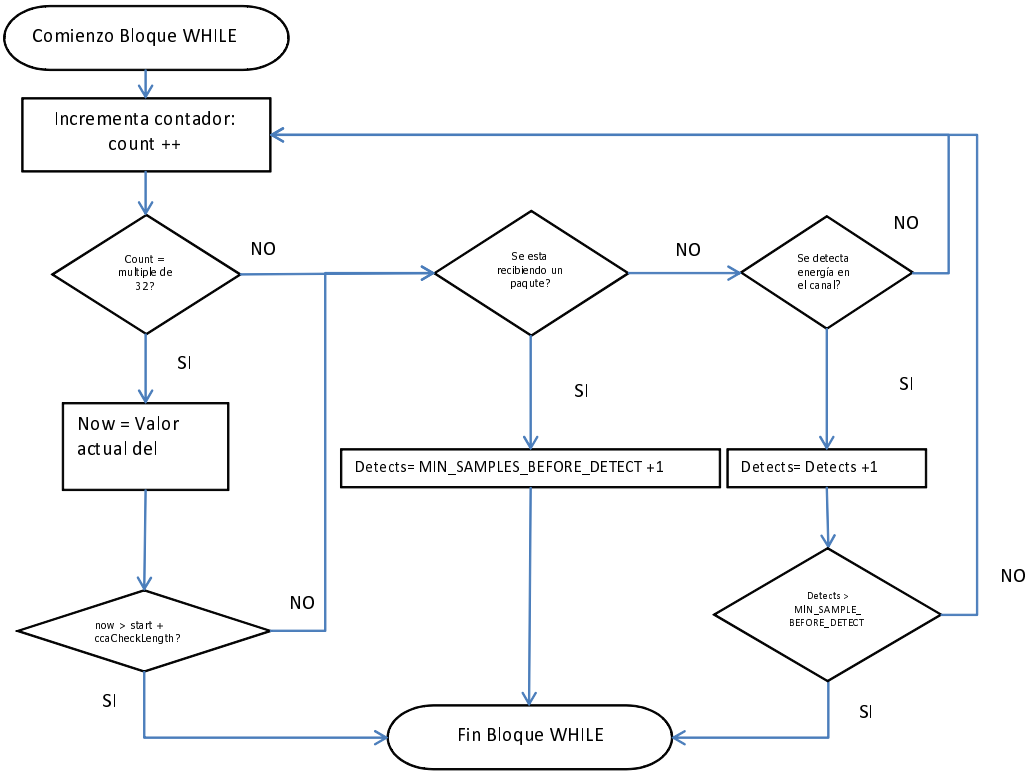

Figura 5.4: Diagrama de Flujo-Ciclo WHILE

```
          break;
        }
        // Leave the radio on for upper layers to perform some transaction
      }
    }

    if(detects > MIN_SAMPLES_BEFORE_DETECT) {
      signal ChannelMonitor.busy();
    } else {
      signal ChannelMonitor.free();
    }

  }
```

Como se muestra en el código, `PowerCycleC` primero ordena prender la radio (las variables de estado que se manejan indican el estado de la radio y se pueden ignorar).

`PowerCycleC` debe esperar recibir la señal `RadioPowerControl.startDone(error_t`

`error)` para saber que la radio se prendió correctamente. Como se muestra más abajo en el código, cuando `PowerCycleC` recibe esta señal, se inicia la tarea `getCca()` que se muestra más abajo en el código y corresponde al chequeo de canal propiamente dicho.

5.2.2. Chequeo de canal

El análisis de esta parte del código resultó fundamental en el diagnóstico de algunos problemas.

La tarea `getCca()` tiene una parte al principio de establecimiento de variables, luego un bloque `while` que es donde se hace el chequeo de canal, y una parte al final que es donde se señala el resultado del chequeo.

Si se observa en la parte final, se ve que la condición para considerar que el canal está libre u ocupado es que la variable `detects` sea menor o mayor respectivamente a la constante `MIN_SAMPLES_BEFORE_DETECT`. La variable `detects` se modifica dentro del bloque `while`.

Dentro del bloque `while(ver 5.4)` se ve que el chequeo de canal se hace mediante las órdenes `call PacketIndicator.isReceiving()` y `call EnergyIndicator.isReceiving()`. Si la primera retorna `TRUE`, se establece `detects` mayor que `MIN_SAMPLES_BEFORE_DETECT` para marcar que el canal está ocupado y se abandona el bloque `while` inmediatamente. Si `call EnergyIndicator.isReceiving()` retorna `TRUE`, se incrementa `detects`, y solo se abandona el bloque `while` si `detects` supera a `MIN_SAMPLES_BEFORE_DETECT`.

La tercer condición para abandonar el bloque `while` es por tiempo, si `detect` nunca llega a ser mayor que `MIN_SAMPLES_BEFORE_DETECT`.

El tiempo máximo para abandonar el bloque `while` en caso de que no se detecte actividad, se establece en la primera parte del código de la tarea `getCca()`, y depende en última instancia de la variable `ccaCheckLength`. Esta variable se observó que presentaba valores muy altos y que determinaba que el chequeo de canal fuera demasiado largo. Esto se analiza en la sección 5.2.4.

5.2.3. Luego del chequeo y apagado de la radio

Luego de realizado el chequeo, se señala `ChannelMonitor.busy()` o `ChannelMonitor.free()` si se detectó o no actividad en el canal respectivamente.

Esta señal vuelve por la interfaz `ChannelMonitor` a `MacC` (figura 5.3).

Dentro de `MacC`, la reciben los componentes `ScpSenderC`, que ignora la señal, y `ChannelPollerC`, que es el mismo que ordena el inicio del chequeo de canal, como se vio más arriba. Este componente básicamente retransmite la señal si se cumple una determinada condición, o ignora la señal si no se cumple:

```
ChannelPoller:
async event void ChannelMonitor.busy()
{

    if(call State.isIdle())
    return;
    // If the check isn't for us, ignore it
    call State.toIdle();
    signal ChannelPoller.activityDetected(TRUE);
    // Move into the idle state and signal that the channel is busy
    }

async event void ChannelMonitor.free()
{

    if(call State.isIdle())
    return;
    // If the check isn't for us, ignore it
    call State.toIdle();
    signal ChannelPoller.activityDetected(FALSE);
    // Move into the idle state and signal that the channel is free

}
```

La condición tiene que ver con la variable de estado que se vio más arriba que guarda `ChannelPollerC`, y normalmente se cumple. Por lo tanto la señal se retransmite casi exactamente igual por la interfaz `ChannelPoller`. La única diferencia es que la información de si el canal está libre o no está dada por un parámetro, que vale `TRUE` cuando hay actividad en el canal, y `FALSE` cuando está libre.

La interfaz está conectada al componente `ScpListenerC` por lo tanto este componente recibe la señal.

`ScpListenerC` es una configuración cuyo diagrama se muestra en la figura 5.5.

Dentro de `ScpListenerC` la señal la recibe el componente `ScpListenerFilterP`:

```
ScpListenerFilterP:
async event void SubChannelPoller.activityDetected(bool detected)
{
uint16_t lastCount;
atomic
{
lastCount = packetCount;
```

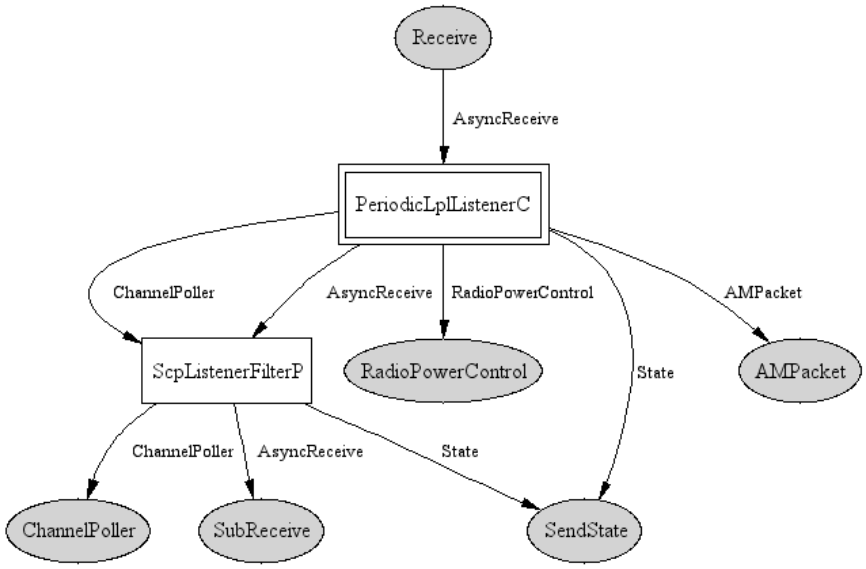

Figura 5.5: Diagrama del componente ScpListenerC del código MLA de SCP. PeriodicLplListenerC ordena apagar la radio durante el funcionamiento del protocolo.

```
packetCount = 0;
}
signal ChannelPoller.activityDetected(detected ||
(lastCount > 0) ||
(call SendState.getState() == S_BOOTING));
}
async event void SubReceive.receive(message_t * msg, void * payload, uint8_t
{
atomic packetCount++;
signal Receive.receive(msg, payload, len);
}
```

En la parte superior del código mostrado se ve cómo maneja `ScpListenerFilterP` la señal recibida. Lo que hace es retransmitir pero modificando el parámetro booleano. Se encontró que esta forma de modificar el parámetro es la causa de que al recibir un mensaje la radio se mantuviera prendida durante dos períodos, debido a la variable `lastCount`.

