Universidad de la República
Facultad de Ingeniería

Pedro Moreira Pérez
Fernando Payret Arbiza
Leonardo Pendás Urgoiti

Tutor
Leonardo Steinfeld

Co-tutor
Pablo Mazzara

xx de mayo de 2009

Agradecimientos

Los integrantes del grupo quisiéramos agradecer especialmente a las siguientes personas:

A Leonardo Steinfeld y Pablo Mazzara por su invalorable colaboración como tutores del proyecto.

A Adriana Coitinho, y la familia Pendás Urgoiti por su amable hospitalidad y apoyo.

A Leticia Cirlinas por los consejos a la hora de la documentación.

A todos los familiares y amigos que con sus amables aportes ayudaron a que se realizara este proyecto.

Resumen

En el presente trabajo se describe el diseño e implementación de una red de sensores inalámbricos de bajo consumo, para su uso en el medio agrícola.

Los pilares de esta implementación son el Standard IEEE 802.15.4 y la plataforma hardware TmoteSky. Considerando cuatro de las capas del modelo OSI, aplicación, red, enlace de datos y física; se realizó el diseño e implementación de las dos primeras. En cambio, para las dos capas inferiores se utilizaron las desarrolladas por el grupo Hurray de la Universidad de Porto, adaptándolas a las necesidades del presente trabajo.

Todo el desarrollo de esta implementación estuvo enmarcado dentro del objetivo de lograr un tiempo de vida de un año con dos pilas AA por dispositivo.

Estructura de la documentación

A continuación se describe el contenido de la presente documentación:

- **Capítulo 1 – Introducción.** Incluye motivación, marco del proyecto, antecedentes, objetivos y solución propuesta.

- **Capítulo 2 – Hardware y software.** Se introduce el hardware a ser utilizado durante todo el desarrollo del proyecto. Se explican conceptos sobre el lenguaje utilizado para programar estos dispositivos, su filosofía y sus diferencias con otros lenguajes clásicos de programación de sistema embebidos. También se exponen nociones básicas sobre el sistema operativo a utilizar.

- **Capítulo 3 – Protocolo de acceso al medio.** Se listan las funcionalidades que se buscan contemplar en los protocolos de acceso al medio para este tipo de redes. Se menciona el agregado de sincronización, el cual lo diferencia con protocolos de otro tipo de redes, y se describe el protocolo MAC utilizado en la aplicación. Por último se explica la implementación MAC utilizada de la cual se partió en el desarrollo y los resultados obtenidos en las pruebas de estas.

- **Capítulo 4 – Componentes Principales de bajo nivel.** Se mencionan los componentes utilizados como punto de partida a la hora de comenzar con el desarrollo e implementación del proyecto. Se menciona parte del proceso de pruebas a los cuales fueron sometidos, y los cambios necesarios para poder utilizarlos correctamente desde la aplicación desarrollada.

- **Capítulo 5 – Diseño de la aplicación.** Se tratan todos los temas referidos al proceso de diseño de la aplicación. Se comienza explicando las funcionalidades principales que esta capa debe cumplir, para luego definir los casos de uso. Se describen las distintas estructuras de datos empleadas en esta capa y se finaliza analizando la implementación.

- **Capítulo 6 – Diseño entidad de red.** Temas referidos al proceso de diseño de la red. Se comienza explicando la primera etapa de desarrollo que consistió en el estudio de los casos de uso. Luego se explica el proceso de selección de la topología de red, y por último el desarrollo de la aplicación, explicando la estructura de datos y la implementación en sí.

- **Capítulo 7 – Análisis de consumo.** Se muestra un análisis de consumo en base a los diferentes parámetros de la red para mostrar la dependencia de la duración de las pilas con los mismos.

- **Capítulo 8 – Pruebas generales.** El objetivo de este capítulo es resumir todo el proceso de pruebas al cual fue sujeta la aplicación y la entidad de red diseñada. Se realiza una introducción sobre el ambiente de test, y los resultados obtenidos en cada etapa.
- **Capítulo 9 – Balance y conclusiones.** En este capítulo se exponen resultados y reflexiones referentes al desarrollo del proyecto, lo cumplido y lo no cumplido respecto a lo propuesto en primera instancia. También se describen mejoras a realizarse en la aplicación que no se implementaron porque o bien excedían el alcance de este proyecto, o por no contarse con los recursos necesarios en el hardware.

Contenido

Capítulo 1

Introducción

1.1. Motivación

El interés principal de este proyecto surge por parte del Grupo de Microelectrónica del Instituto de Ingeniería Eléctrica. La idea presente se centra en el desarrollo actual de las redes de sensores inalámbricos y la aplicación de esta tecnología en el medio local.

La aplicación principal de este tipo de tecnologías se remite al sector agropecuario del país, contemplando el problema de brindar a los agricultores la posibilidad de obtener ciertas medidas de magnitudes ambientales relevantes de sus plantaciones, como lo son la temperatura ambiente, humedad del suelo, etc. y mantener un registro histórico de estas variables. De esta forma, el agricultor será capaz de monitorear la plantación en su totalidad, y podrá tomar determinadas medidas según lo amerite, tales como fumigaciones, riegos, etc. optimizando así la producción.

Este proyecto de ingeniería surge con el fin de desarrollar una red de sensores inalámbricos, de tal forma de brindar una solución al problema descripto anteriormente.

1.2. Marco del Proyecto

En el marco de la línea de investigación sobre redes de sensores inalámbricos aplicado a la industria agropecuaria, el Grupo de Microelectrónica ha desarrollado proyectos con otros organismos para adaptar y aplicar estas tecnologías al agro. Tal es el caso del proyecto Wiseman (PDT S/C/OP 69/08), el cual está en proceso de culminación, en conjunto con el Centro Regional Sur (CRS) ubicado en Juanicó, perteneciente a las Facultades de Agronomia y Veterinaria de la Universidad de la República. En el marco de dicho proyecto se realizaron dos proyectos de fin de carrera Siagro1 (2006) y Siagro2(2007). En este momento está comenzando la ejecución de otro proyecto, SIMPA (FPTA-INIA Nº 280), para la apliación de estos sistema en plantaciones citrícolas.

Adelantándose a la ejecución del mencionado proyecto y continuando la línea de investigación previa, se plantea el presente proyecto de fin de carrera para explorar otras soluciones a las adoptadas con anterioridad para la aplicación de las tecnologías de redes de sensores inalámbricos. El caso particular se remonta a una plantación de naranjos en el departamento de Salto.

El objetivo principal es entonces tener en determinados lugares del campo diferentes sensores que funcionen ininterrumpidamente tomando datos de magnitudes ya mencionadas anteriormente, y que estos datos sean reportados a una central. Además de estos requerimientos, se debería tener presente que no iba a resultar posible un cableado entre estos sensores (tanto de alimentación como para datos) por lo cual el medio de transmisión debería de ser totalmente inalámbrico.

Los motivos mencionados anteriormente dan marco al proyecto que se describe a lo largo de todo este documento, el cual consiste en el diseño, implementación y validación de una red de sensores inalámbricos.

1.3. Redes de sensores inalámbricos

En esta sección se muestra resumidamente el paradigma de las redes de sensores inalámbricos[1].

En los últimos años, las redes de sensores han estado formadas por un pequeño número de nodos conectados por cable a una estación central de procesado de datos. Hoy en día, las tecnologías han permitido centrarse más en redes de sensores distribuídas e inalámbricas. Los últimos avances tecnológicos han posibilitado el desarrollo de dispositivos diminutos, baratos y de bajo consumo, que son capaces tanto de procesar información localmente así como también comunicarse de forma inalámbrica. Además, en muchos casos, se requieren muchos sensores para evitar obstáculos físicos que obstruyan o corten la línea de comunicación.

Cada nodo de una red inálambrica consta de un dispositivo con microcontrolador, sensores y transmisor/receptor, y forma una red con muchos otros nodos, también llamados motes o sensores. Al coordinar la información entre un importante número de nodos, éstos tienen la habilidad de medir un medio físico dado, con gran detalle. Con todo esto, una red de sensores puede ser descrita como un grupo de motes que se coordinan para llevar a cabo una aplicación específica.

El medio que va a ser monitorizado no tiene infraestructura ni para el suministro energético, ni para la comunicación. Por ello, es necesario que los nodos funcionen con baterías y que se comuniquen por medio de canales inalámbricos.

1.4. Requerimientos y objetivos generales

A continuación se listan los objetivos principales buscados en este proyecto y los criterios de aprobación de este.

1.4.1. Objetivos generales

El objetivo general es lograr diseñar, implementar y testear exitosamente una red de sensores inalámbricos. Se deberá tener presente que una vez funcionando la red correctamente, las pilas que alimentan los dispositivos hardware que componen la red, deben durar al menos un año.

[1] Para obtener más detalles, se puede consultar el documento [1].

Otro requerimiento es sobre el protocolo a utilizar a nivel de capa de acceso al medio, el Standard IEEE 802.15.4.

1.4.2. Criterios de éxito y validación

1.4.2.1. Red implementada, probada y validada

Para cumplir con el objetivo principal de lograr un correcto funcionamiento de la red, se plantea como objetivo realizar una prueba en el predio agrícola. De no ser posible ésta por dificultades de traslado o coordinación entre tutores y/o estudiantes, se realizaría la prueba de aceptación con una red cuya topología sea idéntica a la de la aplicación final, con una distribución geográfica similar y una cantidad máxima de diez nodos.
Durante la prueba se deberá verificar la correcta recepción de los datos transmitidos por cada nodo de la red hacia el nodo sumidero. Dichos datos incluirán la medida de la variable ambiental en cuestión (temperatura) e información adicional vinculada al trayecto de cada paquete por la red.
Se realizarán cambios en los parámetros de operación del sistema (período de muestreo, etc.) verificando que dichos cambios se reflejan -visualizando los datos recibidos- en el funcionamiento de la red.
Se deberá minimizar la pérdida de paquetes punta a punta. Esto podrá comprobarse mediante un número de secuencia único por paquete por nodo y marcas de tiempo (time stamp) asociado a los datos en relación a la cadencia de muestreo de la variable u otro método similar.

1.4.2.2. Lograr un tiempo de vida útil para la red

El tiempo de vida útil se refleja en la duración de las pilas del mote. En primera instancia se fija un tiempo mínimo de un año utilizando dos pilas alcalinas AA (carga aprox. 2700 mAh[2]). Para verificar esta especificación se realizará la medida del consumo de un nodo en una red funcionando en condiciones equivalentes a las especificadas en el punto anterior durante 48 hs continuadas. En base a esta medida se estimará el consumo durante un año verificando que la carga extraíble de dicha pila es suficiente para la correcta operación de un nodo.

1.5. Antecedentes

1.5.1. Proyectos anteriores, similitudes y diferencias

En el Instituto de Ingeniería Eléctrica se han llevado a cabo proyectos de fin de carrera referentes al tema de redes de sensores inalámbricos y actualmente existen proyectos en vías de desarrollo en este ámbito. A saber,

Siagro1 y Siagro2: Proyectos de redes de sensores inalámbricos desarrollados e implementados por el grupo de microelectrónica del IIE. Estas redes desarrolladas tienen diferencias en cuanto al protocolo de acceso al medio utilizado.

[2]La plataforma hardware utilizada cuenta con espacio disponible para dos pilas AA

RIBC: Proyecto de grado en redes de sensores inálámbricos desarrollado en el mismo período en que se realizó este trabajo, de características similares en cuanto a objetivos buscados. Consiste en la validación de una red, diferenciándose en los protocolos de comunicación empleados.

SIMSI: Proyecto de grado, orientado a la administración de varias de estas redes, a través de una interfaz web para PC.

Existe revisión de los diferentes protocolos o modos de comunicación desarrollada por investigadores del grupo Hurray[2] del Instituto Politécnico de la Universidad de Porto. Se encuentra el paper escrito por los creadores del código de partida de implementación de la red [3], que trata los paradigmas en la comunicación de redes de sensores inalámbricos. Allí se listan los protocolos tradicionales como lo son *scheduling-based (TDMA), collision-free(CDMA, FDMA) y contention-based(CSMA/CA, MACA)* sus ventajas y limitaciones; y los criterios en la elección de estos en función de la aplicación deseada. Entre estos criterios se listan gestión de energía, tiempos de respuesta, y escalabilidad a la hora del ingreso de nodos a la red. Se presentan protocolos que han surgido a partir de las bases de estos protocolos tradicionales, como respuesta a los criterios antes mencionados. Estos son *LEACH, S-MAC, DB-MAC, IEEE 802.15.4*

Existen también los protocolos ZigBee[7], que completan las capas de comunicación MAC (IEEE 802.15.4) hasta la aplicación. En principio no sería adecuado para aplicaciones similares a las nuestras ya que consumen demasiado (en general se supone que algunos nodos están alimentados desde la red).

Las redes de sensores inalámbricos tienen pocos años de existencia. A pesar de ello, ya existen varios fabricantes trabajando en esta tecnología[3].

1.5.2. Punto de partida, implementación grupo Hurray

El grupo Hurray del Instituto Politécnico de la Universidad de Porto [2] implementan la comunicación a nivel de capa de acceso al medio, según el Standard 802.15.4 en el lenguaje nesC, para el sistema operativo TinyOS y en la plataforma hardware telosb utilizada durante el proyecto. Este código se encuentra disponible (open Source) en la web de la Universidad de Porto. Este fue nuestro punto de partida, del cual se comenzaron a testear funcionalidades básicas, entender el diseño, su funcionamiento y las correcciones necesarias para nuestra aplicación. Sin embargo la solución al problema que se intenta resolver no se encuentra disponible, ya que se requieren funcionalidades en cuanto a nivel de red y de aplicación, no contempladas en el trabajo aquí mencionado.

[3]Entre otros:
CROSSBOW:Especializada en el mundo de los sensores, desarrolla plataformas hardware y software que dan soluciones para las redes de sensores inalámbricas. Entre sus productos encontramos las plataformas Mica, Mica2, Micaz, Mica2dot, telos y telosb.
MOTEIV: Ha desarrollado la plataforma TmoteSky y TmoteInvent. El tipo de mote Tmote Sky el cual será el utilizado durante el proyecto.

1.6. Solución propuesta

La solución propuesta frente al problema planteado fue utilizar una red de topología Cluster Tree. En esta configuración, existe un nodo raíz o sumidero que se encarga de recibir la información de todos los nodos de la red. Este nodo se encuentra conectado a un PC y es allí en donde se registran los datos recolectados. Para ello, cada nodo debe establecer un enlace con el sumidero de forma tal de poder enviarle los datos obtenidos de sus sensores. Al necesitar cubrir un predio de mayor tamaño que el del alcance de transmisión/recepción de los nodos, surge la necesidad de poder establecer un enlace virtual entre el sumidero y los otros nodos de la red. Es por esto que cada nodo, además de cumplir con la función de sensar y enviar los datos, debe cumplir con la funcionalidad de router.

Para llevarlo a cabo, cada nodo debe mantener un enlace físico con el nodo que se encuentra por "encima" (más cerca del sumidero) de él, en el sentido de que este último es quien conoce la ruta al sumidero, o cuál es el enlace de salida que le permita llegar al sumidero.
Esta forma de enviar los datos utilizando otro nodo de la red como puente, es lo que se conoce con el nombre multihop o multisalto. El paquete a nivel de capa de red "salta", transmitiéndose desde un nodo de la red hacia otro, de forma de establecer una comunicación virtual con otro nodo fuera de su alcance de transmisión. Por ello, esta funcionalidad es la única que permitiría en primera instancia poder comunicar cualquier dispositivo en la red con el sumidero.

Cada nodo será capaz de permitirle el servicio de router a otros dispositivos, fijando un máximo de 7 clientes. A su vez cada uno de estos nodos clientes deberá proveer al mismo tiempo el mismo servicio a otros 7 nodos que desen ingresar en la red, formandose así una estructura de rama.

Para poder permitir la funcionalidad de multisalto o router en todos los dispositivos de la red, es necesario manejar dos enlaces MAC en dos canales físicos diferentes. Uno de estos enlaces se encarga de la comunicación con los nodos que se encuentren "por encima" de él, en los cuales el nodo es cliente de los servicios que este le ofrece, esto permite enviar datos propios o de otros nodos que envían a través de él, al sumidero. En el otro enlace, el nodo se encontrará como proveedor del servicio de envío de datos al sumidero para futuros nodos que ingresen a la red, y se encargara de recibir los datos que estos le envíen.

Cada uno de estos enlaces se corresponde con una red especificada por el protocolo del Standard IEEE 802.15.4, cada una totalmente independiente de la otra. Estas redes se caracterizan por la sincronización que debe existir entre los dispositivos de la red, para que no esten estos accediendo al canal durante todo el tiempo, sino que sean períodos cortos los de actividad en comparación con los de inactividad o de no acceso al canal. La necesidad de de que se definan estos dos períodos reside en el requerimiento de bajo consumo, ya que el mayor consumo de energía se da en los períodos de actividad o acceso al canal. Este es el argumento principal que justifica el uso de este tipo de protocolos de sincronización. Al requerirse dos enlaces 802.15.4, es que debe de existir una doble sincronización, una en cada enlace.

Capítulo 2

Hardware y software de desarrollo

En este capítulo se introduce el hardware a ser utilizado durante todo el desarrollo del proyecto. Se explican conceptos sobre el lenguaje utilizado para programar estos dispositivos, su filosofía y sus diferencias con otros lenguajes clásicos de programación de sistema embebidos. También se exponen nociones básicas sobre el sistema operativo a utilizar.

2.1. Descripción del hardware utilizado

Cada nodo de la red está compuesto por una plataforma hardware, en donde se encuentra programada la aplicación final expuesta en este trabajo una vez estando la red en funcionamiento. Durante el proceso de pruebas y desarrollo del proyecto se utilizó en todo momento este mismo hardware.
A continuación se explican los conceptos básicos que caracterizan estos dispositivos.

2.1.1. Datos generales

Estos dispositivos hardware mencionados en el párrafo anterior reciben el nombre de TmoteSky. Constan de los elementos básicos requeridos para un ser un nodo en una red de sensores inalámbricos, estos son: sensores, transceiver inalámbrico y microcontrolador. Tiene incorporada también una interfaz USB para ser programado desde el PC y poder intercambiar datos con este. Se le puede incorporar portapilas, antena (además de la que viene integrada en la placa) y sensores de luminiscencia, humedad, etc..
Estos motes fueron provistos gracias a una compra realizada por el Instituto de Ingeniería Eléctrica. La siguiente figura muestra el dispositivo TmoteSky, el cual es diseñado exclusivamente para las redes de sensores inalámbricas de bajo consumo.

Figura 2.1: Plataforma hardware TmoteSky

A continuación se realiza una breve descripción de los atributos de esta plataforma [8]:

- Transceiver inalámbrico de hasta 250kbps en la banda 2.4GHz compatible con el Standard IEEE 802.15.4 fabricado por Chipcon.
- Microcontrolador MSP430 F1611 de 8MHz (10k RAM, 48k Flash)
- ADC, DAC , y controlador de memoria dinámico (DMA)
- Antena integrada en la placa con rango de 50m indoors 125m outdoors
- Posibilidad de integrar sensores de temperatura, humedad y luz
- Consumo ultra bajo
- Tiempo de transición de modo sleep a modo activo menor a 6 μ s
- Programación e interfaz de transferencia de datos vía USB

Otra información de interés relevante a nivel de datasheet para nuestra aplicación fue la siguiente tabla, en donde se muestra el consumo de todo el hardware completo para los diferentes modos de operación, discriminando según los estados del microcontrolador y el transceiver.

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 2.2: Diferentes valores de consumo para los diferentes estados del transceiver y microcontrolador

2.1.2. Transceiver CC2420

En conjunto con el microcontrolador y los sensores, este es uno de los elementos más importantes que componen el hardware integrado. Este hardware es el encargado de manejar la antena que posee integrada la placa del mote. Es un hardware diseñado por la compañía ChipCon [9], exclusivamente para funcionar bajo la normativa del Standard IEEE 802.15.4. Es decir, operando en las mismas bandas de frecuencia e implementado las funcionalidades de capa física descriptas en la especificación del Standard. Este hardware se comunica con el microcontrolador a través de una interfaz SPI y es manejado desde el software del controlador a través del componente CC2420RecieveP y CC2420TransmitP, también ya disponibles dentro del paquete del grupo Hurray.