En la parte inferior del código mostrado queda claro el manejo de la variable `lastCount`. Cada vez que el mote recibe un mensaje se incrementa. Después, cuando se hace el chequeo de canal se reestablece a cero.

Lo que provoca esto, que no debería hacerlo, es que cada vez que el mote recibe un mensaje, `lastCount` se incrementa y pasa a ser mayor que cero. Entonces, en el

siguiente chequeo de canal (que para entonces el mensaje ya se transmitió y no es necesario mantener la radio prendida), el hecho de que `lastCount` sea mayor que cero, provoca que `ScpListenerFilterP` señale que el canal está ocupado cuando en realidad está libre. Como se verá enseguida esto provoca en última instancia que la radio no se apague cuando debería. Esta era la causa de que la radio se mantuviera prendida durante dos períodos completos cada vez que el mote recibía un mensaje. Al modificar esta parte del código se logró que la radio se mantenga prendida durante un período cada vez que el mote recibe un mensaje.

La señal modificada por `ScpListenerFilterP` es recibida por `PeriodicLplListenerC` que finaliza el ciclo de chequeo de canal apagando o no la radio:

```
PeriodicLplListener:
async event void ChannelPoller.activityDetected(bool detected)
{
if(!(call SendState.isIdle()))
return;
// Don't do anything if the radio is being used
if(!detected)
call RadioPowerControl.stop();
// Stop the radio if the channel is free
}
```

`PeriodicLplListenerC` ordena apagar la radio si, no está siendo usada para enviar un mensaje, y si el parámetro recdibido vale `FALSE`. De esta manera se termina el chequeo del canal.

5.2.4. Duración del chequeo de canal

Se vio más arriba en 5.2.2, que la duración del chequeo de canal cuando no se detecta actividad, depende de la variable `CcaCheckLength`, dentro del código del componente `PowerCycleC`. El reloj que se accede para medir el tiempo es de 32,768 khz y `CcaCheckLength` debe estar en cantidad de ciclos de reloj.

El valor de `CcaCheckLength` se establece en otra parte del código de `PowerCycleC`:

```
PowerCycle:
  async command void ChannelMonitor.setCheckLength(uint16_t ms) {
    ccaCheckLength = ms * 32;
  }
```

Se trata de un comando proveniente de la misma interfaz `ChannelMonitor` conectada a `MacC`. Es de suponer que el parámetro de entrada `ms` venga dado en milisegundos, y que al multiplicarse por 32, dé aproximadamente los ciclos de reloj necesarios, que es el valor asignado a `ccaCheckLength`.

Dentro de `MacC`, el componente que envía el comando es `ScpSenderC`. `ScpSenderC` es a su vez una configuración formada por dos componentes, la interfaz `ChannelMonitor` está conectada a `ScpSenderP`. El código original es el que se muestra:

```
ScpSenderP:
command error_t StdControl.start()
{
// if(call SendState.getState() != S_STOPPED)
// return FAIL;
// Make sure that the radio is off
call ChannelMonitor.setCheckLength((MAX_TONE_TIME + TX_TIME_SCHED) * 32);
call SendState.forceState(S_BOOTING);
return SUCCESS;
// Move to the bootstrap phase
}
```

El comando se llama al iniciarse el componente y el parámetro de se establece como la suma de dos constantes (`MAX_TONE_TIME + TX_TIME_SCHED`) multiplicadas por 32. Es particularmente notable que se multipliquen las constantes por 32, ya que después en `PowerCycleC` se vuelve a hacer una multiplicación por 32. Además, parece razonable que las constantes `MAX_TONE_TIME` y `TX_TIME_SCHED` estén definidas en milisegundos. Por lo tanto, una de las multiplicaciones por 32 parece estar de más.

En 4 se describe que en las primeras pruebas la duración del chequeo de canal era de 300 milisegundos aproximadamente y que eso era mucho. Al observar la doble multiplicación por 32 recién descrita, se eliminó una de ellas y se obtuvo una duración de chequeo de canal de 10 milisegundos que es más cercano a lo que se esperaba. Con este nuevo valor la duración del chequeo de canal sigue siendo suficiente para determinar si hay actividad en el canal o no.

5.3. Sincronismo

Repasando lo visto en el capítulo 2, el mecanismo de sincronización debe asegurar que todos los motes vecinos prendan la radio al mismo tiempo, permitiendo que un mote que quiera transmitir un mensaje, sepa cuándo el receptor va a escuchar el canal.

Es por esto que, previo a enviar un mensaje, el transmisor comienza a enviar el preámbulo un tiempo corto antes de la escucha del canal.

En la implementación MLA de SCP, el momento de prendido de la radio para hacer la escucha del canal, está determinado por un contador de tiempo llamado `LplAlarm`. Adicionalmente a este contador, cada mote tiene otro contador, llamado `SendAlarm` que indica el momento para empezar a enviar el preámbulo. Por lo tanto, `SendAlarm` da su aviso un tiempo corto antes de que `LplAlarm` lo dé.

La sincronización se logra intercambiando información de sincronización entre los motes. Cada cierto tiempo, cada mote transmite el contenido de su contador `LplAlarm`. A su vez, cada vez que un mote recibe la información del contador `LplAlarm` de un vecino, reinicia su propio contador `LplAlarm` y también `SendAlarm` para que coincidan con los del vecino.

Para transmitir la información de sincronización se hace *piggybacking* de la información de sincronización. Cada vez que un mote va a enviar cualquier tipo de mensaje, se le agrega a dicho mensaje un pie (footer) con el contenido de `LplAlarm`.

Por otro lado, si un mote pasa un determinado tiempo sin intercambiar información de sincronización, envía un mensaje explícito de sincronización. Esto está implementado con un tercer contador de tiempo llamado `SyncAlarm` y una variable llamada `piggybacked`. Cada vez que se envía un dato con su respectivo pie de sincronización, a `piggybacked` se le da el valor `TRUE`. A su vez, cada vez que `SyncAlarm` da su aviso, si `piggybacked` vale `FALSE`, se envía un mensaje explícito de sincronización con el contenido de `LplAlarm`, y en caso contrario, se le da a `piggybacked` el valor `FALSE` y no se hace más nada.

5.4. Booteo

El SCP cuenta con un módulo que en el momento de encender la radio a través de la interfaz que provee MLA, `StdControl.start()`, entra en un ciclo de booteo. El mismo es el encargado de sincronizar por primera vez el mote que se enciende con la red de motes.

Esta instancia corre por diferentes módulos hasta terminar en `ScpBootC`. El mismo cuenta con una evento llamado `start()` que cambia el estado de la radio a `boot`. Cuando comienza, verifica que la radio esté booteada, en caso contrario genera un paquete de booteo y lo envía durante 2 períodos T.

```
async command error_t AsyncStdControl.start() {

    if(call BootState.requestState(S_BOOTING) != SUCCESS)
```

```
            return EBUSY;
        call AMPacket.setType(&boot, AM_SCPBOOTMSG);
        call AMPacket.setSource(&boot, TOS_NODE_ID);
        call AMPacket.setDestination(&boot, AM_BROADCAST_ADDR);
        call AMPacket.setGroup(&boot, call AMPacket.localGroup());
        call BootAlarm.start(call LowPowerListening.getLocalSleepInterval() * 2)
        send();
        return SUCCESS;
    }
```

Se inicia una alarma `BootAlarm` para dentro de dos períodos T, mientras mientras el cual siempre está mandando un mensaje de boot.

En el momento en que un eventual vecino recibe un mensaje de sincronismo, el módulo `ScpSyncReceiver` de dicho vecino genera un evento de sincronismo a través del módulo `ScpSyncSender`.

```
    task void sendSyncTask() {


        call ScpSyncSender.sendSyncPacket(); call SubReceive.updateBuffer(msg_);
    }
```

El mismo genera un mensaje de sicronismo enrte los vecinos que determina el comienzo del sincronismo.

Capítulo 6

Consumo

6.1. Introducción.

Uno de los objetivos principales del proyecto es lograr que la red se mantenga en servicio durante un año con el uso de dos pilas AA, por esto se analiza cuál es el consumo de los motes con la implementación utilizada. Se asumen pilas de 1000mAh de carga.