2.1.3. Sensores

El hardware fue diseñado por el fabricante para lograr una escalabilidad adecuada a la hora de incluir sensores en la red, en función de los requerimientos de la aplicación. Durante el diseño que se presenta en este proyecto se buscó continuar con esa filosofía. El diseño final de la aplicación permite obtener una fácil escalabilidad a la hora de integrar por software, la interfaz que maneja el nuevo sensor a incluir.
El único sensor que poseen los dispositivos directo de fábrica es el sensor de temperatura. Este es el único que se encuentra actualmente instalado en todos los motes, y sobre el cual se realizará todas las pruebas para la validación de la red.

2.1.4. Microcontrolador MSP430F1611

Además de las características listadas en la sección de descripción de la plataforma TmoteSky , el microcontrolador posee otras características que se describen a continuación

- Arquitectura RISC
- 5 Modos o estados diferentes llegando a un consumo de 1.1μA en modo StandBy y 0.2μA en modo Off(refresco de memoria RAM dinámica)
- Timer de 16-Bit con tres registros capture/compare
- USART0, o en modo asíncrono, UART, o SPI sincrónico o interfaz I2CTM(utilizada para la comunicación con el transceiver).

- USART1, o en modo asíncrono, UART, o SPI sincrónico o interfaz I2CTM(utilizada para la comunicación USB)
- Regulador de tensión.

2.2. Sistema operativo

En esta sección se explica los conceptos referentes al sistema operativo utilizado, TinyOS versión 2.1.
Este OS es generado (a nivel de código) cada vez que se compila la aplicación, y es cargado en la ROM del microcontrolador cada vez que éste se programa. Es event-driven, en el sentido que funciona a partir de eventos producidos que llamarán a funciones. Ha sido desarrollado exclusivamente para redes de sensores con recursos limitados. El entorno de desarrollo soporta la programación de diferentes microprocesadores y permite diferenciarlos mediante identificadores, es decir, se puede compilar en diferentes plataformas. Todas sus librerías y aplicaciones están escritas en nesC Además de lo nombrado, TinyOS tiene las siguientes características:

- pequeño núcleo de 400bytes entre código y datos
- arquitectura basada en componentes
- capas de abstracción bien establecidas, limitadas claramente a nivel de interfaces, a la vez que se pueden representar los componentes automáticamente a través de diagramas
- amplios recursos para elaborar aplicaciones
- adaptado a los recursos limitados de los motes: energía, procesamiento, almacenamiento(gran limitante en el hardware usado en el proyecto) y ancho de banda
- operaciones divididas en fases (Split-phase)

Por mas información, consultar la wiki, y la web de los creadores [11]

2.3. Lenguaje de desarrollo utilizado, nesC

Aquí se resumen conceptos básicos que debieron ser adquiridos para la comprensión del lenguaje en el cual se desarrolló la aplicación. El estudio se basó fuertemente en el documento disponible en la wiki de tinyOS [12]

La gran diferencia entre la programación en el lenguaje nesC y otro lenguaje como por ejemplo C, es que nesC está pensado para interactuar con un sistema operativo, en nuestro caso tinyOS 2. En esta sección se describirán las principales características provistas por nesC que otros lenguajes no brindan.

- **interfaces**: Son definiciones de comandos y/o eventos, en las mismas se declaran los argumentos que tendrán los mismos. Las interfaces serán implementadas en los módulos que las provean.

- **Módulos**: Los módulos son implementaciones de código, que poseen variables globales, funciones, tareas, etc. Los módulos proveen interfaces, las cuales pueden tener comandos o eventos e implementa los comandos que provee. A su vez un módulo puede utilizar las interfaces que se implementan en otro módulo. Los módulos se comunican entre sí mediante los comandos y los eventos, un módulo llama a un comando implementado en otro módulo y este le puede responder con un evento (el evento esta implementado en el módulo que llamo al comando).

- **Tareas**: Son bloques de código similares a funciones, pero que el scheduler posterga para ejecutarse con menor prioridad. A las tareas no es posible pasarle argumentos, esa es la mayor limitante en el uso de las mismas, pero brinda la gran ventaja de que libera capacidad del microprocesador, ya que ejecuta en el momento en que se encuentra libre bloques de código de menor prioridad.

- **Comandos**: Son declarados en las interfaces e implementados en el módulo que provee la interfaz. Son invocados desde módulos que están conectados al módulo que los implementan, es decir que un mismo comando puede ser llamado desde diferentes módulos.

- **Eventos**: Son declarados en las interfaces e implementadas en el módulo que utiliza el módulo que las provee. Es decir, como respuesta a un comando invocado se puede disparar un evento o en el módulo que provee la interfaz un evento se dispara sin necesidad de un llamado externo. Por tal motivo el módulo principal que utiliza al módulo secundario debe implementar la respuesta al evento disparado.

Capítulo 3

Protocolo de acceso al medio

En este capítulo se introducen las funcionalidades que se buscan contemplar en los protocolos de acceso al medio para este tipo de redes. Se menciona el agregado de sincronización, el cual lo diferencia con protocolos de otros tipos de redes, y se describe el protocolo MAC utilizado en la aplicación. Por último se explica la implementación MAC utilizada de la cual se partió en el desarrollo y los resultados obtenidos en las pruebas de éstas.

3.1. Principal distinción con otros proyectos

Existen varios proyectos en el área de redes de sensores inalámbricos ejecutándose en paralelo con el que se presenta en este documento. La principal diferencia del diseño desarrollado a lo largo de este proyecto, con los abordados por otros grupos, es el protocolo utilizado para acceso al medio de radio frecuencia. Si bien todos éstos están orientados a minimizar el consumo de los motes, existen diferencias a la hora de la elección del protocolo, según el interés deseado. Se listan a continuación los conceptos a tener en cuenta a la hora de elegir el protocolo MAC a utilizar y cuales de estos criterios influyen en la aplicación que contempla este trabajo [1]:

[1] Cabe destacar que no se hizo la selección del protocolo en función de las necesidades de la aplicación, sino que el protocolo a utilizar fue uno de los requerimientos impuesto por los tutores.

Criterio	*¿Es requerido en la aplicación?*
Escalabilidad a la hora de añadir nodos, para redes de sensores dinámicas	Sí. La aplicación debe permitir agregar nodos desde cualquier punto que esté al alcance de un nodo, estando éste ya dentro de la red.
Acotar tiempos de demora	No. Si bien es importante la confiabilidad y la robustez en el envío y recepción de los datos, los tiempos de demora no son un requerimiento
Minimizar las sobrecargas en la red	Sí. Es necesario para el eficiente uso de la energía disponible.
Capacidad para la movilidad de los dispositivos	No. La aplicación no contempla dispositivos móviles.

Tabla 3.1: Criterios en la elección de protocolos MAC y su influencia en la aplicación desarrollada.

3.2. Necesidad de un protocolo para la sincronización

Debido al fuerte requerimiento en el bajo consumo de los nodos de la red, se requiere un uso eficiente de los dispositivos que realizan el mayor uso de la energía disponible. Tal es el caso del transceiver, componente de mayor consumo en todo el hardware a manejar, por lo cual su uso debe ser óptimo. Para ello, debe existir un protocolo que se encargue de manejar este componente de manera tal de asegurarse que al encenderse para transmitir información, se encienda al mismo tiempo el transceiver del otro mote receptor. Por los motivos mencionados anteriormente, es que debe existir cierta sincronización entre los nodos de la red a la hora del acceso al medio inalámbrico. Esto es, entre otros, que exista una descripción de tiempos de sincronización entre nodos en la red para poder cumplir con tal objetivo. Actualmente esta descripción se encuentra estandarizada por la IEEE [4]. En ella se encuentra todo lo referente a nivel de acceso al medio, su interfaz con una capa física (la cual se encarga del correcto manejo del transceiver) y la interfaz que ésta provee a la aplicación.

3.3. Descripción del protocolo IEEE 802.15.4

Esta especificación fue objeto de estudio a lo largo de todo el desarrollo del proyecto, ya que este proyecto se centró fuertemente en ella y se partió de código que implementaba, en parte, las interfaces allí descriptas. El protocolo está basado en el modelo de capas (OSI) y allí se especifica para las dos capas (Física y Mac):

- Requerimientos de funcionalidades.
- Interacción e intercambio de información entre ellas mediante el uso de interfaces.
- Servicios que ofrecen, y servicios que utilizan.
- Formato de tramas, descripción de todos los campos.
- Variables y constantes requeridas y fundamento de que existan.

A nivel de capa física se especifica:

- Frecuencias y canales permitidos.
- Modulaciones en estos canales.
- Detección de energía de canales (ED[2]) e indicador de calidad del enlace (LQI[3]).
- Especificaciones del transceiver, activación y desactivación de éste.
- Transmisión y recepción de datos.

A nivel de capa MAC:

- Sincronización y comportamiento temporal de la red.
- Algoritmos de un acceso eficiente al medio (CSMA).
- Soporte en la seguridad de la red (fuera del alcance del trabajo abordado).

Lo más importante a resaltar son los protocolos para la sincronización. A continuación se explica de manera resumida el comportamiento a nivel MAC allí descripto. Para ello, antes se debe mencionar como va a ser la configuración de los diferentes nodos en la red. Existen dos tipos de dispositivos dentro de la red, los coordinadores (coordinators) y los dispositivos finales (end devices). Los primeros son los encargados de gestionar la red, es decir son quienes realizan la descripción de los tiempos de actividad e inactividad para que el resto de dispositivos finales puedan sincronizarse a éste. Un posible esquema de lo antes mencionado se muestra en la figura:

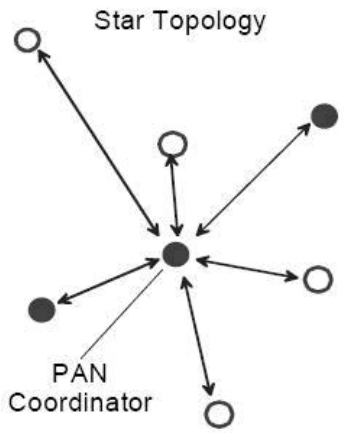


Figura 3.1: Estructura de red estrella (star) a nivel de capa MAC. Figura extraída de [4].

[2]Energy Detection: Mecanismo que permite detectar actividad en un canal en función de los niveles de energía de la señal obtenida

[3]Link Quality Indicator: Es un indicador sobre la calidad en la recepción de los datos

El Standard hace la distinción entre dispositivos con todas las funcionalidades (full function device FFD, en negro en la figura) y dispositivos con funcionalidades reducidas (RFD, en blanco en la figura). En la figura se muestra como los diferentes dispositivos que no son coordinadores se comunican únicamente con el coordinador para esta topología de estrella, que es la usada durante todo el proyecto. El Standard también especifica otros tipos de topología como Peer-to-peer. Lo más importante a destacar es el comportamiento de la red en el tiempo. Como se mencionó a comienzos del capítulo 3.2, existe un período de inactividad (nodos en bajo consumo) y otro de actividad(estado de mayor consumo). El Standard especifica que estos períodos pueden variar en el tiempo, y sus duraciones las especifica el coordinador en tramas que envía periódicamente llamadas tramas beacon. La siguiente figura muestra este comportamiento:

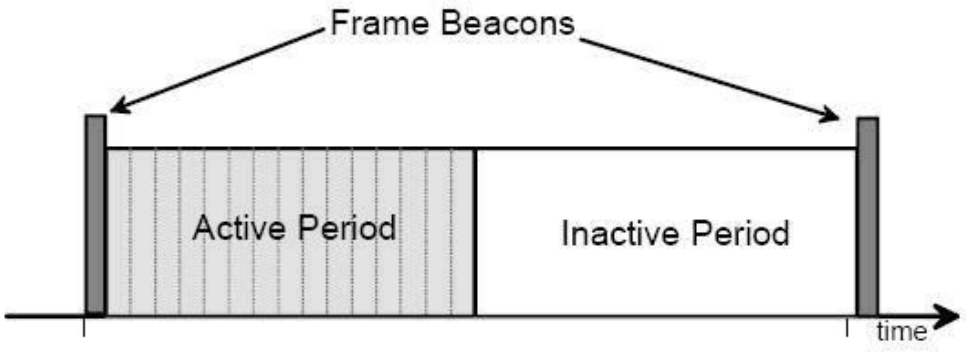


Figura 3.2: Esquema de tiempos en una red IEEE 802.15.4. Figura extraída de [4].

El período de actividad (Active Period) está compuesto por un período de contención de acceso (en donde los dispositivos acceden al medio utilizando CSMA) y un período de tiempos garantidos para el envío de datos (en el que los dispositivos envían datos sin realizar ningún control sobre el estado en el que se encuentra el canal). En cuanto a las especificaciones de capa física, se lista el conjunto de los canales que van desde los 868 hasta los 2400 MHz y las modulaciones permitidas en ellos.
Son un total de 27 canales (del 0 al 26), y el Standard los divide en 3 bandas de frecuencia:

- Banda de 868.3 MHz. Canal 0.
- Banda de 915 MHz. Con canales que se centran en las frecuencias $F_c = 906 + 2$ (k - 1) en MHz, con k = 1, 2, , 10
- Banda de 2450 MHz. Con canales que se centran en las frecuencias $F_c = 2405 + 5$ (k - 11) en MHz, con k = 11, 12, , 26

El transceiver utilizado trabaja en la banda de 2450, en los 16 canales mencionados. A continuación mostramos la información para esa banda de frecuencia.

MHz	*Tasa chip (kchip/s)*	*Modulación*	*Tasa Bit (kb/s)*
2400-2483.5	2000	O-QPSK	250

Tabla 3.2: Especificaciones de capa física del Standard y contempladas por el chip CC2420

También se especifica el formato de trama física, el diagrama de bloques para la correcta modulación y demodulación, formas de pulso de modulación y valores para los chips [4] de transición.

3.4. Implementación provista por grupo Hurray

El grupo de investigadores Hurray de la Universidad de Porto, implementa el Standard utilizando TinyOS versión 1. Posteriormente migraron su implementación a la versión 2.1. La descripción completa de este diseño se encuentra disponible en el documento [5] el cual sirvió de guía a la hora de comprender la implementación, y realizar el testeo de la misma. Este trabajo consta con todas las funcionalidades de la capa física implementadas, y a nivel MAC se listan las funcionalidades implementadas (todas solidarias al Standard IEEE):

- Algoritmo CSMA/CA - versión ranurada;
- Mecanismo GTS[5];
- Transmisión de datos de manera:
  - Directa(a través del algoritmo CSMA)
  - Indirecta[6]
  - A través de GTS;
- Gestión de tramas beacon;
- Construcción y manejo de todo tipo de tramas MAC para direccionamiento corto;
- Mecanismo de asociación, desasociación;
- ED y PASSIVE SCAN de los diferentes canales;

[4]Cada símbolo (medio Byte) se representa en una palabra de 32 bits (chip), utilizando un algoritmo similar al CDMA

[5]Guaranteed Time Slots o ranuras de tiempo garantidas. Son ranuras de tiempo limitadas dentro del período activo(ver3.2) en donde los datos se envían sin el algoritmo de CSMA ya que es sabido de antemano quién tiene asegurada cada una de estas ranuras para el envío o recepción de datos

[6]Mecanismo de transmisión de datos a través de tramas de comando *data request*. El coordinador notifica en el beacon para que dispositivos tiene datos, y a medida que los dispositivos se encuentren disponibles para recibirlos, los solicitan enviando tramas *data request*. Por información mas detallada, ver [4]

Para ubicar al lector en contexto, se muestra en el siguiente esquema el punto de partida.

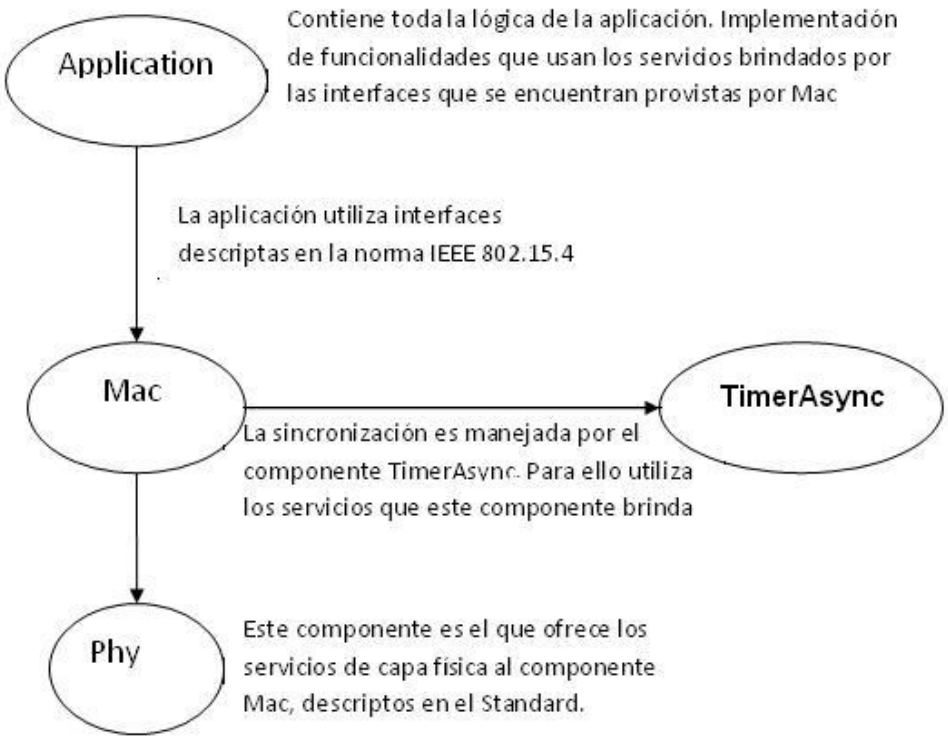


Figura 3.3: Esquema de arquitectura Hurray

Se destacan tres grandes componentes, Mac, Phy, y TimerAsync. El componente Mac es quien implementa las funcionalidades mencionadas en la lista anterior, y es el desarrollo principal que contiene la arquitectura que utilizamos como punto de partida. Estos componentes se interconectan a través de interfaces, como se mencionó en el capítulo anterior de nesC. Es decir, los componentes por sí solos implementan funcionalidades a través de interfaces, y utilizan interfaces de otros componentes. Existen archivos de configuración que indican qué componente se encuentra unido o cableado a qué otro.

El listado de interfaces que se usan o implementan, son archivos que describen eventos y comandos que:

- El componente que los provee(provides), debe tener implementadas como funciones todos los comandos(command) que se describen en ese archivo de interfaz y puede disparar los eventos(event) que allí se describen.

- El componente que los usa(uses) tiene la posibilidad de llamar cualquier comando (command) que se describe en esa interfaz y debe tener definidos todos los eventos(event) que se disparan en esa interfaz.

3.4.1. Selección de funcionalidades ya implementadas para nuestra aplicación

Dentro del código provisto por el grupo Hurray, se realizó una selección de funcionalidades requeridas para la aplicación. Luego se procedió a testear estas funcionalidades para poder comenzar con el desarrollo de nuestra aplicación.

A continuación se listan, dentro de las interfaces provistas por el componente Mac (principal componente en esta implementación), las consideradas relevantes para el diseño. Los criterios principales que tenidos en cuenta en este proceso fueron los necesarios para tener la red funcionando en condiciones normales. Es decir, tener un correcto envío de datos; para una correcta inicialización de la red y permitir el ingreso de nuevos nodos que estén dentro del alcance de la misma, logrando así la escalabilidad requerida. También se da una descripción general de estas funciones.

- MLME_SCAN

  Esta interfaz es la que brinda el servicio de escaneo de los canales utilizados por la red. Esto se utiliza a la hora de conectar un dispositivo en la red, detectar si hay energía en algún canal y obtener la información que circula en ese canal[7]. Permite realizar un escaneo por ED (Energy Detection) para determinar el canal, y también un escaneo pasivo para localizar tramas beacon que contengan cualquier identificador de PAN[8].

```
interface MLME_SCAN
{
  command error_t request(uint8_t ScanType,
                          uint16_t ScanChannels,
                          uint8_t ScanDuration);

  event error_t confirm(uint8_t status,
                        uint8_t ScanType,
                        uint16_t UnscannedChannels,
                        uint8_t ResultListSize,
                        uint8_t EnergyDetectList[],
                        SCAN_PANDescriptor
                        PANDescriptorList[]);

}
```

Figura 3.4: Definición de la interfaz MLME_SCAN

Los parámetros que se pasan y se reciben por valor, son los mismos que se describen en la especificación del Standard: a continuación listamos los realmente utilizados en nuestra aplicación:

ScanType: Tipo del Scan. Nuestra aplicación utiliza ED para detectar los canales en donde hay actividad, y PASSIVE para obtener información de los beacon que se escuchen.

[7]Interfaz disponible en el Archivo MLME_DEVICE_SCAN.nc. Es implementado por MacDevice (MacDeviceM.nc)y utilizado por Red (RedM.nc). Estas mismas configuraciones se pueden observar en el archivo Red.nc

[8]Personal Area Network, sigla del Standard IEEE

ScanChannels: Es un bitmap de largo 16 en donde se especifican los canales físicos en donde se desee hacer el Scan. 0x0001 correspondiendo al canal 11 y 0xFFFF correspondiendo a los 16 canales de 11 a 26.

ScanDuration: Valor del orden del Scan[9].

ResultListSize: Es la cantidad de canales escaneados (no usado ni implementado, tiene un retorno fijo que vale 16, la cantidad de canales físicos)

EnergyDetectList: Para el caso de un ED scan se lista para cada canal, el máximo nivel de LQI obtenido por el transceiver de todas las tramas recibidas durante el tiempo en el que el transceiver se encuentra encendido escuchando dicho canal.

- MLME_BEACON_NOTIFY

Esta interfaz consta de un evento que se dispara en el componente MacM y que se corresponde con el arribo de una trama beacon, la cual contiene la especificación de los tiempos de sincronismo, como lo son, tiempos de inactividad, tiempos de envío garantidos, etc.[10] La declaración en nesC para este componente es:

```
interface MLME_BEACON_NOTIFY
{
    event error_t indication(uint8_t BSN,
                             PANDescriptor pan_descriptor,
                             uint8_t PenAddrSpec,
                             uint8_t AddrList,
                             uint8_t sduLength,
                             uint8_t sdu[]);
}
```

Figura 3.5: Definición de la interfaz MLME_BEACON_NOTIFY

Los parámetros que se pasan por valor, son los mismos que se describen en la especificación del Standard:

BSN: Es el número de secuencia de beacon que envía el coordinador, este se incrementa en cada beacon.

PanDescriptor: Esta estructura guarda información sobre la descripción del PAN, tales como la dirección del coordinador, el canal físico, el id del PAN, el beacon order y el superframe order. También se encuentra el campo SuperframeSpec en donde se guarda la información relevante al beacon, como largo de la supertrama, comienzo de los períodos de tiempo activo y demás.

PendAddrSpec: Se debería corresponder con la lista de dispositivos para los cuales el coordinador posee datos a enviarles. No implementado.

[9]Similar al beacon order y superframe order, el orden determina el exponente para el cual se calculará la duración en símbolos del scan. Ver [4]

[10]Interfaz disponible en el Archivo MLME_DEVICE_BEACON_NOTIFY.nc. Es implementado por MacDevice (MacDeviceM.nc)y utilizado por Red (RedM.nc). Estas mismas configuraciones se pueden observar en el archivo Red.nc

AddrList: Listado con las direcciones de dispositivos, para la transmisión indirecta. No implementado

sduLength: Largo del sdu beacon recibido.

sdu: Payload del sdu beacon recibido.

- MLME_ASSOCIATE

Esta interfaz es por la cual se asocia un dispositivo con un coordinador. La secuencia comienza cuando un dispositivo pide el servicio utilizando el comando *request* desde la aplicación; y esta solicitud es recibida por el coordinador con la llegada del evento *indication*. La aplicación del coordinador procesa el pedido y utiliza el comando *response* para notificarle al cliente sobre el resultado de la solicitud. Al dispositivo obtener la respuesta de la solicitud, lo notifica a la aplicación a través del evento que se dispara en la aplicación *confirm*. A continuación se detalla la declaración de esta interfaz en nesC prevista por el grupo Hurray[11].

```
interface MLME_ASSOCIATE
{
  command error_t request(uint8_t LogicalChannel,
                          uint8_t CoordAddrMode,
                          uint16_t CoordPANId,
                          uint16_t CoordAddress,
                          uint8_t CapabilityInformation,
                          bool SecurityEnable);

  event error_t indication(uint32_t DeviceAddress[],
                           uint8_t CapabilityInformation,
                           bool SecurityUse,
                           uint8_t ACLEntry);

  command error_t response(uint32_t DeviceAddress[],
                           uint16_t AssocShortAddress,
                           uint8_t status,
                           bool SecurityEnable);

  event error_t confirm(uint16_t AssocShortAddress, uint8_t status);

}
```

Figura 3.6: Definición de la interfaz MLME_ASSOCIATE

[11]Como MacDevice y MacCoord implementan la misma interfaz, estas se diferenciaron en dos archivos; y MLME_COORD_ASSOCIATE.nc. Es utilizado por la Red (RedM.nc). Estas mismas configuraciones se pueden observar en el archivo Red.nc. Para el caso del sumidero es solamente una interfaz, MLME_COORD_ASSOCIATE.nc ya que este maneja solo una interfaz MAC (MacCoord)

Los parámetros que se pasan por valor son:

LogicalChannel: El canal físico al cual se desea que el dispositivo se asocie.

CoordAddrMode: Utilizado en el caso que se contemple usar adicionalmente, otro tipo de direcciones que no sean la de 16 bit, como el Standard especifica de 64bit. En nuestra aplicación trabajamos siempre con direcciones de 16 bit.

CoordPANId: La PAN a la cual se desea asociar.

CoordAddress: La dirección del coordinador a la cual se desea asociar.

CapabilityInformation, SecurityEnable, ACLEntry: No implementadas ni utilizadas en la aplicación ya que son campos referentes a la seguridad de los datos en la red.

DeviceAddress: La dirección de 64 bit del dispositivo que desea asociarse y que posee antes de asociarse.

AssocShortAddress: Es la dirección que asigna el coordinador al dispositivo si desea tenerlo registrado y permitir su asociación.

Status: Resultado de la asociación (éxito, razón de falla, etc.)

- MLME_GTS

Esta interfaz es por la cual un dispositivo puede solicitar espacios de tiempos garantidos como se especifican en el Standard IEEE 802.15.4[4]. Este servicio es solicitado por un endDevice dentro de la PAN, y procesado por el coordinador de la misma manera que se describe en la interfaz de asociación. Estos tendrán un máximo de 7 time slots disponibles para asignar a los dispositivos end_devices que lo soliciten. La principal diferencia que se observa es la falta del evento de response, ya que el coordinador no manda un comando de confirmación, sino que la información que necesita el dispositivo final para conocer el resultado y avisar a su aplicación la encuentra en la trama beacon, ya que a partir de ésta, el dispositivo puede saber si se le fue correctamente cedido el tiempo de transmisión garantido.

```
interface MLME_GTS
{
  command error_t request(uint8_t GTSCharacteristics, uint8_t SecurityEnable);

  event error_t confirm(uint8_t GTSCharacteristics, uint8_t status);

  event error_t indication(uint16_t DevAddress,
                           uint8_t GTSCharacteristics,
                           uint8_t SecurityUse,
                           uint8_t ACLEntry);
}
```

Figura 3.7: Definición de la interfaz MLME_GTS

GTSCharacteristics: Este campo contiene información relevante al servicio, tal como el largo o duración del intervalo de tiempo solicitado, la dirección del flujo de datos, desde o hacia el coordinador, y si es una solicitud de alta o de baja del servicio.

SecurityEnable, ACLEntry: No implementados ni usados en la aplicación.

Status: Estado de la respuesta a la solicitud del servicio.

CoordAddress: La dirección del coordinador a la cual desea asociarse.

DevAddress: Dirección mac (macShortAddress) del dispositivo que realiza la solicitud del servicio. Este evento (indication) se dispara en la aplicación del coordinador y debe ser manejado por su lógica.

- MLME_START

Esta interfaz es por la cual un dispositivo coordinador arranca la PAN, es decir, comienza a emitir tramas beacon en un determinado canal físico, y arranca el timer de sincronismo (timerAsync).

```
interface MLME_START
{

  command error_t request(uint32_t PANId,
                          uint8_t LogicalChannel,
                          uint8_t BeaconOrder,
                          uint8_t SuperframeOrder,
                          uint8_t PANCoordinator,
                          uint8_t BatteryLifeExtension,
                          uint8_t CoordRealignment,
                          uint8_t SecurityEnable,
                          uint32_t StartTime);

  event error_t confirm(uint8_t status);

}
```

Figura 3.8: Definición de la interfaz MLME_START

PANId: Identificador de la PAN.

LogicalChannel: Canal físico en el cual se deberá comenzar a emitir beacons.

BeaconOrder: Determina la duración entre dos tramas beacon consecutivas.

SuperframeOrder: Determina la duración del tiempo activo de la red.

PANCordinator: Determina si el coordinador es un PAN coordinator o es solamente coordinador

BatteryLifeExtension, CoordRealignment, SecurityEnable, StartTime: Requerimientos considerados no pertinentes para la aplicación(la mayoría tampoco implementados). Por más información ver documento IEEE.

- MLME_SYNC

Es generada por la capa superior para solicitar la sincronización de un dispositivo final en una beacon_enabled PAN con el coordinador[12]. Se solicita al dispositivo escuchar en el canal, y al detectar una trama beacon, el device ajusta los parámetros del timer de tal forma de sincronisarse con la próxima.

[12]Interfaz disponible en el Archivo MLME_DEVICE_SYNC.nc. Es implementado por MacDevice (MacDeviceM.nc)y utilizado por Red (RedM.nc). Estas mismas configuraciones se pueden observar en el archivo Red.nc

```
interface MLME_SYNC
{

  command error_t request(uint8_t logical_channel,uint8_t track_beacon);

}
```

Figura 3.9: Definición de la interfaz MLME_SYNC

track_beacon: Si el parámetro Trackbeacon es seteado en TRUE, la MAC luego de encontrar el beacon, continúa buscando y procesando los próximos. En el caso que sea FALSE, procesa el primer beacon pero luego los descarta (no implementado, no por la aplicación).

- MLME_SYNC_LOSS

Esta interfaz posee solamente un evento que la capa Mac avisa a la aplicación al perder sincronización con el coordinador[13].

```
interface MLME_SYNC_LOSS
{
   event error_t indication(uint8_t LossReason);
}
```

Figura 3.10: Definición de la interfaz MLME_SYNC_LOSS

- MCPS_DATA

El comando de esta interfaz es generada por la aplicación local, para transferir un dato. Con los parámetros pasados, se forma la trama mac que se recibe en el evento .indication del dispositivo que se indicó como destinatario. Al transferirse el dato, se recibe el evento de confirmación (.confirm).

SrcAddrMode, DstAddrMode: Modos de dirección (16 o 64 bit). La implementación utilizada es de 16, por lo que se utilizó en todo momento del desarrollo direcciones de 16 bits.

SrcPANId, DstPANId: Direcciones de las PAN de origen y destino.

SrcAddr, DstAddr: Direcciones de origen y destino de los dispositivos.

msduLength: Longitud del msdu o carga útil MAC.

msdu: Bytes de Datos.

msduHandle: Puntero a los datos.

TxOptions: Opciones de transmisión (envío en GTS, de manera directa, indirecta, etc.)

mpduLinkQuality: Calidad de nivel de energía recibido en la trama.

[13]Interfaz disponible en el Archivo MLME_DEVICE_SYNC_LOSS.nc. Es implementado por MacDevice (MacDeviceM.nc)y utilizado por Red(RedM.nc). Estas mismas configuraciones se pueden observar en el archivo Red.nc

```
interface MCPS_DATA
{
  command error_t request(uint8_t SrcAddrMode,
                          uint16_t SrcPANId,
                          uint32_t SrcAddr[],
                          uint8_t DstAddrMode,
                          uint16_t DestPANId,
                          uint32_t DstAddr[],
                          uint8_t msduLength,
                          uint8_t msdu[],
                          uint8_t msduHandle,
                          uint8_t TxOptions);

  event error_t confirm(uint8_t msduHandle,
                        uint8_t status);

  event error_t indication(uint16_t SrcAddrMode,
                           uint16_t SrcPANId,
                           uint32_t SrcAddr[2],
                           uint16_t DstAddrMode,
                           uint16_t DestPANId,
                           uint32_t DstAddr[2],
                           uint16_t msduLength,
                           uint8_t msdu[100],
                           uint16_t mpduLinkQuality,
                           uint16_t SecurityUse,
                           uint16_t ACLEntry);

}
```

Figura 3.11: Definición de la interfaz MCPS_DATA

Status: Estado del paquete enviado.

SecurityUse, ACLEntry: Parámetros de seguridad no utilizados.

3.4.2. Testeo de implementación y medidas en laboratorio

Luego del estudio detallado del Standard, se procedió a testear la implementación Mac provista por el grupo Hurray. Se buscó verificar si los servicios que se ofrecían concordaban con las especificaciones. Para ello se partió de implementaciones de test ya implementadas por ellos, que probaban la asociación y envío de datos, y la gestión de GTS y el envío de datos en estos espacios de tiempos garantidos. Estas funciones fueron testeadas sin la herramienta de debug que se usó a lo largo del proceso de desarrollo ya que para ese entonces, no se contaba con el correcto funcionamiento de la función printf. Si bien no fue lo mas agradable ni lo mas correcto realizar el testeo con leds, esta etapa consideramos fue muy importante y debe quedar resaltada y explicada, ya que fueron nuestros primeros desarrollos específicos para el testing, y nuestro primer acercamiento a la comprensión del código ya provisto, y de la verificación con lo especificado en el Standard. Cabe destacar que estos casos contemplaban un caso de funcionamiento normal, por lo que faltarían realizarse un extra de pruebas testeando otras casuísticas.

3.4.2.1. Asociación y envío de datos

Para probar esta aplicación lo que se hizo fue programar un mote como PAN coordinator y otro como end device(esto es desde un archivo de configuración de la aplicación de test). Antes de asociarse, observamos que el dispositivo debe sincronizarse, pero previo a esto, debe de conocer el canal en el cual debe escuchar esa sincronización. Todo comienza en el archivo MacM, ya que lo que realiza la capa mac es disparar la indicación *MLME_SYNC_LOSS.indication* cuando se dejan de escuchar un número de AMaxLostBeacons de beacons. Esto se hace en el evento que indica el fin de una supertrama *async event error_t TimerAsync.sd_fired()*. La aplicación lo que hace es hacer un ORPHANSCAN de los canales, y luego un PASSIVESCAN también de todos los canales. Cuando obtiene la confirmación, se fija en cuál se obtuvo mejor LQI (que en este caso es uno solo) y pide la sincronización en ese canal con el comando *MLME_SYNC.request*. Este comando no tiene respuesta, solamente inicia sus timers con la información de los beacons recibidos. Luego la capa MAC escucha ese canal, y se comienza a disparar el evento *MLME_BEACON_NOTIFY.indication* el cual se observa en la implementación de la capa MAC que esta indicación se da cuando se recibe una trama con su campo de FRAMETYPE en valor BEACON. Cuando le llega esta indicación, la aplicación realiza cierto control si el dispositivo ya tiene levantada la bandera de sincronización (go_associate ==1) y si vio pasar al menos 5 beacons, manda la asociación hacia el pan coordinator. Otra cosa importante que se hace en el archivo MAC(dentro de la función que lanza el evento *BEACON_NOTIFY*) es disparar todos los timers que sincronizan al device en su beacon, a saber, *TimerAsync.set_bi_sd(BI,SD)* es quien marca los intervalos de la supertrama y de la beacon. Como ya se mencionó anteriormente, estos parámetros necesarios para disparar el timer, se obtienen de la trama beacon recibida. Cada vez que los timers disparan el evento *event AsyncTimerFired.bi_fired()*, se manda encender el led azul cuando comienza el beacon y se apaga cuando termina la supertrama. Cuando la aplicación recibe la confirmación de la asociación, dispara el timer: *Timer_Send.startPeriodic(3000);* el cual hace que cuando se dispare su evento al terminar, se mande un dato con su control previo de que no se manden más de 5 datos. Dentro de este mismo evento del timer, si ya se mandaron 5 paquetes de datos, se detiene el timer, y pide desasociarse. Esto lo pudimos controlar en el mote que recibía estos datos verificando que toogleara los leds. Pruebas

extras, testeando otra casuística: Cuando arranca si no ve ninguna trama beacon hace los scans y si no encuentra nada en ninguna dirección física, se queda inactivo, no realiza ninguna otra gestión para asociarse, lo cual se debería hacer desde la aplicación.

3.4.2.2. Gestión de tiempos garantidos (GTS)

Se probó que se enviaba el comando *GTS.request*, y que el coordinador respondía, ya que al endDevice le llegaba la confirmación en el próximo beacon(ya que el evento se dispara en la función *processBeacon()* de la capa MAC). Luego se enviaban tramas periódicamente y vimos que después de que se mandaban las 30 tramas de datos se pedía correctamente el gts request de desasociación. Se verificó que este se desasociaba correctamente ya que se continuaban mandando las tramas (con la opción de enviarlo en un GTS) periódicamente, y el evento de dato llegado en el otro mote no se señalaba. Se probó variar la frecuencia de las tramas transmitidas y alternando el led en la recepción, se observó si se enviaban cantidades impares o pares de tramas en un periodo de gts, es decir que si se podían mandar más de una trama de datos(porque la frecuencia de envío era más alta que el BO) si ésta era impar el led cambiaba de estado y si ésta era par quedaba igual. Pruebas extras, testeando otra casuística: se probó sin conectar el otro mote, el coordinador y se concluyó que debería tener una confirmación fallida con status de NO_ACK(ya que la trama de comando de solicitud del GTS debe ser ACKGED por el coordinador), lo cual no sucedía. No se disparaba la confirmación de GTS, por lo que se revisó el código de la MAC y se encontró que al dispararse el evento del timer de espera de reconocimiento, *T_ackwait.fired()*, no se señalaba el evento de confirmación al estar éste asociándose.

3.4.3. Modificaciones necesarias

Debido al largo proceso de testing que fue sujeto el componente MacM, y que los resultados obtenidos fueron parcialmente satisfactorios, se realizaron varios cambios a la implementación disponible. Esto se debió en gran parte a que esta implementación estaba ligada a aplicaciones concretas y específicas, y no cumplían con el Standard en su totalidad. Para poder cumplir con los requerimientos de esta aplicación, se fueron destacando durante este proceso de testing varios cambios a realizarse en esta implementación, y ello ha traído como consecuencia que el producto usado como punto de partida sea diferente al que se utiliza en el producto final. Si bien en esta etapa dentro del proyecto no se invirtió tiempo en implementar una capa de acceso al medio, se debió entender el código en su totalidad ya que al realizar el testeo en la mayoría de las funcionalidades, estas no estaban completamente implementadas y se encontraron varios bugs de los cuales algunos fueron enviados como consultas al autor. En el próximo capítulo se detallan las modificaciones realizadas a los componentes de bajo nivel.
Cabe destacar que para esta segunda etapa de test ya se encontraba disponible la función printf, herramienta muy útil a la hora de debug, cuyo funcionamiento se detalla en el anexo C.