6.2. Medida

El método utilizado para medir el consumo consistió en relevar con el osciloscopio la caída de tensión de una resistencia de 10 Ohm al 1 % intercalada en la alimentación, como se muestra en la figura 6.1. La necesidad de que el valor de resistencia sea pequeño es para que la caída de tensión sobre la misma no afecte el funcionamiento del mote.

Se observaron claramente dos niveles de consumo, un pico de de 20 mA aproximadamente que corresponde a cuando la radio está prendida, rodeado de niveles de consumo mucho menores que corresponde a cuando la radio está apagada.

Estas medidas resultaron fundamentales para diagnosticar y solucionar problemas de tiempos como se describe en el capítulo 4.

6.3. Niveles de consumo del mote

El consumo de los motes es causado principalmente por la radio y el microcontrolador. En realidad, cada uno de estos componentes tiene varios niveles de consumo

Figura 6.1: Imagen ilustrativa de conexión de resistencia para medir el consumo del mote

Typical Operating Conditions

	MIN	NOM	MAX	UNIT
Supply voltage	2.1		3.6	V
Supply voltage during flash memory programming	2.7		3.6	V
Operating free air temperature	-40		85	°C
Current Consumption: MCU on, Radio RX		21.8	23	mA
Current Consumption: MCU on, Radio TX		19.5	21	mA
Current Consumption: MCU on, Radio off		1800	2400	µA
Current Consumption: MCU idle, Radio off		54.5	1200	µA
Current Consumption: MCU standby		5.1	21.0	µA

Figura 6.2: Tabla de condiciones típicas de funcionamiento del mote Tmote Sky, extraído de la hoja de datos.

que, combinados, determinan que los motes tienen una cantidad significativa de posibles niveles de consumo.

La hoja de datos de la plataforma Tmote Sky usada en la aplicación, especifica, por un lado, de forma bastante sintética el consumo de cinco niveles significativos de funcionamiemto (figura 6.2), y por otro, con un poco más de detalle, el consumo de sus componentes constitutivos. A su vez, las hojas de datos de los componentes arrojan que la información en la hoja de datos del mote no es completa.

De todos modos se pueden sacar algunas conclusiones corroboradas con las medidas. En la tabla de la figura 6.2 se especifican dos niveles (similares) de consumo con la radio prendida. Para simplificar se puede tomar un nivel conservador de 23mA para la radio prendida.

Cuando la radio está apagada, los valores de la tabla de la figura 6.2 son bastante dispares. El problema es que el microcontrolador tiene varios niveles de consumo con diferencias importantes entre ellas. En las figuras 6.3 y 6.4 se muestran las tablas de condiciones típicas de operación del microcontrolador, según la hoja de datos del mote y del propio microcontrolador respectivamente.

Typical Operating Conditions

	MIN	NOM	MAX	UNIT
Supply voltage during program execution	1.8		3.6	V
Supply voltage during flash memory programming	2.7		3.6	V
Operating free air temperature	-40		85	°C
Low frequency crystal frequency		32.768		kHz
Active current at Vcc = 3V, 1MHz		500	600	µA
Sleep current in LPM3 Vcc = 3V, 32.768kHz active		2.6	3.0	µA
Wake up from LPM3 (low power mode)			6	µs

Figura 6.3: Tabla de condiciones típicas de funcionamiento del microcontrolador MSP430 F1611 según la hoja de datos de Tmote Sky

electrical characteristics over recommended operating free-air temperature (unless otherwise noted)

MSP430F15x/16x supply current into AV_{CC} + DV_{CC} excluding external current (AV_{CC} = DV_{CC} = V_{CC})

PARAMETER		TEST CONDITIONS		MIN	NOM	MAX	UNIT
$I_{(AM)}$	Active mode, (see Note 1) $f_{(MCLK)} = f_{(SMCLK)}$ = 1 MHz, $f_{(ACLK)}$ = 32,768 Hz XTS=0, SELM=(0,1)	T_A = –40°C to 85°C	V_{CC} = 2.2 V		330	400	µA
			V_{CC} = 3 V		500	600	
	Active mode, (see Note 1) $f_{(MCLK)} = f_{(SMCLK)}$ = 4,096 Hz, $f_{(ACLK)}$ = 4,096 Hz XTS=0, SELM=3	T_A = –40°C to 85°C	V_{CC} = 2.2 V		2.5	7	µA
			V_{CC} = 3 V		9	20	
$I_{(LPM0)}$	Low-power mode, (LPM0) $f_{(MCLK)}$ = 0 MHz, $f_{(SMCLK)}$ = 1 MHz, $f_{(ACLK)}$ = 32,768 Hz XTS=0, SELM=(0,1) (see Note 1)	T_A = –40°C to 85°C	V_{CC} = 2.2 V		50	60	µA
			V_{CC} = 3 V		75	90	
$I_{(LPM2)}$	Low-power mode, (LPM2), $f_{(MCLK)} = f_{(SMCLK)}$ = 0 MHz, $f_{(ACLK)}$ = 32.768 Hz, SCG0 = 0	T_A = –40°C to 85°C	V_{CC} = 2.2 V		11	14	µA
			V_{CC} = 3 V		17	22	
$I_{(LPM3)}$	Low-power mode, (LPM3) $f_{(MCLK)} = f_{(SMCLK)}$ = 0 MHz, $f_{(ACLK)}$ = 32,768 Hz, SCG0 = 1 (see Note 2)	T_A = –40°C	V_{CC} = 2.2 V		1.1	1.6	µA
		T_A = 25°C			1.1	1.6	
		T_A = 85°C			2.2	3.0	
		T_A = –40°C	V_{CC} = 3 V		2.2	2.8	
		T_A = 25°C			2.0	2.6	
		T_A = 85°C			3.0	4.3	
$I_{(LPM4)}$	Low-power mode, (LPM4) $f_{(MCLK)}$ = 0 MHz, $f_{(SMCLK)}$ = 0 MHz, $f_{(ACLK)}$ = 0 Hz, SCG0 = 1	T_A = –40°C	V_{CC} = 2.2V / 3 V		0.1	0.5	µA
		T_A = 25°C			0.2	0.5	
		T_A = 85°C			1.3	2.5	

NOTES: 1. Timer_B is clocked by $f_{(DCOCLK)}$ = 1 MHz. All inputs are tied to 0 V or to V_{CC}. Outputs do not source or sink any current.
2. WDT is clocked by $f_{(ACLK)}$ = 32,768 Hz. All inputs are tied to 0 V or to V_{CC}. Outputs do not source or sink any current. The current consumption in LPM2 and LPM3 are measured with ACLK selected.

Current consumption of active mode versus system frequency, F-version

$I(AM) = I(AM)\ [1\ MHz] \times f(System)\ [MHz]$

Current consumption of active mode versus supply voltage, F-version

$I_{(AM)} = I_{(AM)\ [3\ V]} + 210\ \mu A/V \times (V_{CC} - 3\ V)$

Figura 6.4: Tabla de condiciones típicas de funcionamiento del microcontrolador MSP430 F1611 según la hoja de datos de Tmote Sky

El nivel de consumo del microcontrolador es manejado por el sistema operativo TinyOS, donde se afirma que se mantiene el menor nivel posible. De las medidas se puede decir que cuando la radio está apagada el microcontrolador se mantiene en uno de sus modos de bajo consumo. Esto tiene sentido porque cuando la radio está apagada el microcontrolador no necesita hacer ninguna operación y se encuentra esperando una interrupción.

Teniendo todo esto en cuenta, resulta razonable estimar un valor promedio de consumo con la radio apagada cercano a los $50\mu A$ como propone la tabla de la figura 6.2. Esto corresponde al microcontrolador operando entre los niveles LPM2 y LPM0.

6.4. Consumo en función de los tiempos y de los niveles de consumo de los motes

Tomando como referencia un año de funcionamiento y asumiendo que los motes tienen dos niveles principales de consumo (radio prendida y radio apagada), el consumo de un mote medido en carga eléctrica se puede expresar como

$$C = I_{ROn} \cdot t_{ROn} + I_{Sleep} \cdot (t - t_{ROn}) \tag{6.1}$$

donde C es el consumo, I_{ROn} es el consumo de corriente cuando la radio está prendida, t_{ROn}es el tiempo en que la radio permanece prendida durante todo el año, I_{Sleep} es el consumo de corriente cuando la radio está apagada y t es el tiempo de referncia de un año.