Capítulo 4

Componentes principales de bajo nivel

En este capítulo se mencionan los componentes utilizados como punto de partida a la hora de comenzar con el desarrollo e implementación del proyecto. Se menciona parte del proceso de pruebas a los cuales fueron sometidos, y los cambios necesarios para poder utilizarlos correctamente desde la aplicación desarrollada.

4.1. MacM

Este es uno de los componentes principales de la arquitectura del sistema, ya que es quién brinda servicios a la entidad de red desarrollada en este trabajo. Si bien la arquitectura de partida de este componente es la sugerida por la Universidad de Porto, se realizaron modificaciones en eventos, comandos y funciones que nos llevan a explicar el componente final.

4.1.1. Modificaciones realizadas a la Implementación de partida

A continuación se muestra para cada servicio las modificaciones necesarias para el correcto funcionamiento de nuestra aplicación.

4.1.1.1. Escaneo de los canales

Este servicio se realiza a través de la interfaz *MLME_SCAN*. El desarrollo de esta función se realiza en los siguientes pasos:

- Tomar la lista de canales físicos en los cuales se deberá realizar el scan y el tiempo de scan por canal.
- Disparar un timer periódico por ese tiempo de duración y realizar el cambio de canal al siguiente, una vez terminado éste.
- Al capturar una trama se debe, en función del tipo de scan, registrar los valores de energía, canal y demás.
- Indicar a la aplicación el resultado obtenido

Para ello, se mantienen las variables:

- *uint8_t current_scanning*: almacena el canal físico en el cual se realiza el Scan
- *bool scanning_channels*: booleano que simplemente deshabilita a procesar un dato de llegada.
- *SCAN_PANDescriptor scan_pans[16]*: es una estructura con información de los tiempos de la red y direcciones del coordinador[1], una para cada canal. Se retorna con datos relevantes si el scan solicitado es pasivo. Si es scan de detección de energía se devuelve vacío.
- *uint8_t scantype*: mantiene la información si es un scan pasivo o de detección de energía.

Al comenzar un scan se levanta la bandera *current_scanning* para que frente a la llegada de datos se guarde en una variable global, la cual es retornada a la aplicación al expirar el timer de scan. Al finalizar el scan se baja la bandera y se dispara el evento *signal MLME_DEVICE_SCAN.confirm*;

Se debió modificar:

- Agregado de encendido del transceiver al inicializar el scan y apagado de esta al finalizarse.
- Seteo y reseteo de dirección de broadcast en la interfaz AddressFilter al inicializarse y culminarse respectivamente para que el componente CC2420RecieveP no realice el filtrado usual.
- Hacer el cambio de scan en función de la lista de canales seleccionados y terminar al completarse el scan en los canales seleccionados (se realizaba en todos los canales, enlenteciendo el funcionamiento de duración del scan)
- Orphan scan: quitamos esta funcionalidad ya que su uso requiere mantener el transceiver encendido todo el tiempo por parte del coordinador. Active scan no se encontraba implementada ni nos sería de utilidad por los mismos motivos.

Las siguientes figuras son del standard y muestran el diagrama de flujo para el passive scan de la red. Para el caso de tratarse ED el algoritmo permite recibir cualquier tipo de trama, sea beacon o no.

[1] para más información sobre la estructura, ver [5]

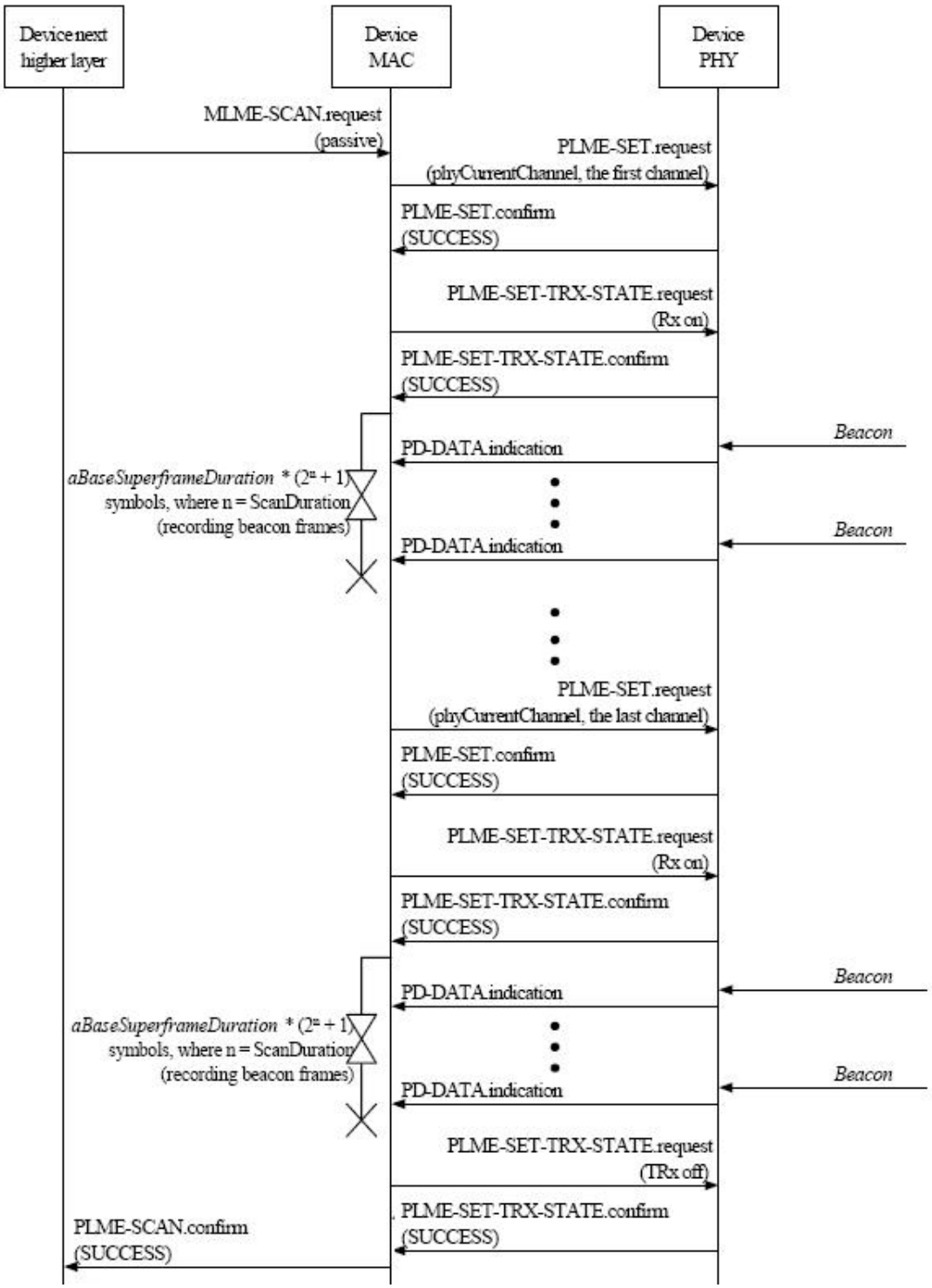


Figura 4.1: Diagrama de secuencia extraído de [4] para el passive scan

4.1.1.2. Asociación con el coordinador

En esta interfaz no se realizaron cambios ya que la funcionalidad se encontraba correctamente implementada y funcionando tal cual se muestra en el standard. Se logró testear

el envío y recepción de tramas de comandos y reconocimiento y el funcionamiento de los eventos. En la siguiente figura se muestra todo el flujo de mensajes a nivel MAC intercambiado entre el coordinador y el dispositivo final.

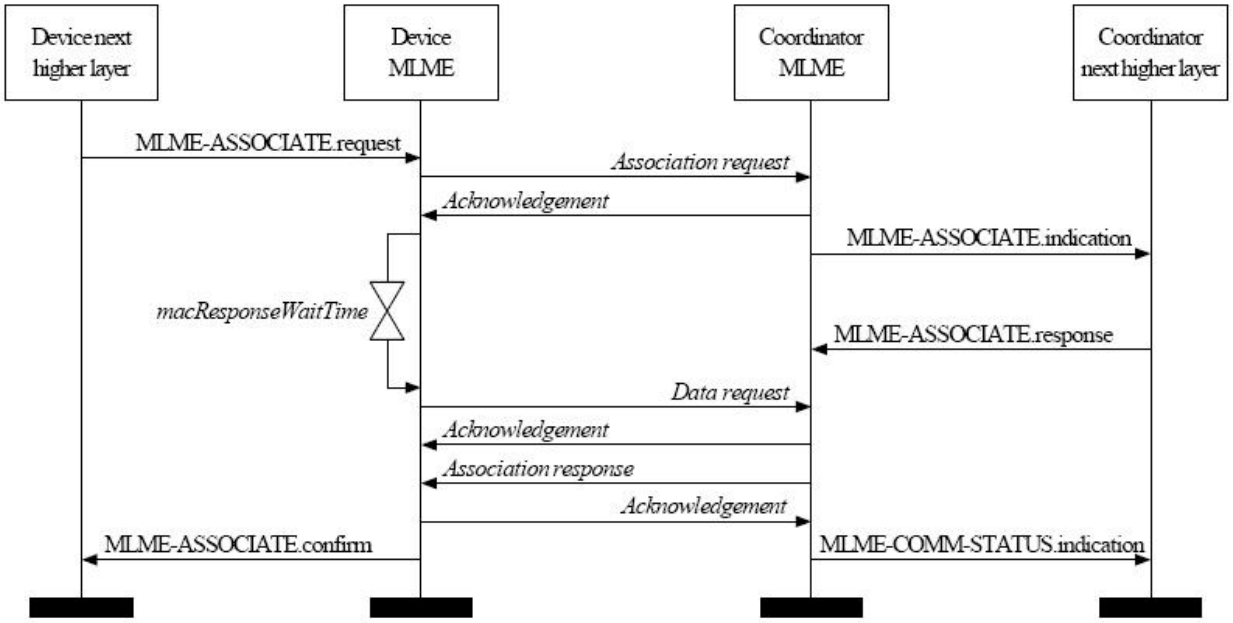


Figura 4.2: Diagrama de secuencia extraído de [4] para la asociación solicitada por un dispositivo hacia el coordinador

Lo único que no se encuentra implementado es el timer de expiración de *macResponseWaitTime*. Es decir, el dispositivo no espera ese tiempo, sino que envía el comando *data request*.

4.1.1.3. Comienzo del coordinador, arranque de la PAN

Como se mencionó anteriormente, la interfaz del coordinador que da la orden de comenzar la PAN y comenzar a emitir beacon es *MLME_START* Lo único que realiza es habilitar el timerAsync para que comience a llevar registro del tiempo, y dispare los eventos necesarios para poder enviar las tramas en el instante adecuado, y también enciende el transceiver para poder comenzar con el envío.

Los comandos:

- *call TimerAsync.set_backoff_symbols(backoff)*; //Tiempo de duración de retroceso (algoritmo CSMA)
- *call TimerAsync.set_bi_sd(BI,SD)*; //Seteo de duracion de la supertrama

setean al timer con los valores que la aplicación envía en el START

4.1.1.4. Recepción de beacons del coordinador

Esta función se encontró implementada correctamente y su funcionamiento se explica a continuación.

Cuando un nuevo nodo Device es agregado a la red, este comienza a realizar un escaneo en búsqueda de actividad, una vez que elige un canal empieza a escuchar beacons en él. Una vez terminado el escaneo, cada vez que se dispare el evento `PD_DATA.indication`, se encola la tarea `data_indication()`. En esta tarea se diferencia entre las distintas posibles tramas MAC que pueden llegar, las mismas pueden ser:

- TYPE_DATA: en este caso recibió un dato
- TYPE_ACK: en este caso recibió una confirmación
- TYPE_CMD: en este caso recibió un comando
- TYPE_BEACON: en este caso recibió un beacon

En el caso de que se reciba un beacon, la tarea llama a la función `process_beacon`. Esta función obtiene todos los datos relevantes que vienen en una trama de beacon, entre ellos el Beacon Order, el Superframe Order, la dirección MAC del coordinador, las características de los GTS, etc. Una vez que procesa toda esta información se sincroniza con los beacon, esto lo lleva a cabo iniciando el TymerAsync, con los valores de tiempo presentes en el beacon.

Otros puntos de interés que se realizan en conjunto son:

- Procesar la lista de GTS y la lista de direcciones pendientes (para gestionar los GTS y saber si el coordinador está esperando datos de él)
- Arrancar el Timer nuevamente con los valores de BO y SO de la red obtenidos en los campos del beacon recibido(habiéndose antes comunicado a TimerAsync)

La interfaz es *MLME_BEACON_NOTIFY* y el evento es *indication*

4.1.1.5. Administración de GTS

A continuación se muestra el flujograma para la solicitud del servicio(tanto para solicitar el servicio, o para darse de baja y que este pueda ser utilizado por otro dispositivo).

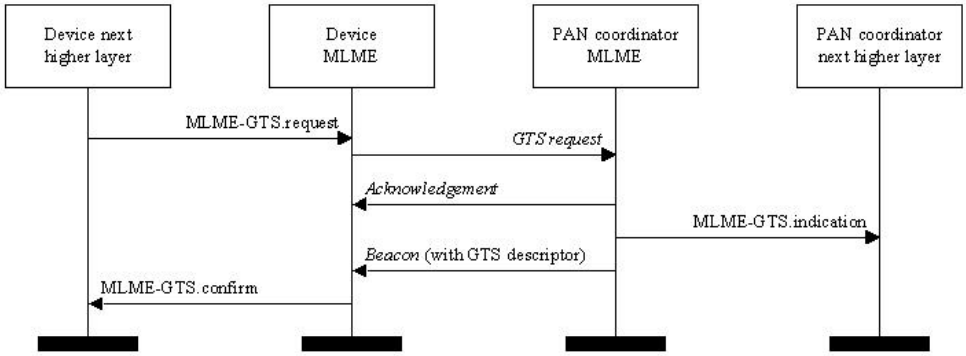


Figura 4.3: Diagrama de secuencia extraído de [4] para la solicitud de gts

Al solicitar el servicio, el coordinador debe:

- Chequear si no excede el máximo de dispositivos utilizando time slots. Según el Standard se podrá tener un máximo de 7. Es decir, en función de los dispositivos que hayan solicitado el servicio, queda definida la duración de los time slots del CAP, ya que la duración del período activo es fija en 16 time slots.
- Registrarlo en la estructura que manejan estos datos(GTSinfoEntryType)[5]de tal forma de que la función que se llama a la hora de armar el beacon lo tenga en cuenta y modifique los parámetros de supertrama. De esta forma se le comunica al dispositivo si se le fue adjudicado el servicio o no.
- Notificar a la aplicación de un nivel más alto la solicitud del GTS(indication).
- Dejar libre el espacio de tiempo asignado, para el uso exclusivo de GTS(recibir/transmitir)con el dispositivo que hizo la solicitud del servicio.

El device debe:

- Poder reconocer en el próximo beacon si fue aceptado o no el GTS.
- Notificar a la aplicación de un nivel más alto el resultado de la solicitud del GTS(confirm).
- Dejar libre el espacio de tiempo asignado, para el uso exclusivo de GTS(recibir/transmitir)con el coordinador.

No se encontraron diferencias entre el standard y se testeó el envío de datos a través de este servicio verificando la correcta transmisión y recepción de los datos.

Un agregado a esta funcionalidad que no se encontraba implementado, ni tampoco del todo claro explicado en el standard, es la dada de baja al servicio desde el lado del proveedor. Es decir, que sea el propio coordinador el que elimine el GTS solicitado por un dispositivo, que el resto de los dispositivos puedan seguir funcionando correctamente,

y que el GTS liberado quede disponible para cualquier otro dispositivo que lo solicite. La implementación de este requerimiento, se manejo a nivel de capa de red en la aplicación desarrollada, por mas información ver sección 6.

4.1.1.6. Envío y recepción de datos

No se encontraron mayores diferencias con lo mencionado en el standard, y se logró testear con resultados satisfactorios el envío y recepción de datos a través de esta interfaz.

Como se explicó en la sección 4.1.1.4, cuando se recibe una nueva trama de MAC, se dispara el evento `PD_DATA.indication` y esta encola la tarea `data_indication()`. En el caso de la recepción de datos el tipo de la trama será TYPE_DATA, en este caso se llama a la función `indication_data`, la misma se encarga de obtener información de la trama (Direccion MAC de origen, MPDU). Una vez obtenida dicha información,se dispara el evento `MCPS_DEVICE_DATA.indication` o `MCPS_COORD_DATA.indication`, para que sea procesado por la capa superior, cual es el evento disparado depende de quién fue que recibió el dato, el nodo cuando funciona como coordinador o cuando funciona como dispositivo final.

Por otro lado, la situación cambia cuando se quiere enviar un dato, en este caso todo comienza con el llamado del comando `MCPS_DEVICE_DATA.request` o `MCPS_COORD_DATA.request` desde la capa superior. En el mencionado comando se llama a la función `create_data_frame`, se llama en dos situaciones distintas dentro del comando, una en el caso de envió de datos en los GTS y otra en el envió de datos en el CAP (contention access part), realizando en este caso CSMA. En funcionamiento normal, es decir una vez asociado y transmitiendo datos, un nodo enviará sus datos en el CAP, cuando funciona como coordinador, realizando CSMA y un nodo funcionando como dispositivo final enviara datos a su coordinador en el GTS asignado.

4.2. Componente TimerAsync

Como se mencionó anteriormente, este componente es quien implementa las funcionalidades necesarias para un correcto sincronismo entre el PAN coordinator y los otros dispositivos de la red IEEE 802.15.4. El diseño de este componente fue algo que no se tuvo que realizar, ya que la definición de la funciones y la interfaz con el componente MacM estaban ya hechos por este grupo de desarrolladores.

4.2.1. Implementación de partida

El fuerte principal de este componente está en utilizar el oscilador externo de 32kHz, el cual se encuentra conectado a el MSP430 [8] y permite realizar la cuenta de manera independiente del reloj del microcontrolador. Cada vez que este periférico interrumpe al microcontrolador se dispara el evento que hay dentro del componente: *async event void AsyncTimer.fired()*. Allí incrementa un contador y si este es igual a los valores característicos de la supertrama, se disparan los eventos que correspondan, estos son:

***async event error_t before_bi_fired()*:** Evento que se dispara una cantidad configurable de símbolos antes de la llegada de beacon, y es requerido para poder encender el

transceiver y que el endDevice pueda recibir correctamente el beacon, y para el coordinador pueda enviarlo correctamente.

***async event error_t sd_fired()*:** Se dispara una vez culminada la supertrama.

***async event error_t bi_fired()*:** Se resetea el contador de símbolos y avisa a la Mac el comienzo de un nuevo beacon.

***async event error_t time_slot_fired()*:** Indica el comienzo de un timeSlot dentro de la supertrama. Es necesario para los espacios de tiempos garantidos, en donde cada dispositivo dentro de la red posee un espacio de tiempo garantido(si así lo gestionó)

***async event error_t before_time_slot_fired()*:** Se utiliza para que dispositivos que no van a usar ese timeSlot apaguen el transceiver y dispositivos que tienen ese timeSlot cedido, lo enciendan para minimizar así el consumo de los dispositivos.

La siguiente figura muestra el comportamiento en el tiempo de este componente:

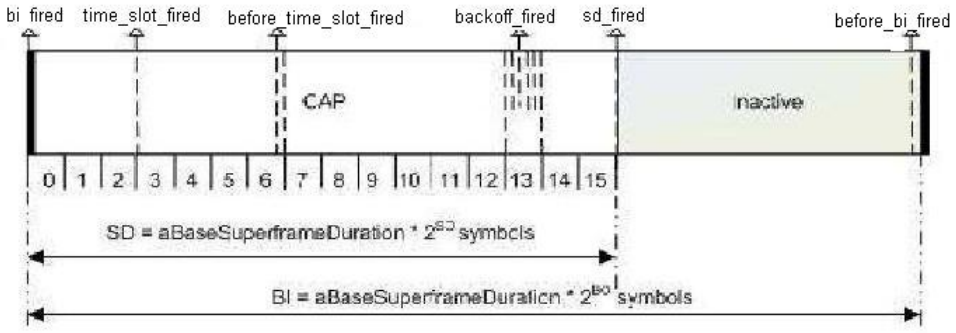


Figura 4.4: Eventos de Sincronismo del componente TimerAsync, desarrollo Hurray

Cabe destacar que la diferencia de tiempos entre los eventos *before_time_slot_fired* y *time_slot_fired*, es la misma que entre los eventos *before_bi_fired* y *bi_fired*, y se corresponden con la cantidad de símbolos necesarios para el hardware poder encender el transceiver. Para el caso del CC2420 por hojas de datos [9], se tiene un valor de 1120μs.

Todos estos eventos se definen en la interfaz TimerAsync, se disparan desde el componente TimerAsyncM (por ejemplo, con la sentencia signal *TimerAsync.before_bi_fired()*;) los cuales deben de estar definidos como funciones en MacM, es decir, en el código del componente que utiliza este componente, en algún lugar se debe de implementar el evento.

Nota: El uso de la palabra reservada async indica que al dispararse estos eventos, los mismos poseen prioridad frente a cualquier otra tarea que pueda estar ejecutando el sistema operativo TinyOS, y es propio de nesC.

4.2.2. Modificaciones realizadas.Cambio de estado del microcontrolador

En primera instancia se observó un correcto funcionamiento de este dispositivo, y que este contaba con todas las necesidades requeridas por MacM para un eficaz sincronismo. No obstante, el algoritmo que utiliza el componente para llevar registro de la cuenta, no

es del todo eficaz en temas del consumo, porque el oscilador externo está constantemente interrumpiendo al microcontrolador y llevándolo a un estado de actividad cada 305.2 μ s (10/32.768kHz).

Por las hipótesis tenidas en cuenta en el análisis de consumo 7, el estado del microcontrolador debe ser standby[2] durante todo el tiempo de inactividad. De no ser así, no se estaría cumpliendo con el requerimiento de consumo superior al año.
El hecho de interrumpir al micreocontrolador constantemente es algo que debió de solucionarse, por lo que se procedió a realizar una leve modificación para intentar anular la actividad del microcontrolador durante todo el período de inactividad.

Para lograr este objetivo, se modificó el componente para que en el tiempo inactivo detuviera la actividad del contador. A continuación se explican las variables agregadas y la lógica implementada:

Se introducen las variables:

bool estoy_inactivo: Indica si el mote se encuentra en el período de inactividad o en la supertrama.

uint32_t ticks_inactivo: indica la cantidad de ticks de inactividad.

El siguiente flujograma muestra la lógica que se efectúa al dispararse el evento del timer hardware: AsyncTimer.fired().

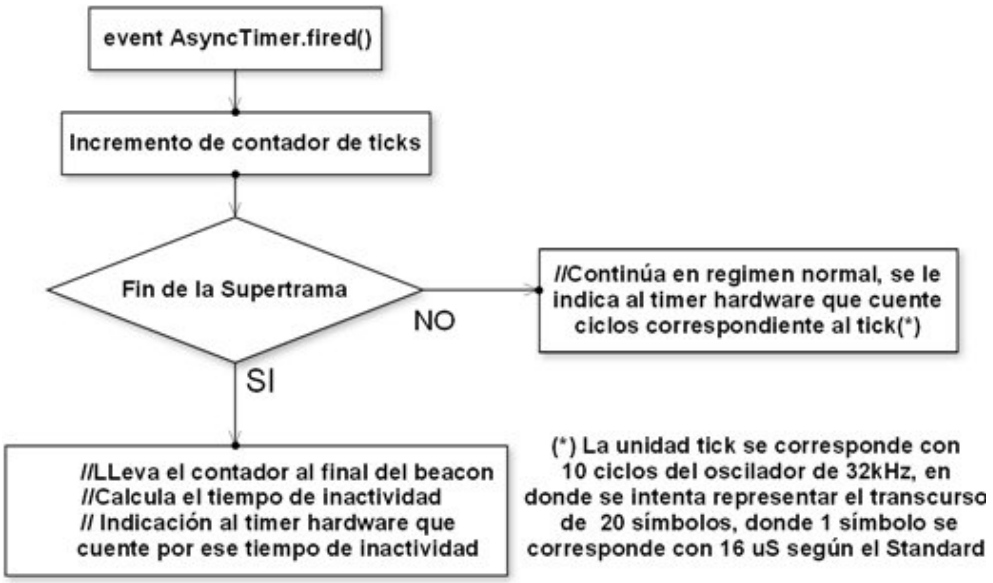


Figura 4.5: Modificaciones a la lógica del TimerAsync

Al llegar al fin de la supertrama se calcula el tiempo de inactividad como :
ticks_inactivo = before_bi_ticks - sd_ticks - 1; las constantes before_bi_ticks y sd_ticks son las que limitan el intervalo de inactividad, se le resta uno, porque antes de comparar contra las constantes primero se incrementa en uno el contador de ticks.

[2]Ver figura 2.2

Al entrar en período inactivo, el timer no interrumpirá más al micro hasta que se termine la supertrama. Lo que se hace a nivel de programación es modificar la constante con la que se le carga al llamado del timer:

```
call AsyncTimer.start(10); //Durante la supertrama
call AsyncTimer.start(ticks_inactivo*10); //Cuando termina la supertrama
```

4.2.3. Agregado de funcionalidades-Doble Interfaz con Mac

Para poder cumplir con el objetivo propuesto del bajo consumo en los motes, fue necesario lograr la sincronización entre estos. Se tuvo presente también que la entidad de red deberá manejar dos entidades(MacCoordM, y MacDeviceM)para poder hacer de router entre dos redes IEEE 802.15.4. La primera solución al problema de poder sincronizar el dispositivo con dos de estas redes, fue poder tener dos instancias del componente TimerAsync, y cada una de ellas conectarla a su correspondiente red. Esta solución no fue exitosa, ya que el módulo no es genérico, y por lo tanto no permite tener varias instancias del componente. Este problema no fue nuevo para nosotros ya que al querer instanciar dos componentes Mac, tampoco se nos permitió debido a la unicidad de componentes en nesC. En ese momento se optó por simplemente crear un nuevo componente y renombrarlo. Es decir, tanto MacCoord como MacDevice tiene definidos cada uno una instancia de TimerAsync, por lo cual el problema estaría solucionado. El diagrama de estos componentes sería:

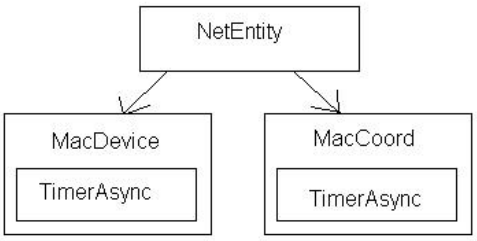


Figura 4.6: Esquema sugerido para solucionar el problema

El problema que esto trajo fue que el componente timerAsync es único, por mas que cada componente de la capa MAC tuviera cada uno una instancia de TimerAsync, ambos iban a estar conectados al mismo componente, de la misma manera que sucede con los servicios de la capa física (componente Phy), y con los leds(leds). La verdadera conexión que hace TinyOS es:

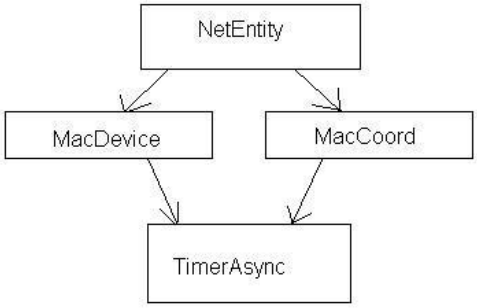


Figura 4.7: Resolución de nesC para componentes únicos como TimerAsync

Sucede que todos los eventos que se disparan desde el componente TimerAsync son capturados en los dos componentes MacDevice y MacCoord, por lo que se debió buscar una solución. La solución de crear un nuevo componente y renombarlo no fue viable porque observamos que el TimerAsync utiliza los servicios de un componente que usa el oscilador externo: *Alarm (T32khz,uint32_t)* del cual concluimos no se pueden tener varias instancias. La solución final fue crear un nuevo componente que llevara el registro de todos los tiempos, tanto del coordinador como del endDevice, es decir, tanto MacDevice como MacCoord se encuentran unidos a este nuevo timer. La diferencia con el principal, es que además de tener registro de todas las constantes encargadas de disparar los eventos de sincronización (que se setean en función del BeaconOrder, y del SuperframeOrder), también se encuentran seteadas las constantes que disparan los eventos hacia la otra Mac. El siguiente esquema de tiempos muestra el funcionamiento del nuevo componente.

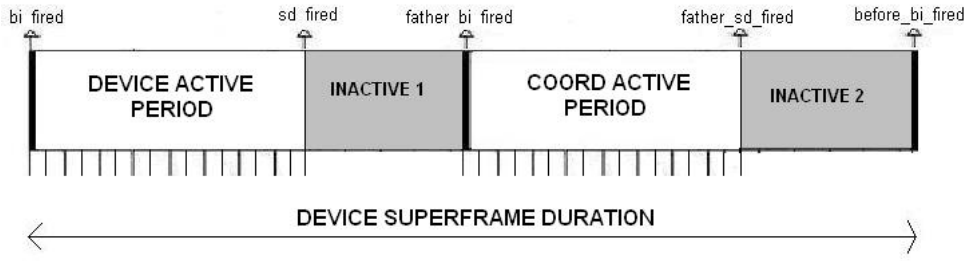


Figura 4.8: Eventos agregados para el manejo de las dos redes IEEE 802.15.4

La longitud del intervalo se corresponde con la duración de una trama beacon. Es decir, el evento before_bi_fired antecede a que se dispara nuevamente bi_fired para la llegada de una nueva supertrama. Existen eventos que se reiteran y se nombran allí con el prefijo father; lo cual hace referencia a los eventos que utiliza la interfaz del coordinador(MacCoord).
A continuación se explican los tiempos principales mostrados en la figura:

DEVICE ACTIVE PERIOD: Es el tiempo de actividad de la interfaz que se utiliza para poder comunicar al dispositivo con su padre, es utilizado por MacDevice para poder transmitir datos hacia el coordinador.

COORD ACTIVE PERIOD: Es el tiempo de actividad de la interfaz que se utiliza para intercambiar datos con los hijos, aquí es donde se disparan los eventos que utiliza MacCoord para enviar y recibir tramas con los hijos.

INACTIVE 1 y 2: Son tiempos de inactividad que existe entre ambas supertramas. En la realidad estos tiempos son más largos que los tiempos de inactividad.

Los parámetros que regulan estos tiempos son:

BO: Beacon order u orden del beacon. La cantidad de ticksCount, es decir, hasta donde cuenta el temporizador hardware depende exclusivamente de este valor

SO: Superframe order u orden de supertrama. Determina el largo del tiempo activo, es decir, especifican que tan grandes son los slots o ranuras de tiempo que se muestran en la figura de duración de tiempo activo.

Nota: Observar que el tiempo comprendido entre el fin de DEVICE ACTIVE PERIOD y el final de INACTIVE 2, es un tiempo de total inactividad para la interfaz del lado de MacDevice, es decir que durante todo ese tiempo no se establecerá comunicación alguna con el coordinador del dispositivo. Si durante este período se requiere enviar un dato a través de esta interfaz (como puede ser la captura de un dato del sensor u otro evento asíncrono), se deberá de esperar hasta el otro período de actividad.

Se mantuvo el criterio de no interrumpir el microcontrolador en los períodos de inactividad. Para ello se ejecuta una lógica similar a la que se explica en la figura 4.5, pero en este caso calculando dos períodos de inactividad diferentes. También se lleva el control de la variable *uint8_t actual_beacon* la cual puede tomar los valores BEACON_DEVICE o BEACON_COORD indicando el estado de actividad actual del componente. Esto es necesario porque la lógica para el manejo de los eventos:

- before_time_slot_fired().
- backoff_fired()
- time_slot_fired()

es la misma tanto estando en un período de actividad o en otro, con la diferencia de en que beacon se encuentre actualmente el dispositivo. Al comenzar el beacon se resetean los valores que necesarios para el control de estos eventos, y antes de disparase se consulta por el estado actual.

La próxima figura muestra la interacción de este componente con el componente MacDevice. Se indica también donde se encuentra el período de actividad de MacCoord.

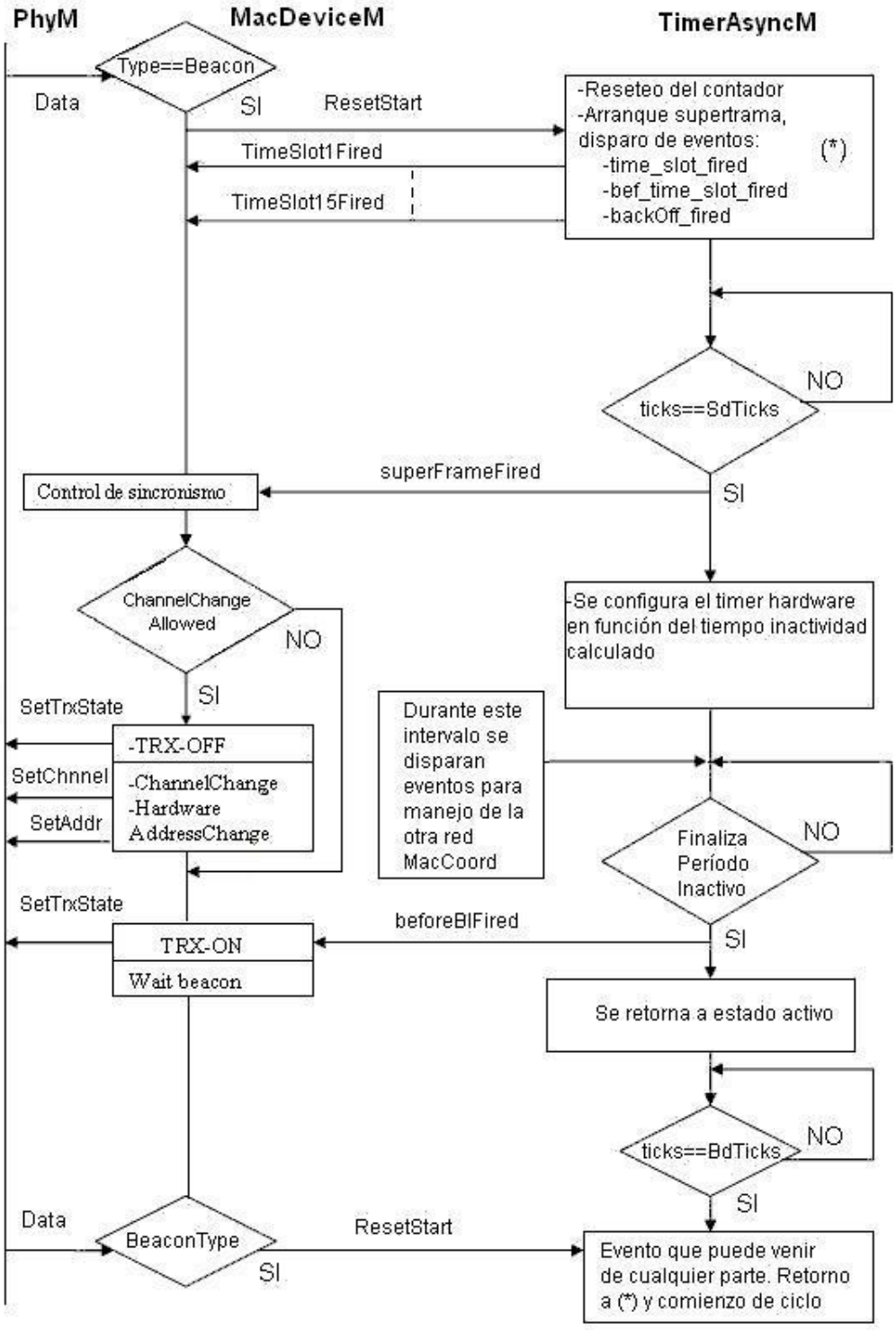


Figura 4.9: Diagrama de la sincronización final. Interacciones con MacDevice y MacCoord

Se muestra del lado izquierdo el componente PhyM que es el encargado de recibir los datos desde el transceiver, y le indica a MacDevice a través de la interfaz PD_DATA. MacDevice se encarga de analizar el header y en caso de ser una trama de comando, con

valor BEACON_TYPE, en el campo type, entonces se dispara, lo que hace es resetear el timer pero no desde 0, sino que un valor constante que es la suma del tiempo que el paquete está en el aire, y el tiempo de procesamiento de la trama. De esta manera se asegura que ambas supertramas terminaran al mismo tiempo. Al terminar la supertrama se realizan algunos controles, e interacciones con el canal físico para el correcto control de acceso al medio. Luego viene el período de tiempo en donde el timer dispara los mismos eventos de time_slots para el coordinador y luego al finalizar el período activo la MAC enciende el el transceiver frente a la orden del timer, para poder estar disponible y poder escuchar el próximo beacon y repetir el ciclo nuevamente.

4.2.3.1. Implementación de interfaces de doble MAC

Debido a la necesidad de implementar doble capa MAC, fue necesario realizar la habilitación de cada una de estas componentes en forma independiente. Esto se debe a que el nodo debe alternar entre dos canales físicos diferentes: uno de estos para comunicarse con el coordinador y el otro para generar sincronismo y comunicarse con sus hijos en la interfaz que se encuentra funcionando como coordinador.

Las interfaces creadas fueron las siguientes:

SON_FATHER_SWITCH : Habilita al componente MacDevice a conmutar de canal una vez terminada la supertrama. También habilita la recepción de datos provenientes de la física.

FATHER_SON_SWITCH : Habilita al componente MacCoord a conmutar de canal una vez terminada la supertrama. También habilita la recepción de datos provenientes de la física.

El agregado de estas interfaces se debió a que al tener que utilizar dos componentes MAC cableadas hacia la misma componente física, se debía deshabilitar los eventos de recepción de datos provenientes de esta última, en función de la supertrama en ejecución. Es decir, al estar dentro de la supertrama de MacDevice, los eventos de datos provenientes de MacCoord quedarían deshabilitados y viceversa.

4.3. PhyM

Este módulo es el nexo entre el software y el hardware utilizado. Se encarga del manejo del principal componente hardware en la comunicación entre dispositivos de la red, el chip de radio CC2420. La capa física debe proveer un grupo de primitivas, especificadas en el standard IEEE 802.15.4, estas primitivas permiten la correcta transferencia de datos y el correcto manejo del transceiver.

4.3.1. Partida

Se partió de la implementación realizada por el grupo Hurray, la misma estaba pensada para ser utilizada en aplicaciones con Beacon Order pequeño[3], es por este motivo que cierta cantidad de elementos no estaban implementadas en forma correcta, para nuestra aplicación. Los elementos no contemplados en esta implementación eran los siguientes:

- No se realizaba el apagado del transceiver.
- No se contemplaba el cambio de estado del transceiver, entre transmisión y recepción.

Para el manejo del transceiver, simplemente era prendido al arranque de la aplicación y luego no era apagado nunca más. Para poder manejar en forma correcta el transceiver, ya sea para prenderlo o apagarlo y cambiar su estado entre transmisión y recepción es necesario cumplir con los estados indicados en la hoja de datos del chip.
Como se observa en la figura 4.10, es necesario realizar algunos pasos previos antes de tener el transceiver listo para poder transmitir o recibir. En primera instancia es necesario encender el regulador de voltaje. Una vez que se encendió el regulador de voltaje, es necesario mediante el strobe [4] *SXOSCON* encender el oscilador, para sacarlo del estado *Power Down* y llevarlo al estado de espera *IDLE*. Desde este estado, es posible mediante los strobes *SRXON* y *STXON* poder llevar al CC2420 a los estados de receptor o transmisor respectivamente. Por otro lado, para apagar el transceiver es necesario utilizar el strobe *SXOSCOFF* para apagar el oscilador. Una vez apagado este último, estamos en el estado*Power Down* y para dejar completamente inactivo el transceiver es necesario por último apagar el regulador de voltaje.