Se introducen

$$C_{ROn} = I_{ROn} \cdot tROn \tag{6.2}$$

$$C_{Sleep} = I_{Sleep} \cdot (t - t_{ROn}) \tag{6.3}$$

donde C_{ROn} es el consumo total debido al tiempo en que la radio permanece prendida y C_{Sleep} es el consumo total debido al tiempo en que la radio permanece apagada. De esta forma

$$C = C_{ROn} + C_{Sleep} \tag{6.4}$$

El tiempo t_{ROn} depende de los tiempos de prendido y apagado de la radio. Esto se puede expresar como

$$t_{ROn} = t_{Listen} \cdot N_{Listen} + t_{TX} \cdot N_{TX} + t_{RX} \cdot N_{RX} \tag{6.5}$$

donde t_{Listen}es el tiempo de escucha de cada vez, N_{Listen}es la cantidad de veces que se hace escucha de canal durante un año, t_{TX} es el tiempo de transmisión de un mensaje, N_{TX} es la cantidad de veces que se transmiten mensajes durante un año, t_{RX} es tiempo de recepción de un mensaje, y N_{RX}es la cantidad de veces que se reciben mensajes.

Se introducen

$$C_{Listen} = I_{ROn} \cdot t_{Listen} \cdot N_{Listen} \tag{6.6}$$

$$C_{TX} = I_{ROn} \cdot t_{TX} \cdot N_{TX} \tag{6.7}$$

$$C_{RX} = I_{ROn} \cdot t_{RX} \cdot N_{RX} \tag{6.8}$$

donde C_{Listen} es el consumo debido a la escucha, C_{TX} el consumo debido a la transmisión de mensajes y C_{RX} el consumo debido a la recepción. De esta forma se obtiene

$$C_{ROn} = C_{Listen} + C_{TX} + C_{RX} \tag{6.9}$$

y

$$C = C_{Listen} + C_{TX} + C_{RX} + C_{Sleep} \tag{6.10}$$

De esta forma el consumo queda discriminado según sus diferentes causas.

Por otro lado, las cantidades N dependen a su vez de la frecuencia con la que se hacen las escuchas transmisiones y recepciones, como se detalla a continuación. En primer lugar

$$N_{TX} = \frac{t}{T_{MSG}} \tag{6.11}$$

donde t es el tiempo total de referencia de un año y T_{MSG} es el período cada el cual se transmite un mensaje.

Análogamente

$$N_{RX} = n \cdot \frac{t}{T_{MSG}} \tag{6.12}$$

dende n es la cantidad de vecinos que tiene cada mote. Esto es así porque cada mote al hacer una escucha y detectar actividad, mantiene la radio prendida hasta recibir el mensaje, y recién después descarta el mensaje si determina que no era el destinatario.

Finalmente

$$N_{Listen} = \frac{t}{T} - N_{TX} - N_{RX} \tag{6.13}$$

donde T el período cada el cual se hace una escucha de canal. Tener en cuenta que cada transmisión y recepción de mensajes se hace en el mismo momento en que se iba a hacer una escucha de canal, por eso la resta.

Por lo tanto

$$C_{TX} = I_{ROn} \cdot t_{TX} \cdot \frac{t}{T_{MSG}} \tag{6.14}$$

$$C_{RX} = I_{ROn} \cdot t_{RX} \cdot n \cdot \frac{t}{T_{MSG}} \tag{6.15}$$

$$C_{Listen} = I_{ROn} \cdot t_{Listen} \cdot t \cdot \left(\frac{1}{T} - \frac{t_{TX} + n \cdot t_{RX}}{T_{MSG}} \right) \tag{6.16}$$

De esta forma, todos los consumos quedan expresados en función de parámetros de tiempo de la aplicación.

Repasando, los parámetros que determinan el consumo del mote son:

- I_{ROn} - Consumo de corriente con radio prendida
- I_{Sleep} - Consumo de corriente con radio apagada
- t_{Listen} - Tiempo que dura una escucha de canal
- T - Período cada el cual se hace una escucha de canal
- t_{TX} - Tiempo que dura un envío de mensaje
- T_{MSG} - Período cada el cual un mote envía un mensaje
- t_{RX} - Tiempo que dura una recepción de mensaje
- n - Cantidad media de vecinos de cada mote

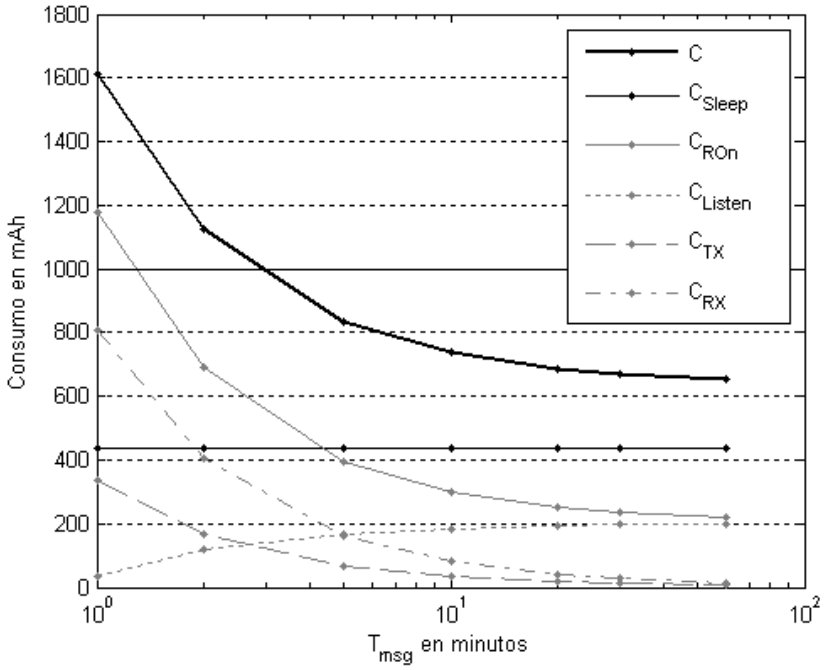

Figura 6.5: Consumo anual de un mote para valores de T_{MSG} entre 1 minuto y 1 hora, comparado con la referencia de carga de una pila de 1000mAh

I_{ROn} e I_{Sleep} dependen del mote y están discutidos en la sección 6.3, t_{Listen}, T, t_{TX} y t_{RX} dependen de cómo se fijen los parámetros del protocolo SCP, n depende de la distribución de los motes y T_{MSG} depende directamente de los requerimientos de la aplicación.

En la sección 6.3 se propone tomar los valores $I_{ROn} = 23mA$ e $I_{Sleep} = 50\mu A$. En el capítulo 4 se establecen $t_{Listen} = 10ms$, $T = 10s$, $t_{TX} = 100ms$ y $t_{RX} = 60ms$. Finalmente, asumiendo una distribución de los motes en cuadrícula, se puede asumir $n = 4$.

6.5. Resultados

Tomando como referencia los valores recién indicados, y para T_{MSG} entre 1 minuto y una hora, se obtienen los niveles de consumo de la gráfica de la figura 6.5.

Tomando como referencia una carga total de las pilas de 1000mAh, resulta que para T_{MSG} de 1 y 2 minutos, el consumo anual sobrepasa el límite, y para T_{MSG} mayores la carga de la pila es suficiente para un tiempo de operación de un año.

C_{Sleep} permanece prácticamente constante debido al ciclo de trabajo bajo, y asumiendo el nivel de consumo $I_{Sleep} = 50\mu A$ que se tomó en 6.3, C_{Sleep} resulta consid-

ereable. Para T_{MSG} mayor a 5 minutos, C_{Sleep} es la principal causa de consumo.

También para T_{MSG} mayor a 5 minutos, la principal causa de consumo por radio prendida (C_{ROn}), es debido a la escucha (C_{Listen}).

Este límite inferior para T_{MSG} de 5 minutos es satisfactorio para las aplicaciones agrícolas para la que está pensado este trabajo. Si una determinada variable se mide cada media hora por ejemplo, un mote puede transmir su medida y la de 5 motes más que deba retransmitir.

Capítulo 7

Capas superiores

7.1. Introducción

Paralelamente a lo trabajado en la capa MAC que era el énfasis del proyecto, también se trabajó con las capas superiores con la intención de cerrar una aplicación completa funcionando.

Las capas superiores incluyen la capa de red, la idea inicial era tomar el protocolo de red *Collection Tree Protocol* (CTP) que es parte de las bibliotecas de TinyOS. Siguiendo el modelo de capas, se suponía que el protocolo de capa de red debía ser independiente del protocolo de control de acceso al medio, y por lo tanto, CTP se iba a poder montar sobre nuestra implementación SCP. Sin embargo, al probar CTP implementada sobre nuestra implementación SCP, resultó que eran incompatibles. Postriormente se comprobó que todas las implementaciones que introducen un protocolo de acceso al medio diferente al original de TinyOS tambien deben modificar la capa de red, mostrando que el CTP no es independiente de las capas inferiores originales de TinyOS.

A raíz de esto se decidió tomar otra implementación que incluía hasta la capa de red, pero con una implementación de SCP distinta a la nuestra, a los efectos de probar el resto de la aplicación. El resto de la aplicación se pudo implementar sin problemas tanto sobre el CTP de TinyOS como en la otra implementación.