[3] Explicación brindada por los autores, a quienes se consultó vía email

[4] Comandos de orden del chip CC2420, mediante los cuales el microcontrolador envía ordenes por la interfáz SPI

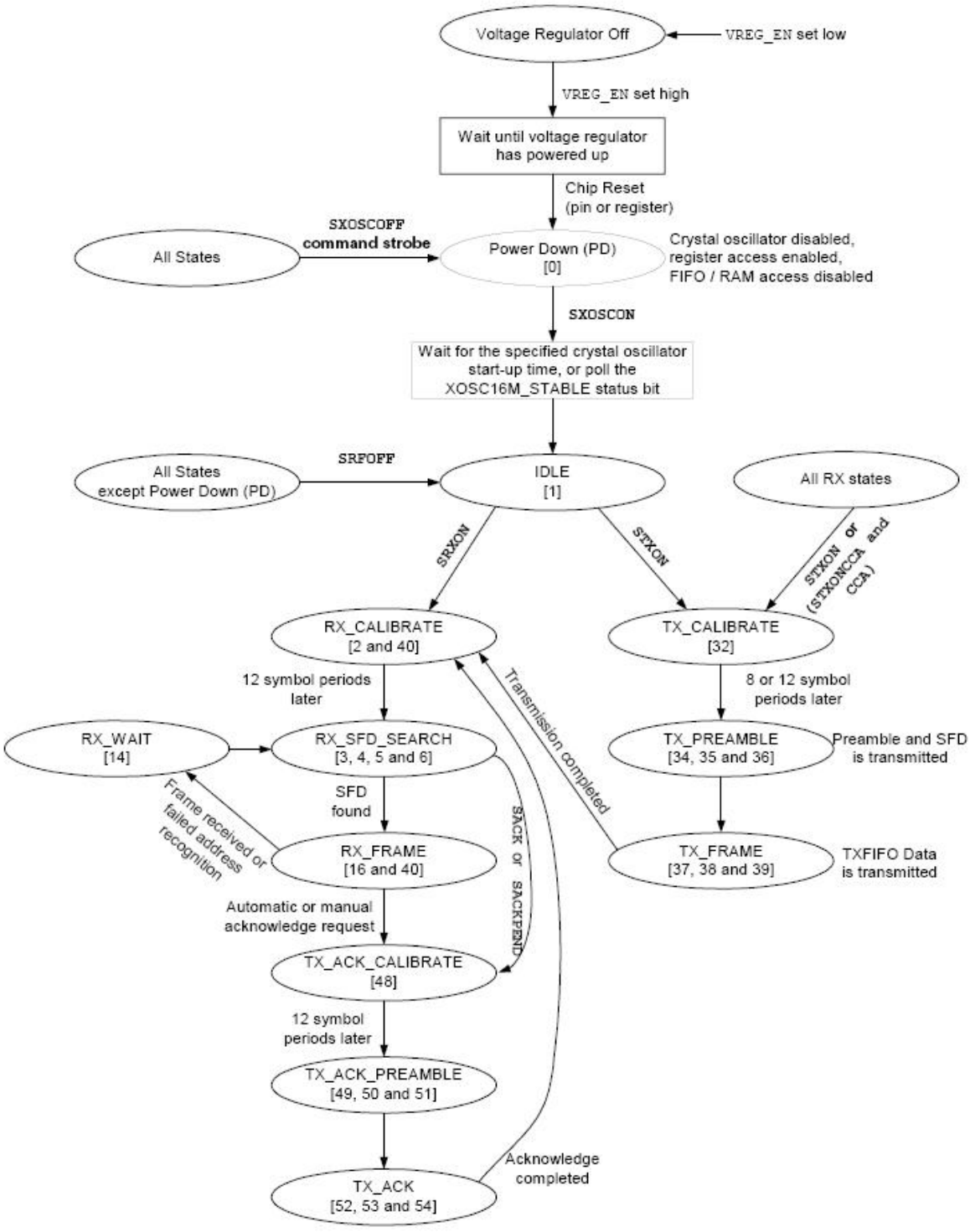


Figura 4.10: Diagrama de estados del transceiver CC2420, extraído de [9].

El motivo por el cual es fundamental manejar en forma correcta el transceiver es el consumo. Para poder lograr el objetivo del bajo consumo es necesario poder apagar el transceiver en los tiempos de inactividad y encenderla un instante antes de la llegada de un beacon. Como la implementación de Hurray era para beacon order pequeños, no era conveniente apagar el transceiver, debido a que el tiempo de establecimiento del regulador de voltaje, cada vez que se enciende el transceiver, es significativo frente al tiempo de beacon presente en su aplicación. Este es el motivo por el cual el mencionado grupo no realizó el manejo del transceiver.

4.3.2. Modificaciones realizadas

Debido a lo mencionado en la sección 4.3.1, fue necesario realizar ciertas modificaciones en la primitiva `PLME_SET_TRX_STATE.request`, para lograr que cumpliera el objetivo del correcto manejo de los estados del transceiver. Esta primitiva originalmente no tenía el código que lo contemplara, por tal motivo fue implementada según lo especificado en el standard IEEE 802.15.4. A `PLME_SET_TRX_STATE.request` se le pasa como argumento el estado al cual quiere pasarse el transceiver, los mismos pueden ser:

- RX_ON - Manda prender el transceiver como receptor
- TX_ON - Manda prender el transceiver como transmisor
- TRX_OFF - Manda apagar el transceiver, esperando que termine de recibir o transmitir
- FORCE_TRX_OFF - Manda apagar el transceiver, en este caso en forma directa sin esperar a que termine de procesar.

la tarea de esta primitiva es la de conmutar el transceiver entre esos tres posibles estados, teniendo en cuenta las precauciones previstas en el standard para antes de realizar el cambio.

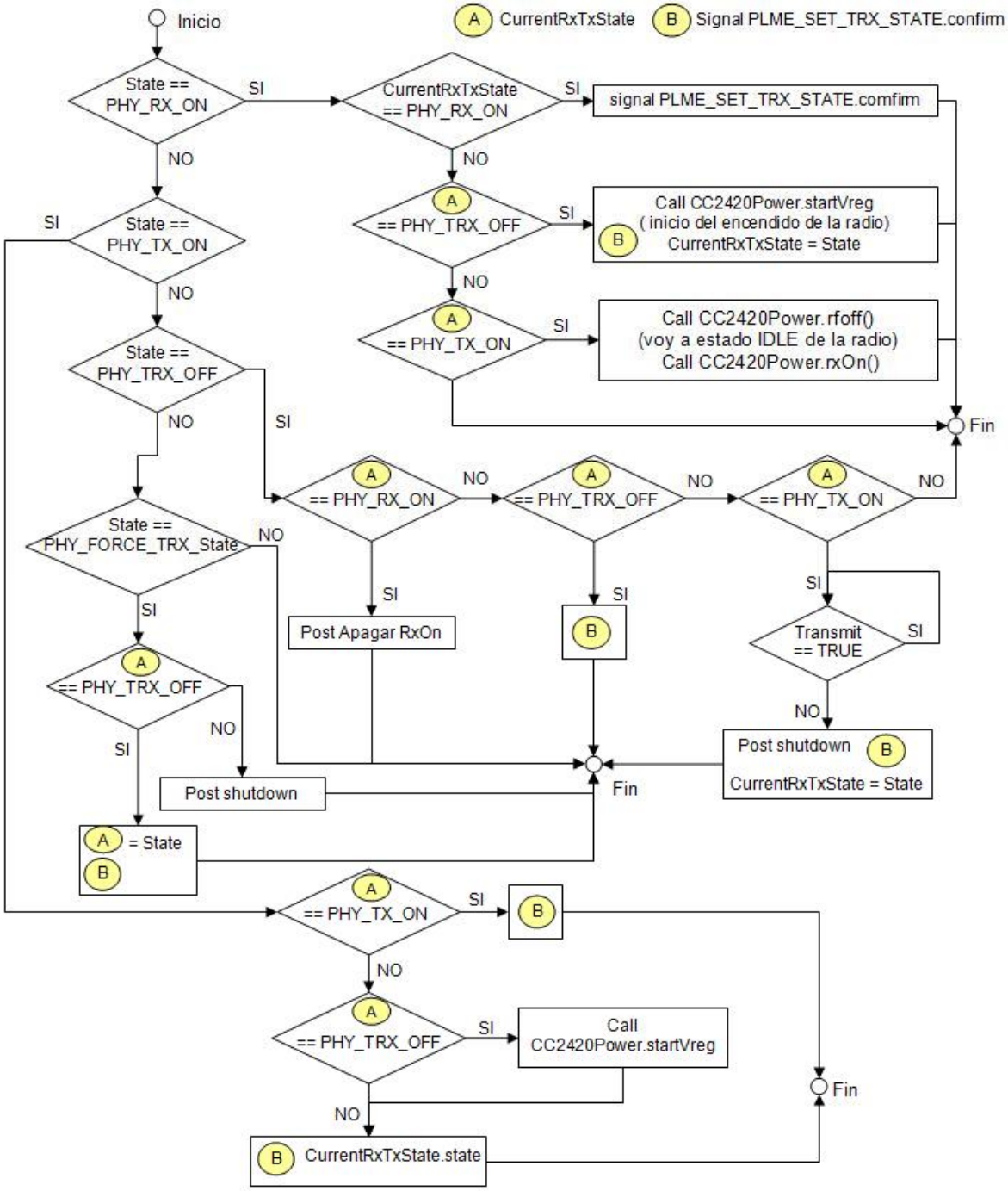


Figura 4.11: Diagrama de flujo del comando `PLME_SET_TRX_STATE.request`. STATE es el parámetro de la función y sobre el cual se ejecutan las validaciones.

Aparte de los cambios realizados en el comando `PLME_SET_TRX_STATE.request`, se debieron implementar algunas tareas.

- `task void shutdown()`: esta tarea manda apagar el transceiver, llama a los comandos `SubControl.stop()`, `CC2420Power.stopVReg()` y se encola la tarea `stopDone_task()`.

Estos tres elementos mandan a detener los elementos utilizados para el uso del transceiver y apagan el regulador de voltaje, logrando así apagar en forma correcta el transceiver.

- `task void Apagar_RXON()`: esta tarea es utilizada para evitar apagar el transceiver mientras se está recibiendo un paquete, una vez que se termina de recibir el paquete, encola la tarea `shutdown()`, mandando apagar el transceiver.

4.4. CC2420ReceiveP

Este componente es el encargado de recibir las tramas directas del chip, guardalas en la memoria del microcontrolador, e indicarle al componente PhyM cuando este la trama recibida, y a que dirección de memoria ir a buscarla. Se encuentra cableado directamente PhyM y a MacM. A continuación se listan otras funcionalidades que realiza:

- Configuración del chip CC2420 a través del bus SPI(registros internos).
- Lectura de la cola de datos FIFO interna del chip utilizada para el almacenamiento de bytes recibidos a través de la misma interfaz anterior.
- Borrado de FIFO ante la recepción de trama incorrecta.

En el primer punto se encuentran dos items importantes, que son la configuración del canal físico del chip, y la configuración de las direcciones del standard (*macShortAddress, macCoordAddress, macPANID*). Estos parámetros últimos son fundamentales en la recepción de datos, porque en este componente se realiza un filtrado por estas direcciones, según sea el tipo de trama recibida. Esto permite que los eventos no lleguen a los componentes de más alto nivel, con el fin de evitar sobrecarga innecesaria en el procesador. Para ello, deben configurarse correctamente los valores de estas direcciones por las cuales se realiza el filtrado. Se debe permitir al dispositivo de más alto nivel (la capa MAC en este caso corresponde asumir tal rol) poder setear estos valores para poder recibir solamente las tramas que considere sean para el. La interfaz que permite esto y que conecta este componente con MacCoord y MacDevice recibe el nombre de AddresFilter y se muestra en la siguiente figura.

```
interface AddressFilter {

    command error_t set_address(uint16_t mac_short_address,
                                uint32_t mac_extended0,
                                uint32_t mac_extended1 );

    command error_t set_coord_address(uint16_t mac_coord_address,
                                      uint16_t mac_panid);

    command error_t enable_address_decode(uint8_t enable);

    }
```

Figura 4.12: Código de interfaz AddresFilter, que permite el filtrado de tramas MAC

Desde MacCoordM y MacDeviceM se puede llamar a este comando (por ejemplo con la sentencia *call AddressFilter.set_coord_address(dirFiltroDeseada,panIdFiltroDeseado*) cuando se quieran recibir tramas en la Mac solamente con estas direcciones.
En la siguiente tabla, se muestran para la trama esperada a recibir, cuales son los valores permitidos para poder recibir la trama en los componentes Mac.

Tipo de trama a filtrar	***Campos de la trama***	***Valores permitidos***
Beacon	Dir. MAC de origen	Dir. del coordinador, seteada por AddresFilter
Datos	Dir. MAC de destino	Dir. de MAC local, seteada por AddresFilter
Comando [5]	Dir. MAC de destino	Dir. de MAC local, seteada por AddressFilter
Comando	PANId de destino	PanId local seteado por AddresFilter

Tabla 4.1: Criterios de Filtrado de tramas

Es decir, para la recepción de una de las posibles tramas de capa MAC, se deberá de haber seteado con el addresFilter la dirección de MAC correcta, y el valor de PanId correcto. Estos valores pueden ser obtenidos por el componente Mac desde un Scan (para el PanId) y de la confirmación de la asociación (para la dirección MAC). Si las direcciones se setean en un valor de broadcast preestablecido (MAC_COORD_BROADCAST = 0xFFFF) el filtrado no se realiza.

Cabe destacar que el filrado de trama de datos no se encontraba implementado. Esto lo pudimos verificar al dispositivo recibir tramas en la MAC que no eran para el, es decir que no tenía su dirección de destino en el campo de dirección de destino de la trama. Por lo cual esta es una funcionalidad que se debió implementar y testear, lo cual fue logrado exitosamente.

Otra función importante es la recepción de tramas, al completarse la recepción de esta, el integrado CC2420 interrumpe al procesador, el cual se encarga de leer el buffer fifo en donde se encuentra la trama recibida. Al completar la recepción, se genera el evento receive el cual por la interfaz Receiveframe se conecta con PhyM, el cual dispara el evento PD_DATA.indication en MacCoord y MacDevice.

En la siguiente figura se puede observar el conexionado para el filtrado y recepción de datos con los componentes nombrados en el párrafo anterior.

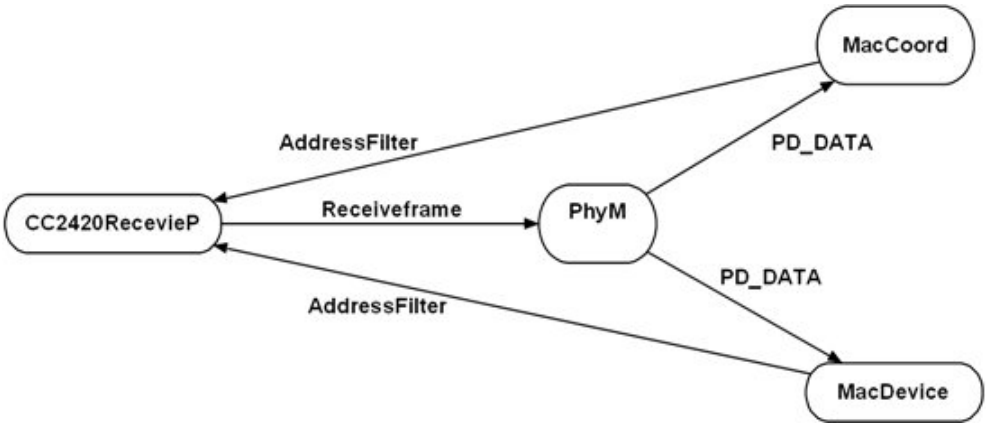


Figura 4.13: Conexión entre los componentes Mac, Phy y CC2420 para el filtrado y recepción de datos

4.4.1. Modificaciones realizadas en base a pruebas. Manejo de AddressFilter

La existencia de este filtro llevó a realizar leve modificaciones en el código para poder permitir desde la aplicación que maneja las dos entidades Mac con las que cuenta el sistema, poder modificar los valores de los parámetros que se manejan en el filtro. Esto se debe a que como se muestra en la figura, el componente CC2420 es único en el sistema, y sus parametros deben de poder cambiarse desde MacMCoord y desde MacMDevice. Para ello se muestran en la siguiente tabla todos los llamados a AddressFilter explicando en qué momento de las supertrama se llaman, desde qué componente, y qué valores se pasan por referencia.

Componente	***Momento en que se ejecuta***	***Objetivo buscado***	***Valores***
MacMCoord	Al terminar la supertrama del coordinador	Permitir recibir las tramas de la red IEEE del device	Direcciones de la red IEEE del device
MacMDevice	Al terminar la supertrama del Device	Permitir recibir las tramas de la red IEEE del coordinador	Direcciones de la red IEEE del coordinador
MacMDevice	Al solicitarse un SCAN desde la red	Permitir detectar cualquier trama en el canal de scan seleccionado	BroadCast

Tabla 4.2: Uso del filtrado para el manejo de dos redes IEEE 802.15.4

Capítulo 5

Diseño de la aplicación

En este capítulo se tratarán todos los temas referidos al proceso de diseño de la aplicación. Se comienza explicando las funcionalidades principales que esta capa debe cumplir, para luego definir los casos de uso. Se describen las distintas estructuras de datos empleadas en esta capa y se finaliza analizando la implementación.

5.1. Funcionalidades principales

En toda pila de protocolos, la capa de aplicación se encuentra en la parte superior y atiende las necesidades propias del usuario de la red. En este caso la capa de aplicación se ocupará principalmente de las tareas relativas al manejo de sensores, tomar las medidas pertinentes, cambiar entre un sensor y otro de entre los que tenga disponibles, y demás funcionalidades que no son exclusivas para el usuario pero sí para mantener un adecuado funcionamiento de la red en sí.

El cometido principal es tomar medidas de variables ambientales y que los datos recabados sean almacenados para el posterior análisis de los mismos. Brindar al usuario datos confiables y seguros de la manera más eficiente posible es la finalidad del presente proyecto, por lo cual la tarea de la capa de aplicación incide directamente sobre el desempeño de la red.

El propósito de la red es disponer de motes dispersos en un área de interés que se encuentran recolectando información de diferentes sensores y enviarlos a un nodo central, al cual confluyen todos los mensajes para luego transferirlos al router o PC (de ahora en adelante PC) al que se encuentra conectado. Para la red de sensores es transparente si quien se comunica con el sumidero es un router o un PC; en este proyecto siempre se utilizó un PC.

El alcance de este proyecto es transportar correctamente los datos hasta la interfaz del nodo Sumidero con el PC y cumplir con el protocolo definido en ella. La tarea de almacenar la información en una base de datos así como el transporte de los datos brindados por el nodo central atañe al grupo de proyecto de fin de carrera llamado SIMSI. Cabe mencionar que la definición de los mensajes y secuencias que rigen la interfaz entre los dos grupos de proyecto fue una tarea realizada en conjunto estudiando y analizando las

necesidades de cada grupo.

La idea global de ambos proyectos es brindar servicio a diferentes áreas, en donde cada una será cubierta por una red de sensores inalámbricos y un PC. A cada PC se conecta solamente una WSN (Wireless Sensor Network) la cual es denominada "hoja", y por cada hoja hay un solo nodo conectado al PC. Fue necesario trabajar en conjunto, de manera coordinada para tomar decisiones que se ajustaran a ambos grupos de proyecto, con el fin de lograr la mejor compatibilidad. Un ejemplo de esto es la definición de un protocolo en la interfaz PC-sumidero.

Para completar el propósito final de ambos proyectos resta hablar de la interfaz con el usuario, la cual es también implementada por SIMSI [1]. Considerando que ambos proyectos se desarrollaron simultáneamente, no se contó con mencionada interfaz por lo que para las pruebas fue necesario implementar una propia. Dicho programa fue creado en JAVA y se lo describe en la Sección 5.5.

A interés de este proyecto, tenemos tres entidades de capa de aplicación que se diferencian claramente: la red de sensores inalámbricos, el PC y la interfaz con el usuario. Claramente en una visión global de ambos proyectos se pueden distinguir más entidades pero ello excede los objetivos y los intereses del presente proyecto.

Cabe destacar que a lo largo del desarrollo del proyecto no se contó con el PC empleado por SIMSI, en cambio se utilizaron computadores personales, los cuales empleaban los puertos USB para adquirir los datos enviados por el nodo central y también enviar los mensajes necesarios para la gestión de la red.

Finalmente, la implementación consistió en un computador personal ejecutando una aplicación cliente para adquirir y enviar los mensajes por la interfaz USB y desplegando dicha información en la interfaz de usuario realizada en JAVA. Simultáneamente la WSN estaría funcionando y generando información con destino el nodo central que luego se reflejarían en la PC.

[1] http://simsi.no-ip.org/

Considerando ahora únicamente la WSN y sus funcionalidades, se definen dos tipos de nodos claramente diferenciados:

- mote ***no sumidero*** (no conectado al PC)
- mote ***sumidero*** (conectado al PC)

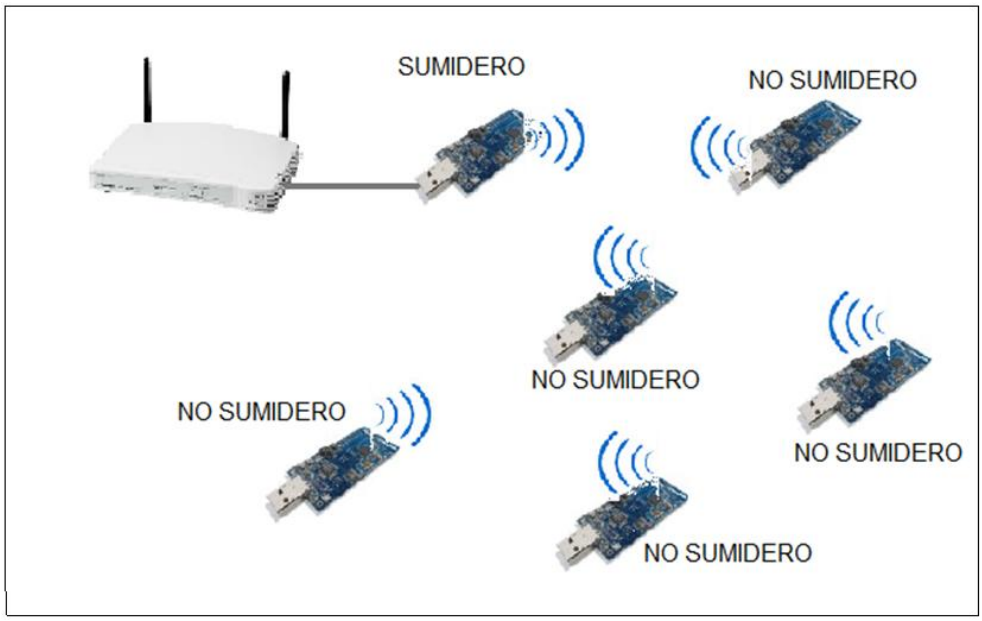


Figura 5.1: Esquema de la WSN (red de sensores inalámbricos).

La tarea principal de los No Sumideros es tomar los datos necesarios y enviarlos al Sumidero. Para ello, una vez que arrancan, deben inicializar las capas inferiores (entiéndase Red, Acceso al medio y Física) y luego de finalizada esta etapa notifican al Sumidero de su nacimiento.

A nivel de capa de aplicación los nodos no conocen a los demás, únicamente saben de la existencia del nodo Sumidero, pero el Sumidero sí sabe de la existencia de todos ellos y también sabe cómo llegar a cada uno. La tarea de cómo llegar a cada uno es atribuida a la capa de red y es un servicio brindado a la capa de aplicación.

La aplicación de los No Sumideros simplemente solicitan a la capa de red entregar determinado mensaje al Sumidero. En cambio el Sumidero puede solicitar entregar un mensaje solamente a un nodo en particular, pero también puede solicitar un envío *BROADCAST*.

El envío *MULTICAST* no fue implementado porque no se consideró necesario.

5.2. Estudio de casos de uso

Antes de comenzar a ver los casos de uso, se describirá la funcionalidad de los mensajes y los diferentes tipos existentes.

Debido a los requerimientos de la aplicación, no es necesario la comunicación entre nodos No Sumideros porque todo mensaje generado en uno de ellos tiene como destino el Sumidero. Esta particularidad simplifica mucho la implementación de la capa de red pero principalmente ahorra procesamiento al momento de rutear paquetes.

Se hace necesario aquí detallar los diferentes tipos de mensajes que pueden atravesar la capa de aplicación para argumentar lo mencionado líneas atrás, pero primero se detallan algunos parámetros claves en la capa de aplicación.

Una vez finalizada la inicialización de las capas inferiores se comienza a adquirir datos, para ello inician un timer que les indica el paso de los minutos.

Para el muestreo de sensores se utilizan dos parámetros que son únicos en la toda la red, o sea que los mismos una vez que se fijan, se realiza en todos los nodos y con los mismos valores. Los parámetros mencionados son el período de muestreo y el sensor a utilizar. En la capa de aplicación se denominan *T_MUESTREO* y *SENSOR_ID* respectivamente.

T_MUESTREO se mide en minutos y representa el tiempo entre dos medidas seguidas. Claro está que no se admite un valor nulo para este parámetro y el valor máximo es 255 que equivale a 4 horas y 15 minutos.

Se desprende de sus cotas que el valor mínimo que este campo puede tomar es 1 minuto, por lo tanto la mayor cadencia que puede haber de datos provenientes de un mismo nodo es de 60 datos por hora.

Por otro lado se definieron los identificadores de cada sensor, acordando con SIMSI la tabla a utilizar que es la siguiente:

SENSOR_ID	***SENSOR***
0x01	Temperatura
0x02	Humedad
0x03	Presión
0x04	Luminiscencia
0x05	...

Tabla 5.1: Tabla de identificadores de sensores

Estos parámetros son fijados a solicitud del usuario y son seteados mediante los mensajes de *CAMBIO_CONFIGURACIÓN* explicado más adelante.

A nivel de capa de aplicación, cada mote tiene asignado un número que lo identifica de

manera única en la red. Dicho número se le denomina *MOTE_ID* y debe ser configurado en cada mote a la hora de la programación.

El *MOTE_ID* es la identidad de cada mote, y se utiliza en la identificación de los mismos por parte del usuario así como también por parte de la base de datos al guardar la información de las medidas realizadas.

Hay 256 valores posibles para dicha constante ya que se trata de un valor de 8 bits. En el caso que una aplicación requiera más nodos integrantes en una hoja, es posible cambiar dicha restricción pero conllevaría un gasto más de recursos, razón por la cual motivó a seleccionar la cantidad de motes mencionada. Obviamente para las aplicaciones comunes el valor actual es suficiente.

Al nodo Sumidero se asignó de forma arbitraria el número 0, pero cabe destacar que este mote no realiza medidas de sensores sino que funciona como gateway y administrador central de la WSN.

A continuación se trata los diferentes tipos de mensajes que atraviesan la capa de aplicación y que hacen posible las configuraciones necesarias y el intercambio de información a través de esta capa.

***NACIMIENTO*:** Es generado por el mote recién asociado a la red, con el objetivo de informar al Sumidero y a la aplicación de la PC de su nueva asociación y estos agregarlos a la lista de motes integrantes de la red.

***CAMBIO_CONFIGURACIÓN*:** Cuando el Sumidero es informado del nacimiento de un nuevo mote, le envía en ese momento un mensaje del tipo *CAMBIO_CONFIGURACIÓN* para setearle los actuales valores de los parámetros de *T_MUESTREO* y *SENSOR_ID*, a lo que esto el mote recién nacido responde con un *ACK*. En los casos que el usuario ingresa una nueva configuración se envía un mensaje de *CAMBIO_CONFIGURACIÓN* pero esta vez es un mensaje de *BROADCAST*.

***SENSADO_PILAS*:** También son generados desde la aplicación de la PC y enviados en *BROADCAST*, a lo que cada mote responde con un único mensaje del mismo tipo pero con la medida del voltaje de las pilas. Si no se recibe una respuesta por parte de algún mote la aplicación puede volver a intentar una nueva solicitud pero esta vez solamente al mote en cuestión.

***MUERTE*:** Son generados por la capa de red y viajan hasta el sumidero a nivel de red. Son dirigidos a la capa de aplicación solamente cuando llegan al sumidero. Representan el abandono de la red por parte de los diferentes nodos ya sea por problemas en el hardware, culminación de la vida útil de la batería, etc.

***DATO*:** Son aquellos que transportan los valores de las medidas de los sensores. Vale aclarar que no hay mensaje de reconocimiento para estos casos.

***TOPOLOGÍA*:** Son generados por la aplicación de la PC y recibidos por el Sumidero. Estos son los únicos mensajes que el Sumidero no los reenvía a la red pero responde con un mensaje del mismo tipo por cada mote que tiene en su registro. En cada

mensaje se informa el *MOTE_ID* y el *MOTE_ID* que tiene asignado como padre (esta relación es utilizada a nivel de capa de red, pero es útil para visualizar la relación de dependencia en la interfaz de usuario). La idea principal de esta funcionalidad es que la aplicación de la PC pueda solicitar un resumen de la topología de red.

Para cada uno de los mensajes se asignó un identificador, y la tabla de asociación es la siguiente:

tipo_mensaje	***valor***
SENSADO_PILAS	0x01
ACK	0x02
NACIMIENTO	0x03
MUERTE	0x04
DATO	0x05
TOPOLOGIA	0x08
CAMBIO_CONFIGURACION	0xFF

Tabla 5.2: Tabla de identificadores para los distintos tipos de mensajes

A partir de la funcionalidad general definida en la sección anterior, se desprenden los siguientes casos de uso:

1. Nacimiento de un mote.
2. Solicitud de cambio de configuración.
3. Solicitud de sensado de pilas.
4. Adquisición de datos.
5. Muerte de un mote.
6. Solicitud de topología.

A continuación se describen cada uno de los casos de uso recientemente mencionados:

5.2.1. Nacimiento de un mote

Una vez arrancado el mote, debe invocar la inicialización de la capa de red, lo que desencadena en la misma todos los eventos necesarios para funcionar correctamente.

Al finalizar esta inicialización la aplicación debe informar al Sumidero que acaba de nacer, esto lo hace mediante el envío de un mensaje de *NACIMIENTO* creado exclusivamente para este fin. En el mensaje le informa además de la dirección de red, el *MOTE_ID* asignado.

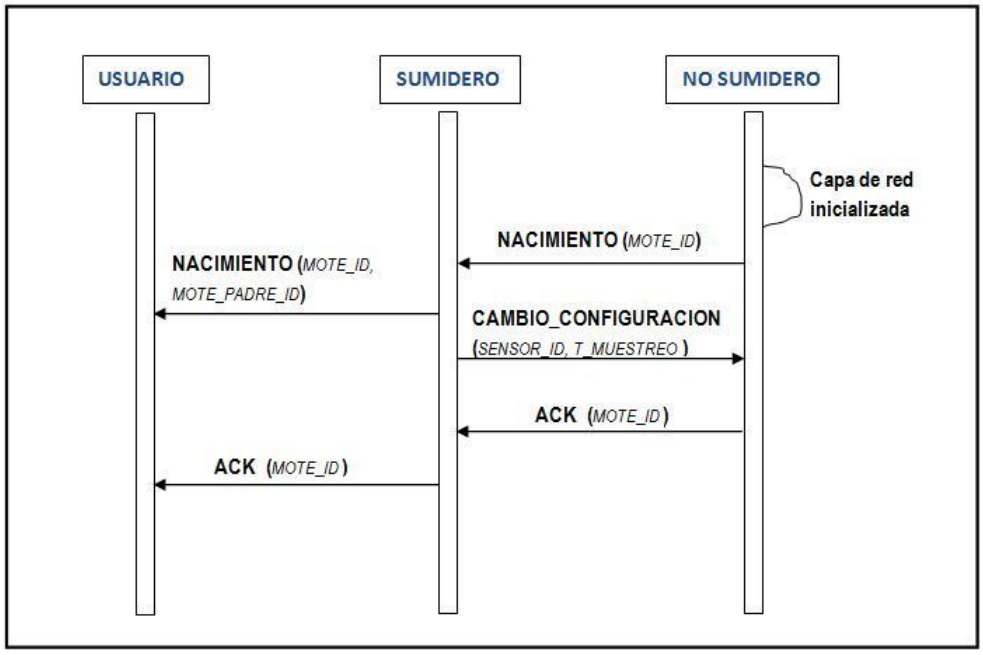


Figura 5.2: Diagrama de secuencia para el caso de uso de Nacimiento de un mote

5.2.2. Solicitud de cambio de configuración

Cada vez que el usuario cambie la configuración de la red (frecuencia de muestreo y sensor a utilizar) se genera un mensaje en la aplicación de la PC, el cual es recibido por el Sumidero y reenviado a todos los motes de la red. Cada mote No Sumidero al detectar que se trata de un mensaje de *CAMBIO_CONFIGURACIÓN*, hace los cambios de período de muestreo y sensor a utilizar y envía un reconocimiento con destino el Sumidero.

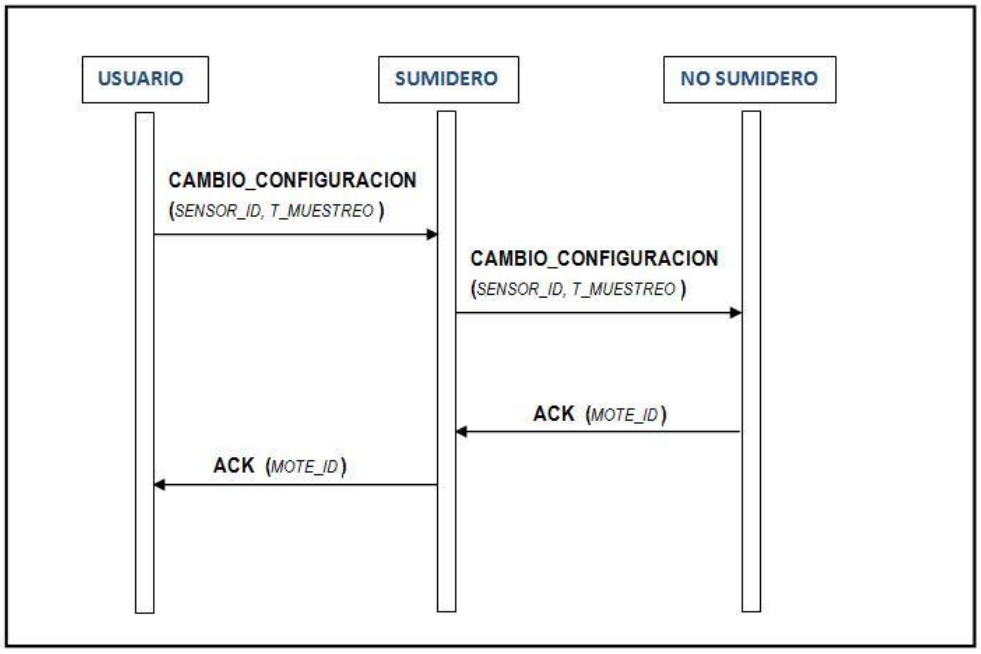


Figura 5.3: Diagrama de secuencia para el caso de uso de Solicitud de cambio de configuración

5.2.3. Solicitud de sensado de pilas

Cuando el usuario lo desee, podrá solicitar la medida del voltaje de las pilas de cada mote. Ante este evento, se genera un mensaje desde la aplicación de la PC, con destino todos los motes de la red, solicitando la información deseada.

Al recibir dicho mensaje, el mote toma la medida del sensor adecuado y envía el dato al Sumidero.

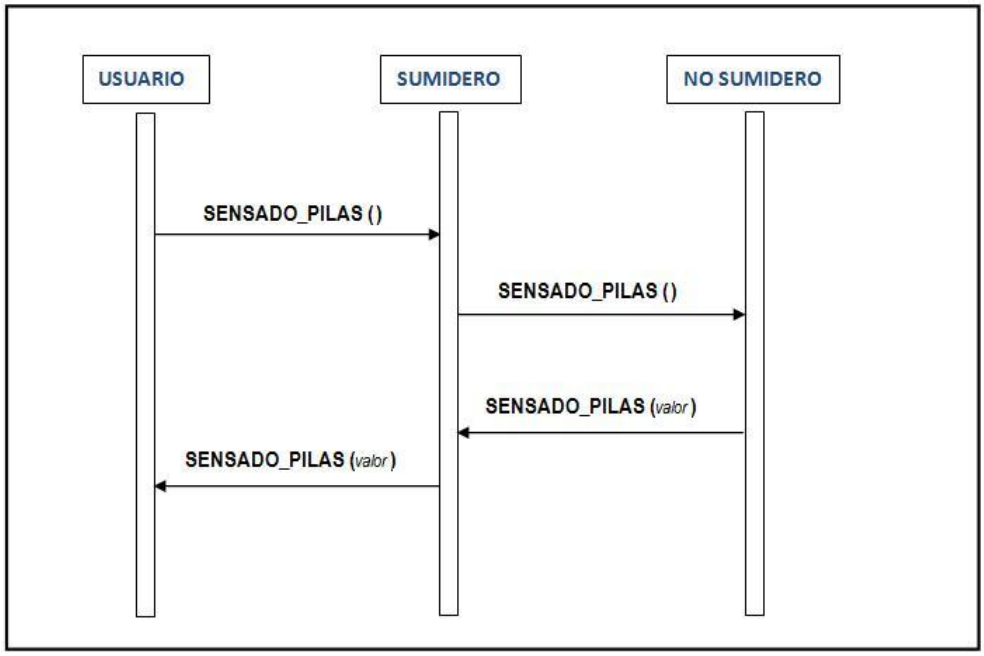


Figura 5.4: Diagrama de secuencia para el caso de uso de Solicitud de sensado de pilas

5.2.4. Adquisición de datos

La capa de aplicación del no sumidero debe disponer de un Timer que periódicamente indicará el momento en el que se debe tomar la medida. El período es el configurado en el caso de uso de Cambio de Configuración.

Una vez tomada la medida, se envía la información al Sumidero.

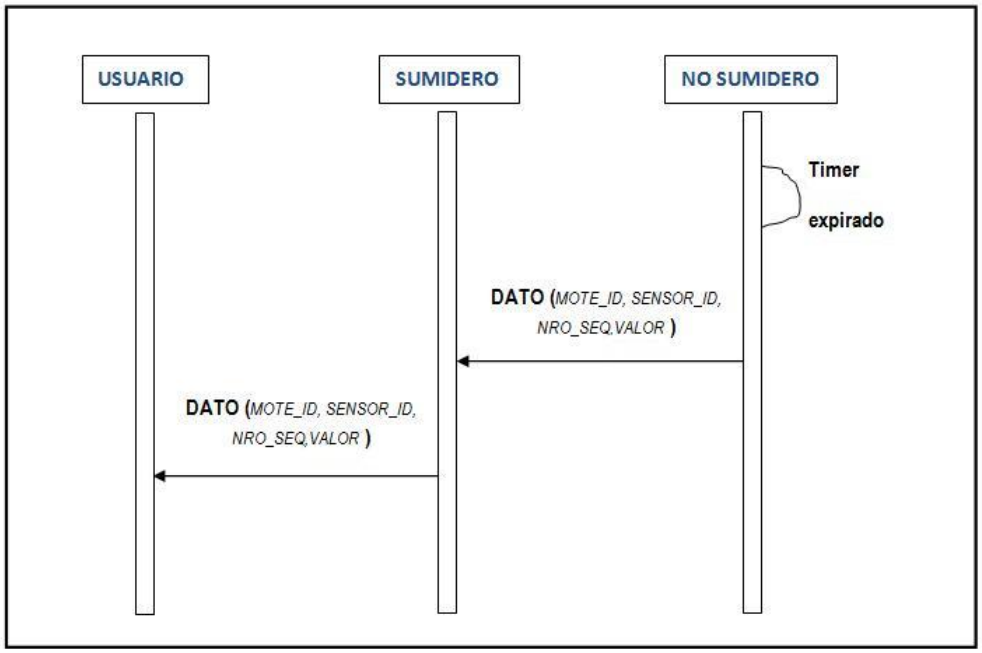


Figura 5.5: Diagrama de secuencia para el caso de uso de Adquisición de datos

5.2.5. Muerte de un mote

Como ya fue mencionado anteriormente, cada mote será identificado en la red por un número único denominado *MOTE_ID*. Dicho número es utilizado entre otras cosas para identificar los motes que se acaban de asociar a la red o los que por alguna razón se desasocian de la red (*MUERTE* de un mote). Las razones a las que se hace referencia pueden ser varias, entre ellas pueden estar el agotamiento de baterías, desperfecto del hardware, etc.