7.2. Recolección y Diseminación

7.2.1. Introducción

Uno de los objetivos que tiene la red en este proyecto es que sea capaz de transmitir mensajes utilizando multi-hop, para esto es necesario implementar en nuestra

red algún protocolo de ruteo capaz de redireccionar los mensajes hacia un nodo central, a la vez es necesario que la red sea capaz de difundir un valor de algunas variables en el sistema, como ser el periodo de recolección de datos o el valor base de la estampa de tiempo. Todos los protocoles de red se encontraran por encima de la capa MAC implementada. La capa MAC, esta se encargara de administrar el acceso al medio.

7.2.2. Recolección

Para lograr la recolección de datos de la red se optó por el *CTP* ("Collection Tree Protocol") [2], ya que el mismo cumple los requerimientos del sistema y es uno de los protocolos estándares del tinyOS, esto asegura su correcta funcionalidad dado que está probado.

El CTP esta basado en una red tipo árbol, esto significa que existe en ella un único nodo raíz, y luego existen semillas que son las que llevan información a la raíz. Para el sistema propuesto se toma como el nodo raíz el mote que se encuentra conectado a la PC, y como nodos semillas a los motes que están relevando información de la planta[1].

Se dice que CTP es un protocolo que se basa en la idea de realizar el "mejor esfuerzo", este es capaz de realizar un reconocimiento de la calidad del canal basado en la pérdida de mensajes. Además resuelve el tema de loop cerrados y es capaz de converger a una red estable y asimilar rápidamente a motes nuevos.

CTP está diseñado para tráficos relativamente bajos con sistemas de ancho de banda limitado, que son los parámetros en los que se espera que la red de este proyecto trabaje.

7.2.3. Diseminación

La diseminación [3] consiste en una implementación que se encarga de mantener sincronizadas variables a elección en la red, básicamente funciona por inundación, en el momento en que se realiza un cambio de la variable observada esta se empieza a inundar desde el mote que genera el cambio a todos los motes posibles, a su vez cada mote que escucha el cambio transmite a todos los motes que estén a su alcance este nuevo valor, si el que escucha ya había cambiado su variable no realiza nada, pero si en cambio tiene que realizar el cambio, no solo corrige su valor a nuevo sino que difunde este nuevo valor, este algoritmo converge rápidamente a estabilizar el valor en la red.

En el sistema del proyecto, el único mote que iniciará una diseminación será el mote raíz, este se encargará de iniciar la diseminación de loa parámetros de

[1] se toma por planta como área de trabajo

configuración, como ser el valor base de la estampa de tiempo o el período de muestreo.

7.3. Identificación de Paquetes

Este capitulo pretende establecer los paquetes que se utilizaran en la comunicación inalámbrica.

7.3.1. Paquete de Medida

Este paquete tiene como propósito informar por parte del mote hoja, al mote central la medida de uno de sus sensores, es importante observar que el paquete en cuestión contiene una estampa de tiempo.

Paquete de Medida			
Registro	Descripción	Ident	Bits
1	Numero de Mote	moteId	16
2	Tipo de Sensor	sensorType	5
3	Numero de Sensor	sensorId	3
4	Medida	reading	16
5	TimeStamp	timestamp	16

7.3.2. Paquete de control

Este paquete tiene como propósito setear por parte de mote central alguna variable de los motes periféricos, o informar los por parte de los motes periféricos del estado de una variable. Es posible que un mote informe del estado de la completitud de sus variables a través de consecutivos mensajes control.

Paquete de Control			
Registro	Descripción	Ident	Bits
1	Numero de Mote	moteId	16
2	Setear	setValue	8^2
3	Numero de variable	varID	8
4	Valor	value	32
5	TimeStamp	timestamp	32

7.3.3. Paquete de Topología

Este paquete tiene como propósito informar al mote central de quién es el padre de cada uno de los motes en cuestión.

Topología			
Registro	Descripción	Ident	Bits
1	Numero de Mote	moteId	16
2	Mote Padre	fatherId	16
3	TimeStamp	timeStamp	32

7.4. Time Stamp

7.4.1. Introducción y planteo del problema

En una red de sensores en donde cada uno de los nodos genera medidas de alguna variable ambiental, y estas medidas son transmitidas desde los nodos ubicados en la planta al nodo central, se hace imprescindible ordenar los datos, y conocer por un lado el nodo que hizo la medida, y por otro, el instante en que se tomaron las medidas. De esta forma se puede reconstruir cómo se encontraba la planta en un instante dado, obteniéndose una "foto" de la situación de la planta una hora dada.

El problema principal es que el nodo sumidero que recolecte los datos no puede confiar en la hora de llegada de los datos para estimar la hora de la medida, porque en general se desconoce la latencia entre la medida del dato y su llegada al nodo sumidero.

Una forma robusta de conocer el instante en que se toman las medidas, es que cada nodo sepa la hora global y que estampe en los mensajes enviados con las medidas, dicha hora global (Time Stamp).

El único nodo que tiene acceso a la hora global es el nodo sumidero que está conectado a la computadora que recolecta los datos. Se asume que la hora de dicha computadora es la hora global, y que se quiere conocer los momentos de las medidas respecto a la hora de computadora.

Para que el resto de los nodos tengan un registro de la hora global, el nodo sumidero lo debe transmitir periódicamente. El problema principal de esta solución es el mismo que el problema original, no se conoce la latencia de transmisión de los mensajes, y por lo tanto tampoco se conoce la latencia de transmisión de la hora global a partir del nodo sumidero al resto de los nodos.

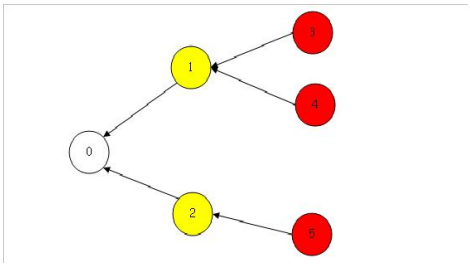

Figura 7.1: Diagrama de Tiempo

7.4.2. Propuesta

El nodo sumidero transmitirá cada mediodía y cada medianoche la hora global y los nodos tendrán un contador de segundos que represente la cantidad de segundos transcurridos desde el principio del año.

Se propone utilizar 32 bits, los contadores de los motes cambiarían cada un segundo. De esta propuesta se obtiene las siguientes posibilidades.

El nodo sumidero transmitirá la hora a sus nodos adyacentes, y estos retransmitirán la hora a los siguientes en la jerarquía, y así sucesivamente hasta llegar a los nodos más alejados. Debido a la experiencia practica en anteriores proyectos se decide utilizar diseminación para controlar el valor de la hora global, es sabido que en la practica que el tiempo de latencia en el cual la variable se actualiza es despreciable para la aplicación que esta pensada la red.

Cada nodo que recibe la varibale a controlar resetea su reloj y el contador de segundos interno, luego de esto cada vez que tenga que mandar un mensaje imprimira el tiempo de la variable sumado al contador interno. De esta forma nos aseguramos de seguir al reloj del sistema dentro de un ragno aceptable.

7.5. Comunicación PC-Mote

7.5.1. Introducción

La comunicación PC-Mote es uno de los puntos fundamentales de la aplicación a desarrollar debido a muchos puntos, el primero y sin lugar a dudas es que esta red tiene como objetivo especifico monitoriar algunas variables ambientales y desplegar esta información al usuario de la red, si esta información no es desplegada en pantalla o mostrada de alguna forma, que permita alguna toma de decisión no serviría de nada. Por esto es imprescindible que la red sea capaz de enviar la información por un puerto de comunicación a un sistema mas inteligente, para

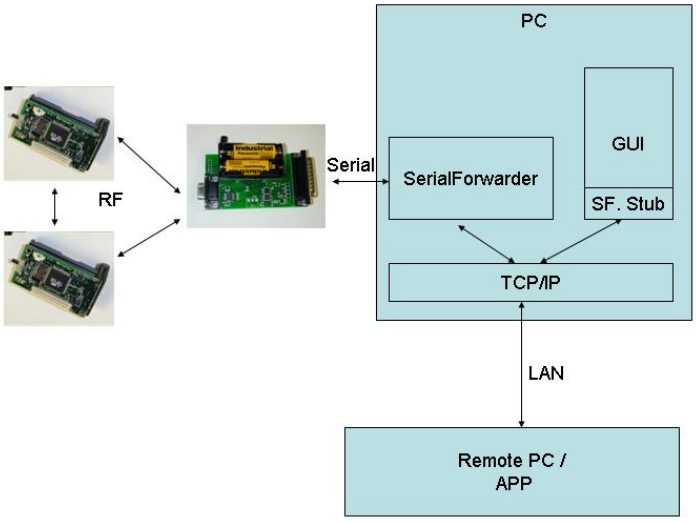

Figura 7.2: Comunicación TCP

que este tome alguna decisión, sea desplegar en pantalla, archivar para histórico, activar un dispositivo, etc

7.5.2. Puerto Serial

Los dispositivos telosb vienen con un puerto serial incorporado, es este el que permite esta comunicación. En la aplicación a desarrollar se elije que el único mote que es capaz de mandar información por el puerto serial es el que se encuentra conectado al PC, el mote raíz.

El mote raíz es el encargado de servir como interfaces entre el mundo del ordenador un sistema mas inteligente que la red a desarrollar, y el universo de los motes donde la comunicación es básicamente inalámbrica.

El PC contendrá una aplicación en java que se encargara de recibir la información de dicho mote y desplegará en pantalla, además de archivar la en un formato de archivo que sea capaz de trabajarlo luego como una base de datos.

7.5.3. Interfaz TCP/IP

Se decidió realizar la lectura del puerto serial a través de un puerto TCP, esto representa una gran ventaja debido a que es posible recabar la información de nuestra red en cualquier parte del globo terráqueo.

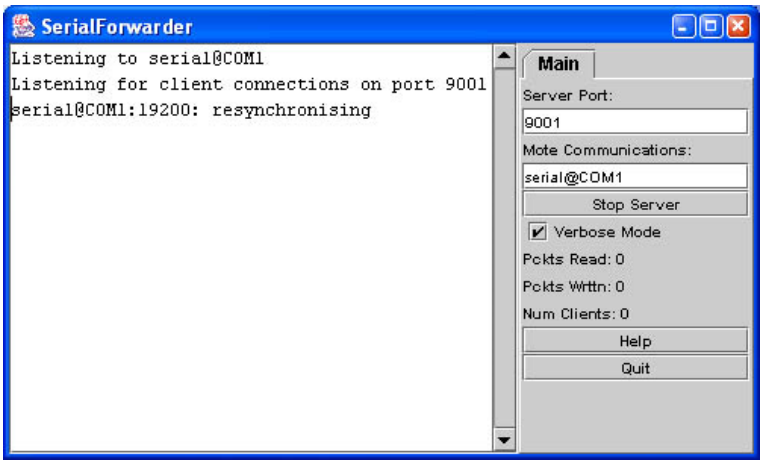

Figura 7.3: SerialForwarder

Para lograr esto se utiliza el software ya implementado por tinyOS SerialForwarder, este programa se encarga de re direccionar la lectura y la escritura del puerto TCP al puerto serial.

7.5.4. MIG

MIG significa "Message Interface Generator for NesC", esta es una herramienta que viene con el tinyOS, esta se encarga de generar una clase java a partir del archivo .h que define un tipo de mensaje.

Esta clase generada esta diseñada para poder interpretar los bytes en forma de un mensaje ordenado, identificando sus campos y su contenido. Se generan autmáticamente métodos capaces de leer un campo o escribirlo.

De esta forma MIG es un gran aliado al momento de generar un sistema en el PC, dado que simplifica y ordena la forma en la que java trabaja, pudiendo tratar a los mensajes como objetos propiamente dichos.

7.6. Manual de Usuario

7.6.1. Interfaz de Usuario.

7.6.1.1. Introduccion

La interfaz diseñada para la recolección de datos y almacenamiento esta formada por las ventanas de datos, conexión y topología según muestra la figura 7.4. El

Figura 7.4: Interfaz de usuario, en esta imagen se muestra la pestaña de conexiónes

programa tiene la opción de observar los datos en la maquina donde se encuentra el mote raíz conectado al puerto serial, o acceder a estos desde cualquier sitio mediante internet. También posee una descripción sencilla de la topología de la red en tiempo real. A su vez se eligió por practicidad el servidor de datos Mysql para almacenar y analizar la información recolectada.

7.6.1.2. Instructivo de uso

1. Seleccionar la ventana "Conexión".

2. Si el usuario se encuentra en la maquina donde se encuentra el mote raíz, haga click en "Puerto Serial". Por defecto se configura el puerto serial USB0[3], en caso que el mote se encuentre conectado en otro puerto, cargar el nombre del puerto en donde dice "Puerto". Luego hacer click en "Conectar".

3. En caso contrario, seleccionar "Internet". Luego cargue la dirección IP del servidor donde se encuentra conectado el mote raíz. Luego hacer click en

[3]Las direcciones de puerto en Linux son de la fomra /dev/ttyUSB0 mientras que para los usuarios windows sera de la foram com0

Figura 7.5: Interfaz de conexión

"Conectar"[4].
Importante: Una vez que se estable una conexión entre el programa y el puerto serial o el puerto TCP/IP la interfaz indica que esta acción se realizo satisfactoriamente con la palabra "Conectado" según se observa en 7.5.
Una vez que el programa se encuentre conectado se puede observar los diferentes datos que se estan recibiendo desde los distintos motes que forman la red o observar la topologia de la misma.

4. Si el usuario quiere observar los datos que se están recibiendo, seleccionar la ventana datos. ver figura 7.6.

5. En cambio, si el usuario quiere observar la topología de la red, seleccionar la ventana Topologia, ver figura

6. En el caso que el usuario desee almacenar la información que esta recibiendo en la ventana de "Datos" hacer click "habilitar Mysql" y luego presione

[4]La maquina donde se encuentra el mote raiz debe de estar corriendo un programa que retrasmita la información recibida por el puerto serial a traves de un puerto TCP/IP.

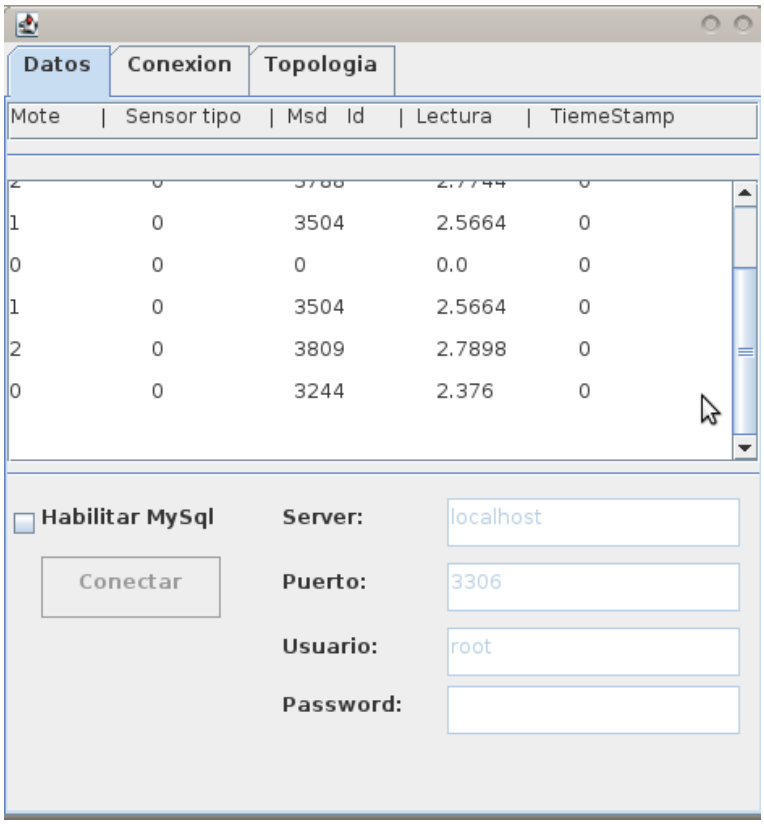

Figura 7.6: Ventana de Datos

Figura 7.7: Ventana de Topologia

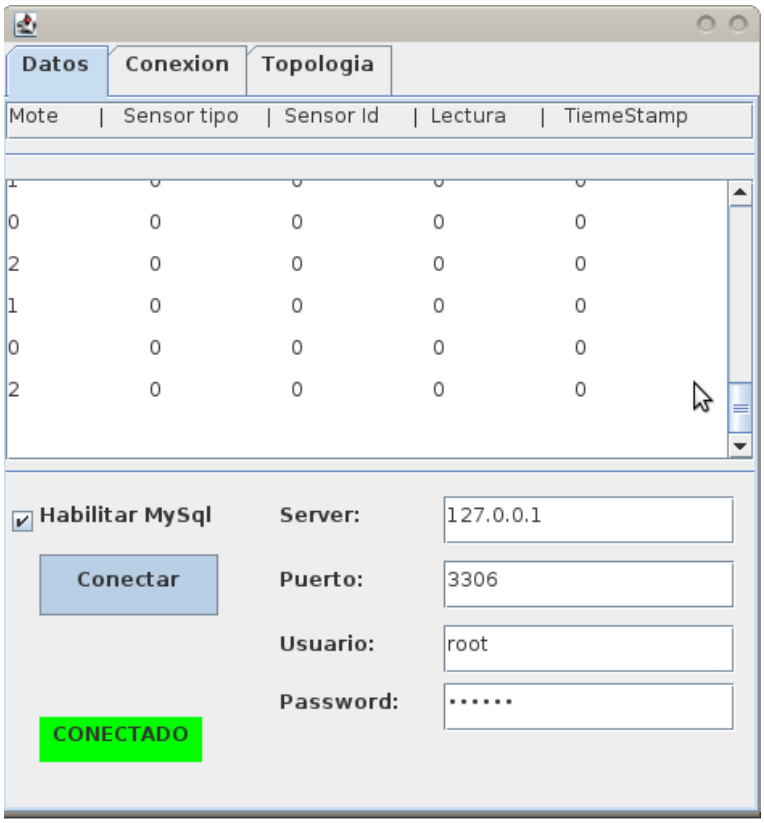

Figura 7.8: Conexion a MySql

"Conectar", si la conexión fue satisfactoria, aparece el mensaje "Conectado" según se muestra en la figura 7.8. A partir de este momento toda la información está siendo guardada en la base de datos Sql.
Es bueno notar que es posible conectarse con un servidor de datos que ofrezca su servicio a través de internet, para eso solo es necesario conocer la IP del servidor y tener un usuario registrado.

7. Para recuperar los datos guardados en el servidor de datos Sql, acceder a la planilla excel "Historico_Datos_Sensados" la cual previamente fue creada en la maquina el usuario.

Capítulo 8

Conclusiones

8.1. Conclusiones sobre SCP

Como se vió en el capítulo 2, entre los protocolos estudiados, el protocolo Schedualed Channel Polling (SCP), es el más eficiente para nuestra aplicación porque vincula las ventajas de los protocolos con sicronización, como S-MAC, combinando además la ventaja de LPL. La naturaleza de los datos que se estarán recibiendo de los sensores, o sea la periodocidad con que se reciben y envían datos hacen que el SCP sea uno de los protocolos más aptos para esta aplicación. Luego, se envestigó si este protocolo se encontrababa implementado, ya que si no era ese el caso no se podria utilizar, la implememtación de un protocolo no estaba dentro del alcance del proyecto. El grupo de trabajo de MLA tenia implementado este protocolo, por lo cual se decidió utilizar el protocolo SCP implementado por ellos.

8.2. Conclusiones sobre MLA

MLA, prometía y promete ser una plataforma para desarrollo de aplicaciones muy interesante, la flexibilidad que brinda a la hora de elegir el protocolo que mejor se adapte a una aplicación específica es su principal fortaleza.

Lamentablemente luego de trabajar con él, a nuestro criterio han sacrificado flexibilidad del conjunto, por robustez de los protocolos específicos. Específicamente, en el caso del protocolo SCP de MLA, este protocolo debe en teoría ser robusto con la sincronización de los motes, en cambio, la implementación realizada por MLA en este sentido es muy pobre, dado que la sincronización de los motes se realiza de forma muy promiscua, sin ningún tipo de elección de mote(padres) a sincronizar, dejando abierta la posibilidad de que por un mote se generen focos de desincronización o comportamientos erráticos. Tampoco se tiene en cuenta el retardo al momento de enviar la información de sincronización.

A nuestro criterio MLA consiste en un esfuerzo positivo para los futuros programadores de NesC, solo es necesario profundizar y darle robustez a los módulos que implementan los protocolos.

8.3. Conclusiones sobre SCP-MLA

A esta implementación de SCP se le encontraron los siguientes inconvenientes.

Constantede tiempo de escucha: Existe una constante que define el tiempo de escucha del canal que esta mal calculada. Esta constante se encuentra multiplicada dos veces por 32, generando tiempos de escucha mayores de lo necesario. Modificando esta constante pasamos de tener tiempos de escucha de 280 ms a 10 ms. Esto es un punto muy débil de la implementación. Estos tiempos iniciales descartarían inmediatamente esta capa, como capa de bajo consumo.

Sincronización: los módulos que se encargan de la sincronización están mal implementados debido a que no se realiza ninguna discriminación sobre qué mote esta enviando un reloj. La implementación está diseñada para que al escuchar cualquier mensaje que lleve información de reloj, automáticamente el mote se sincroniza de a cuerdo a ese reloj. Esto genera grandes percances a la hora de sincronizar la red en su conjunto. Este comportamiento no se podía modificar sin modificar excesivamente el código por lo cual no se realizó.

Además en el momento de tomar la información del reloj que entrega otro mote no se asume ningún tiempo de desfasaje entre que se envía el mensaje y el momento de recepción y proceso. Esto genera inconvenientes y atrasos al momento de recibir un paquete. Este último inconveniente fue controlado por una constante que fue seleccionada iterando y relevando la sincronización con el osciloscopio hasta llegar a un valor en atraso en milisegundos que corrija el tiempo de propagación.

Constantes de control: existen una serie de constantes que ofician de controladora del SCP, como ser:

1. Tiempo en que se enciende la radio antes para que el mote pueda enviar su mensaje
2. Tiempo en que la radio luego de despertarse para enviar su mensaje se mantiene enviando un paquete de preámbulo.

Estos tiempos ofician como constantes de control de variables discretas, dado que en conjunto estas sirven para asegurar que habrá comunicación sin

importar el pequeño desfasaje que tengan los motes.

Estas constantes eran muy pequeñas para el excesivo desfasaje que generaba los módulos de sincronismo, para la correcta transmisión de datos se modificaron a las siguientes constantes

1. Tiempo de encendido antes, 10ms (un tiempo de escucha).
2. Tiempo de preámbulo, 20 ms (durante este tiempo envía paquetes de preámbulo)

Esto nos asegura la correcta sincronización.

8.4. Concluciones sobre el Consumo

Segúnn lo visto en el capítulo de consumo, una vez que se realiza la modificación al código original (apagar la radio un tiempo pequeño luego de recibir datos, y en el caso que no halla más datos para recibir) se logra alcanzar el objetivo del consumo sin poner en riesgo la sincronizacion de la red.

8.5. Concluciones sobre Capa de Red-Capa MAC

Según lo mencionado en el capítulo de capas superiores, al momento de unir el protocolo de la capa de red con el protocolo de la capa Mac no fue posible compatibilizar el comportamiento de las capas. Para lograr esto se debía modificar el satck, lo cual no estaba dentro del alcance del proyecto.

Bibliografía

[1] Technical report, http://www.primidi.com/2007/02/27.html.

[2] The collection tree protocol (ctp). *http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html*.

[3] Dissemination. *http://www.tinyos.net/tinyos-2.x/doc/pdf/tep118.pdf*.

[4] http://mantis.cs.colorado.edu/index.php/tiki-index.php.

[5] http://tinyos.net/.

[6] *IEEE standard 802.15.4*. http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf.

[7] C.Enz A. El-Hoiydi, J-D Decotignie and E. Le Rouz. Wisemac: An ultra low power mac protocol for the wisenet wireless sensor networks. *Procedding of the First ACM SenSys Conference, Los Angeles, CA*, Julio 2003.

[8] Sunghyun Moon; Taekjoo Kim; Hojung Cha. Enabling low power listening on ieee 802.15.4-based sensor nodes. *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE*.

[9] Miguel Tasende Diego Baccino, Carolina Etchart. Siagro2. 2007.

[10] Sergio Mena Doce Jose Lopez Lopez. Desarrollo de un demostrador para evaluar tÃ©cnicas cross-layer en sistemas de comunicaciones inalÃ¡mbricos. *https://upcommons.upc.edu/pfc/bitstream/2099.1/4987/1/memoria.pdf*, 10 de Marzo de 2008.

[11] Jason Hill Joseph Polastre and David Culler. Versatile low power media ccess for wireless sensor networks. *Procedding of the 2nd ACM SenSys Conference*, pages 95 – 107, Noviembre 2004.

[12] Octav Chipara Chenyang Lu Kevin Klues, Gregory Hackman. A component-based architecture for power-efficient media access control in wireless sensor networks. *SenSys'07*, 2007.

[13] Philip Levis. Tinyos programming. *http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf*, 2006.

[14] Diego Mendez Nicolas Piriz, Sebastian Monzon. Si-agro. *IIE*, 2006.

[15] Pierre Baldi Raja Jurdak and Cristina Videira Lopes. Adaptive low power listening for wireless sensor networks. *IEEE TRANSACTIONS ON MOBILE COMPUTING, VOL. 6, NO. 8*, AUGUST 2007.

[16] Ivan Stojmenovic, editor. *Handbook of Sensor Networks: Algorithms and Architectures.*

[17] Koen Langendoen Tijs van Dam. An adaptive energy-efficient mac protocol for wireless sensor networks. 1:10, 2003.

[18] F. Silva W. Ye and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. *SenSys*, 1:14, 2006.

[19] John Heidemann Wei Ye and Deborah Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. 12:493 – 506, JUNE 2004.

Apéndice A

Filosofia de NESC

[13]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 bajo 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 una cantidad 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.

NesC combina ciertos aspectos de la orientación a objetos, en el sentido de que se basa en una programación orientada a interfaces y de la orientación a eventos, en el sentido de 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. A las librerías y a aplicaciones se les va 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 que son interfaces en

el sentido POO, por lo que en un momento dado se podría cambiar un componente por otro siempre y cuando este otro proporcionase la misma interfaz y este cambio no afectaría al código de la implementación de la aplicación.

De esta explicación se puede inducir, que se van a tener dos partes diferentes como mínimo para un componente, la parte de implementación (*módulo*) que estará programada hacia componentes y la parte de *configuración* que permitirá decidir qué componentes son los que estoy utilizando para proporcionar dichas interfaces a mi componente, esto se denomina *wiring*.

En cuanto a la orientación a eventos, se puede 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.

Apéndice B

El chip cc2420

El chip de la radio cc2420[10] es un dispositivo complicado que se ocupa de la mayoría de los detalles de bajo nivel de la comunicación saliente y entrante de paquetes mediante su electronica. La especificación del comportamiento adecuado de este hardware requiere una implementación de la pila radio bien definida. A pesar de que mucha de la funcionalidad está disponible dentro del propio chip, hay varios factores a considerar cuando se implementa la pila radio.

El software de la pila radio que maneja el sistema radio del CC2420 consiste en varias capas de funcionalidad que se asientan entre la aplicación y el hardware. El nivel más alto de la pila modifica datos y cabecera de paquete, mientras que el nivel más bajo determina el comportamiento en transmisión.

B.1. Descripción de capas

Las capas que se encuentran dentro de la pila radio, están en el siguiente orden según muestra la Figura B.1:

ActiveMessageP: Esta es la capa de mayor nivel de la pila, responsable de tratar detalles en la cabecera del paquete como la dirección de nodo de destino, la identificación de la fuente que ha transmitido un paquete y acceder al payload del paquete.

UniqueSend: Esta capa genera un byte de número de secuencia de datos (DSN: Data Sequence Number) único para la cabecera del paquete de datos. Este byte se incrementa cada vez que sale un paquete, empezando por un número generado de forma pseudo aleatoria. Un receptor puede detectar paquetes duplicados comparando la fuente del paquete recibido y su byte DSN con paquetes previos. DSN se define en la especificación del estándar 802.15.4.

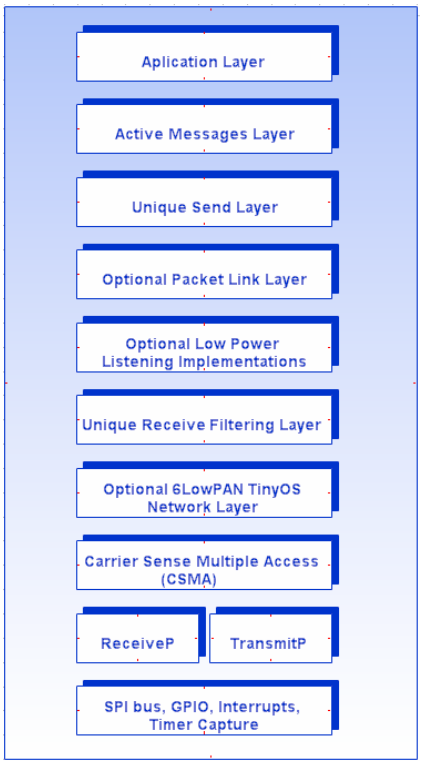

Figura B.1: Representación de la pila de capas del cc2420

PacketLink: Esta capa proporciona funcionalidades automáticas de retransmisión y es responsable del reintento de transmisión de un paquete si no llega un reconocimiento del receptor. PacketLink se activa por cada mensaje, significando que el mensaje saliente no usará PacketLink a menos que se configure antes de ese momento para hacerlo.

CC2420AckLplP/CC2420NoAckLplP: Estas capas proporcionan implementaciones asíncronas de escucha de baja potencia (*Low Power Listening*). Ambas capas las soporta el componente *CC2420DutyCycleP*. Este componente es responsable de activar y desactivar el sistema radio y comprobar las recepciones de paquetes. En cuanto se detecte actividad en el canal radio, *DutyCycleP* le transfiere la responsabilidad de realizar alguna transacción al componente *LowPowerListeningP* y apagar el sistema radio cuando sea conveniente. Las transmisiones *Low Power Listening* se activan para cada mensaje, y la capa retransmitirá continuamente el paquete saliente hasta que se escuche cualquier respuesta del receptor o hasta el vencimiento del tiempo de transmisión.

UniqueReceive: Esta capa mantiene el historial de las direcciones fuentes y byte DSN de unos pocos paquetes que han llegado con anterioridad, y ayuda a filtrar los paquetes recibidos duplicados.

TinyosNetworksC: Esta capa permite a la pila radio tinyos 2.x interactuar con otras redes que no sean de tinyOS. Las especificaciones 6LowPAN propuestas incluyen un byte de identificación de red después de la cabecera estándar 802.15.4. Si se usan tramas de interoperabilidad, la capa de envío proporciona la funcionalidad para la configuración del byte de red para paquetes salientes y filtro de paquetes entrantes que no sean de tinyOS.

CsmaC: Esta capa, que se trata en el apartado 1.4.3, es responsable, aunque no es su función principal, de definir el byte de información FCF (Frame Control Field) 802.15.4 en el paquete saliente, que es un campo de 2 bytes que indica el tipo de trama MAC, proporcionando un backoff por defecto cuando se detecta un uso del canal radio, y la definición del procedimiento de encendido y apagado del sistema radio, en otras palabras, aplica el MAC del estándar IEEE 802.15.4.

TransmitP/ReceiveP estas capas son responsables de interactuar directamente con el canal radio a través del bus SPI, las interrupciones y líneas GPIO.

A continuación trataremos con más detalle las capas que se han utilizado en mayor grado en este TFC. Éstas son CsmaC, UniqueSend, UniqueReceive, PacketLink, ActiveMessageP y TransmitP/ReceiveP.

B.2. Capa CsmaC (CSMA/CA)

La capa CsmaC (CSMA/CA) del chip cc2420 se encarga de la implementación del control del acceso al medio, es decir, implementa el protocolo CSMA.

B.2.1. Radio Backoff

Un backoff es un período de tiempo que el sistema radio espera antes de intentar transmitir. Cuando el sistema radio necesita aplicar un backoff, puede elegir entre tres periodos de backoff: initialBackoff, congestionBackoff, y lplBackoff.

Estos períodos son las ventanas dentro de las cuales se elige un valor. Estos se implementan a través de la interface RadioBackoff, que señala una petición de backoff para especificar el período backoff. Los componentes que están interesados en el ajuste del periodo de backoff pueden usar los comandos de la interface RadioBackoff. Esto permite a múltiples componentes que escuchan expresamente para adaptarse ajustar el periodo de backoff para los paquetes.

Con un menor periodo de backoff, tendremos una transmisión más rápida y lo más probable es que el transmisor ocupe excesivamente el canal. Los períodos de backoff deberían ser tan aleatorios como fuera posible para prevenir que dos transmisores accedan al canal a la vez. Se detallan a continuación.

InitialBackoff es el período de backoff más corto, requerido para el primer intento de transmisión de un paquete.

CongestionBackoff es un periodo más largo que se usa cuando se encuentra ocupado el canal o hay una colisión. Al usar un período de backoff más largo en este caso, es menos probable que el transmisor ocupe excesivamente el canal o que haya una colisión.

LplBackoff es el periodo de backoff usado para paquetes que se entregan con baja potencia de escucha. La escucha de baja potencia es una técnica de muestreo del canal que se basa en intervalos de muestreo muy cortos por parte del receptor, mientras que el emisor antes de enviar un paquete antepone un preámbulo por lo menos tan duradero como el periodo que transcurre entre los pequeños intervalos de muestreo de los receptores, para así asegurarse que todos ellos escucharán el preámbulo y ponerse en marcha. Esto aumenta la eficiencia en el consumo de energía. Se puede encontrar más información acerca de este mecanismo en [8, 1].

B.2.2. Comprobación de canal libre (CCA, Clear Channel Assessment)

Por defecto la capa CSMA de la pila radio optimiza y realiza una comprobación de si el canal está libe o no (CCA) antes de transmitir. Si el canal no está libre, se espera un período aleatorio de back-off, antes de realizar un nuevo intento de transmisión. El chip cc2420 en sí mismo proporciona una orden de estrobe para transmitir el paquete si el canal está disponible. Para decidir si se transmite o no con la comprobación de si el canal está libre u ocupado. Por defecto cada paquete se transmitirá con el CCA habilitado.

Si las capas por encima del CSMA quieren deshabilitar la evaluación del canal antes de la transmisión, deben interceptar el evento *RadioBackoff. requestCca(...)* para ese mensaje y usar en el cuerpo del evento el comando *RadioBackoff. setCca (FALSE).*