Dado que el mote Sumidero lleva un registro de todos los motes existentes en la red, al nacer o morir uno de ellos, se debe reflejar en la tabla agregándolo o eliminándolo según corresponda.

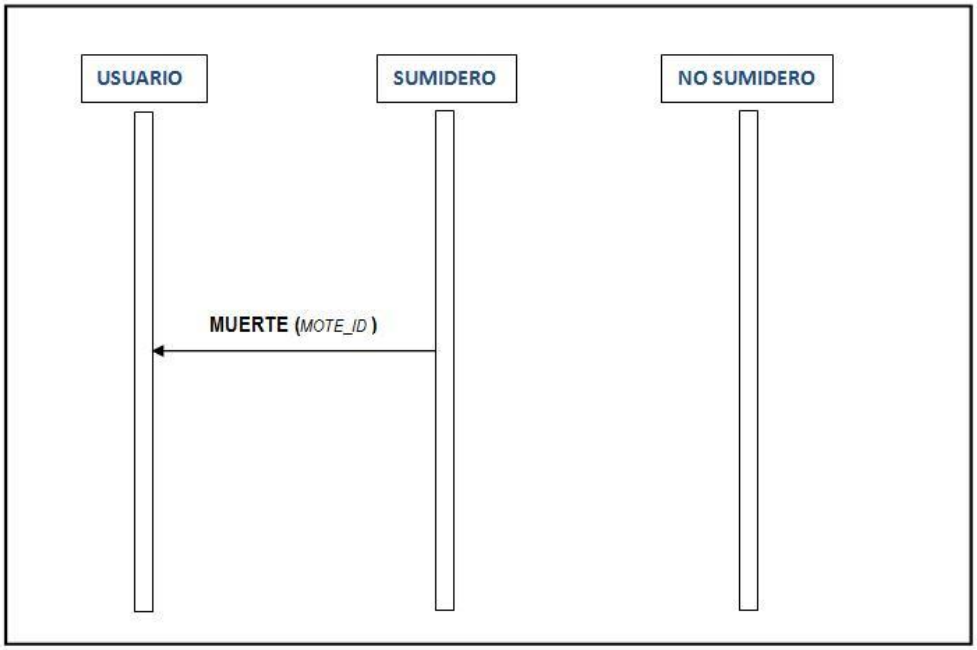


Figura 5.6: Diagrama de secuencia para el caso de uso de Muerte de un mote

El nodo sumidero es notificado de la muerte de un mote a nivel de capa de red, luego de esto el mensaje se dirige a la capa de red y a partir de este instante continúa el diagrama de secuencia anterior[2].

[2]Por más detalles dirigirse al capítulo 6

5.2.6. Solicitud de topología

La solicitud de topología es generada por la aplicación de la PC con el objetivo de recibir un resumen de los motes que se encuentran asociados a la red. Ante la recepción de una solicitud de topología el Sumidero responde con un mensaje del mismo tipo por cada mote que tiene en su registro.

En cada mensaje además del *MOTE_ID* del mote que actualmente está asociado a la red, se incluye también el *MOTE_ID* del mote padre.

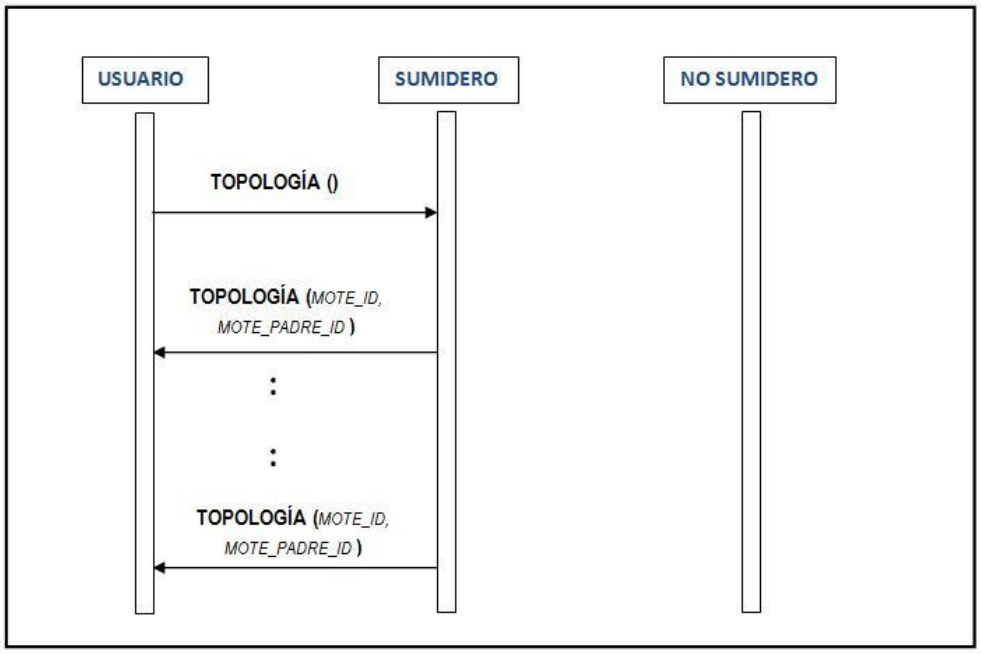


Figura 5.7: Diagrama de secuencia para el caso de uso de Solicitud de topología

5.3. Estructura de datos

A partir del estudio de la información necesaria para llevar a cabo la aplicación es que se definieron dos tipos diferentes de mensajes que se manejarán en la capa de aplicación y la interfaz PC-mote.

Los mensajes utilizados están diferenciados según donde se origina el mensaje.

Los mensajes provenientes del PC llevarán información de:

- Cambio de configuración
- Solicitud de sensado de pilas
- Solicitud de topología

En cambio la información que proviene del mote y se dirige a la PC puede ser:

- Dato del sensor utilizado
- Dato de sensado de pilas
- ACK
- Topología
- Nacimiento de un mote
- Muerte de un mote

Según lo planteado anteriormente se nombraron los mensajes del siguiente modo:

mensaje_pc_t
: representa el mensaje generado en la PC y dirigido a la red

mensaje_mote_t
: representa el mensaje generado en la red y dirigido a la PC

La declaración de mensaje_pc se describe en la siguiente figura:

```
typedef struct mensaje_pc {
  nx_uint8_t tipo_mensaje;
  nx_uint8_t sensor_id;
  nx_uint8_t t_muestreo;   // periodo de muestreo en minutos
} mensaje_pc_t;
```

Figura 5.8: Estructura de datos definida en NesC para el caso de mensaje_pc

El campo tipo_mensaje indicará el tipo de mensaje y se diferenciarán según tabla:

tipo_mensaje	*valor*
SENSADO_PILAS	0x01
TOPOLOGIA	0x08
CAMBIO_CONFIGURACION	0xFF

Tabla 5.3: Tabla de identificadores para los distintos tipos de mensajes originados en la PC

El campo sensor_id se regirá por la tabla 5.2 que puede ser modificada cada vez que se agregue un nuevo sensor no declarado.

El campo t_muestreo contiene el valor del tiempo de muestreo a configurar, se medirá en minutos y nunca podrá ser igual a cero mientras el tipo_mensaje sea igual 0xFF.

La declaración de mensaje_mote se describe en la siguiente figura:

```
typedef struct mensaje_mote {
  nx_uint8_t tipo_mensaje;
  nx_uint8_t mote_id;
  nx_uint8_t campo1;
  nx_uint8_t campo2;
  nx_uint16_t dato;
} mensaje_mote_t;
```

Figura 5.9: Estructura de datos definida en NesC para el caso de mensaje_mote

El campo mote_id indica el número de mote en donde se origina el mensaje.

Los dos siguientes campos son utilizados dependiendo del tipo de mensaje.

En los casos de Sensado_Pilas, *ACK* y Muerte; el campo1 y campo2 no son utilizados. En cambio, en caso de Nacimiento campo1 almacena el *MOTE_ID* del mote padre. Si se

trata de *DATO*, campo1 indica el *SENSOR_ID* al que corresponde la medida y campo2 contiene el número de secuencia correspondiente a dicho mensaje.

El campo dato almacena el valor de la medida del sensor.

5.4. Implementación

El primer punto a tener en cuenta es que en la capa de aplicación los nodos se diferencian en dos tipos: mote sumidero y el mote no sumidero. Por lo tanto se programan diferente para cada uno de los casos, contemplando las funcionalidades de cada uno.

El mote SUMIDERO debe actuar como gateway entre la PC (o la red que esté conectado) y nuestra red de motes NO SUMIDEROs. Tiene la capacidad de conocer toda la red de motes, sus IDs, los motes padre de cada uno, sus estados y sus direcciones de capa de red.

En cambio el mote NO SUMIDERO solamente se encarga de tomar las medidas y enviarlas al SUMIDERO, a una cadencia que se le informará desde la red externa y que él debe ser apto de recibir.

Cada mote NO SUMIDERO se identificará en la capa de aplicación por un número único y constante que se grabará en el mote a la hora de programarlo. Dicho número será distinto de cero, ya que el cero será reservado para el SUMIDERO.

5.4.1. Sumidero

En esta sección se describirá la aplicación del SUMIDERO.

Se utilizan tres archivos:

- Sumidero.nc
- SumideroM.nc
- sumidero.h

El primero es el de configuración donde se realiza los cableados de los componentes. El segundo es el módulo de la aplicación, donde se programan todas las funcionalidades. Por último se tiene sumidero.h en donde se define las diferentes constantes y estructuras a utilizar.

5.4.1.1. Sumidero.nc

Se presenta en este archivo la declaración de los componentes a utilizar: SerialActiveMessageC y Red, representando a la interfaz USB y la capa de red respectivamente.

Para el componente SerialActiveMessageC se utilizan 3 interfaces que se detallan a continuación:

```
SumideroM.SplitControl->SerialActiveMessageC;
SumideroM.UartReceive->SerialActiveMessageC;
SumideroM.AMSend -> SerialActiveMessageC.AMSend[100];
```

Figura 5.10: Cableado de las interfaces en la aplicación Sumidero

La interfaz SplitControl se utiliza para iniciar y detener el componente.

La interfaz UartReceive provee los eventos necesarios para indicar el arribo de mensajes.

Por último la interfaz AMSend se utiliza para enviar mensajes por el USB.

También se cablean las dos interfaces de la capa de red: RED_DATA, RED_MNGT y SplitControl (que se renombra como Init para evitar conflicto con la interfaz SplitControl de SerialActiveMessageC).

```
// Interfaces de capa de red
SumideroM.RED_DATA -> RedSumidero;
SumideroM.RED_MNGT -> RedSumidero;
SumideroM.Init -> RedSumidero.SplitControl;
```

Figura 5.11: Cableado de las interfaces de datos en la aplicación Sumidero

La interfaz RED_DATA provee los comandos y eventos relacionados con el flujo de datos.

La interfaz RED_MNGT provee únicamente un comando: el encargado de pasar el valor del tiempo de muestreo a la capa de red.

La interfaz SplitControl está relacionada con la inicialización y detención del componente.

En la siguiente figura se ilustra gráficamente el cableado de la aplicación del sumidero, descripto líneas atrás.

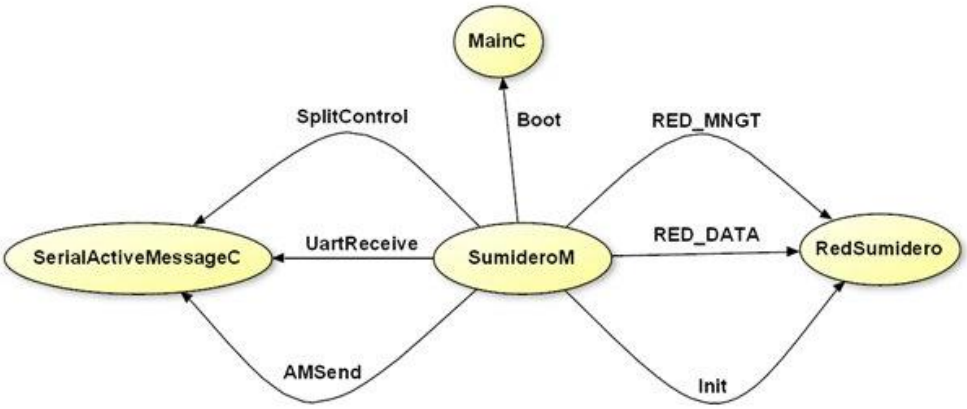


Figura 5.12: Componentes, sus interfaces y el cableado de la aplicación

5.4.1.2. SumideroM.nc

Se comenzará nombrando y describiendo las *variables globales* que se utilizan en la aplicación:

- `message_t saliendo`: Estructura utilizada para encapsular los mensajes salientes del mote, pueden ser hacia la PC o hacia la red.

- `message_t msj_nacimiento`: Estructura utilizada para encapsular exclusivamente los mensajes de nacimiento.

- `mensaje_mote_t topología`: Estructura utilizada solamente para el envio de mensajes en respuesta de una solicitud de topología.

- `mensaje_pc_t mensaje_conf`: Estructura utilizada solamente para el envío de mensajes de configuración que se generan en el Sumidero y tienen como destino únicamente los motes que acaban de unirse a la red, con el fin de actualizar su configuración a la actual.

- `paridad_t motes[256]`: Arreglo en donde el mote sumidero almacena la información de la red: estado de cada mote y su mote padre. La posición i está reservada exclusivamente para el mote i.

- `uint8_t mote`: Indice de la estructura motes. Para recorrer el arreglo se utiliza este índice, es útil en el caso que se solicita que se envíe la topología de la red; para lo cual se va recorriendo el arreglo y enviando el número de cada mote con su respectivo padre, de esta manera se puede realizar un esquema de la red.

Se utilizan dos *funciones* y una *tarea*:

- `void enviar_topologia(uint8_t motein)`: Es la función encargada de recorrer la estructura motes e ir enviando los mensajes por el USB para informar del estado de cada mote activo. Si el mote está vivo (conectado a la red) se envía un mensaje de *TOPOLOGÍA*; en caso contrario se ignora. Por lo tanto si en la red hay solamente dos motes "vivos", cuando se solicite la topología por parte de la PC, ésta recibirá dos mensajes: cada uno con la información de cada mote. Si hay conectados 90 motes; recibirá 90 mensajes de *TOPOLOGÍA*.

- `void procesar_nacimiento(mensaje_mote_t* mensajenacim)`: Es la función que procesa los mensajes del tipo NACIMIENTO provenientes de la red.
  Al llegar un mensaje tal, ésta actualiza la estructura motes y luego envía un mensaje de NACIMIENTO a la PC informando el número de mote recién nacido y el número del mote padre.

- `task void procesar_muerte()`: Dicha tarea se encarga de tomar la dirección de red que se encuentra almacenada en la variable DirRedOrigen y quitar de estructura motes el mote con dicha dirección de red (acaba de morir) y todos los motes que dependieran de su conexión al Sumidero.

Un punto importante en la capa de aplicación es que no provee ninguna interfaz, por lo que no hay comandos a implementar. Lo que se debe encargar la capa de aplicación es de implementar los eventos que disparan las interfaces que utiliza. Estos eventos son:

- event void Boot.booted()
- event void SplitControl.startDone(error_t result)
- event void Init.startDone(error_t result)
- event void SplitControl.stopDone(error_t result)
- event void Init.stopDone(error_t result)
- event message_t *UartReceive.receive[am_id_t id](message_t* msg, void* payload, uint8_t len)
- event error_t RED_DATA.indicationsum(uint32_t DirOrigen, mensaje_mote_t* mensajemote, uint8_t codigo_msj)

- event void AMSend.sendDone(message_t* msg, error_t result)

Para una comprensión a nivel general del funcionamiento de la capa de aplicación se describe la secuencia de eventos y comandos que se van invocando a medida que transcurren los eventos:

1. Ante el evento Boot.booted () llamada de inicialización del componente USB (SerialActiveMessageC).

2. Ante el evento SplitControl.startDone (SerialActiveMessageC) iniciado con éxito llamada de inicialización de capa de Red con el comando SplitControl.start(). En caso que la inicialización no se realizara con éxito, se vuelve a intentar.

3. Ante el evento de capa de red iniciada con éxito: Init.startDone queda a la espera de arribos de mensajes provenientes de la red o de la PC.

Los anteriores *eventos* intervienen en la fase de inicialización de la capa de aplicación, pero luego tenemos dos eventos relacionados con el arribo de mensajes:

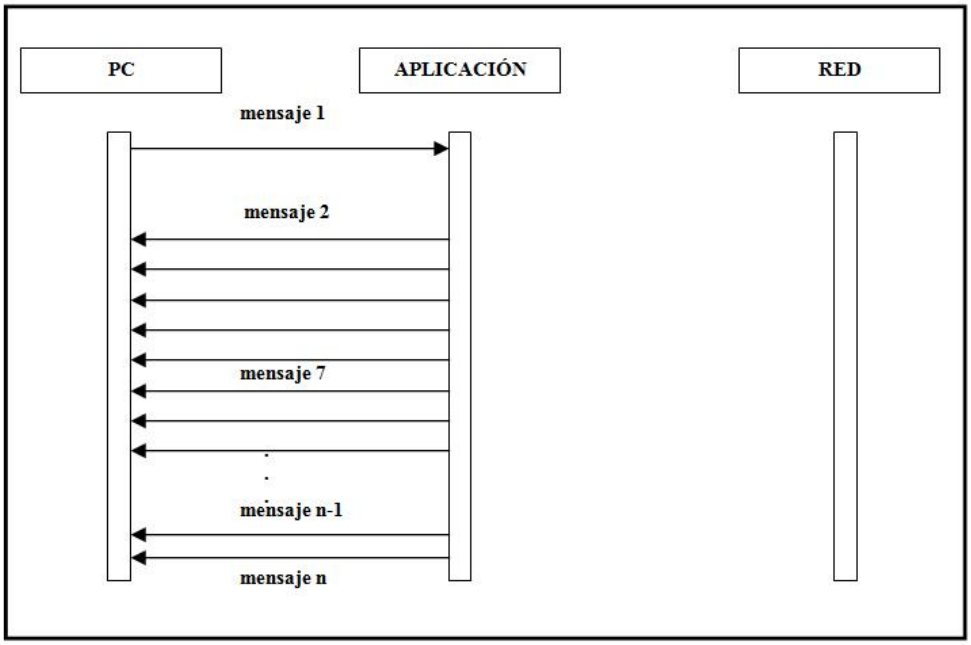
- `event message_t *UartReceive.receive[am_id_t id](message_t* msg, void* payload, uint8_t len)`: Este evento indica el arribo de mensajes desde el puerto USB, la clave aquí es que este evento se dispara únicamente si coincide con el ID que se adjunta en la interfaz, que como se mencionó antes, los mensaje_pc tienen un identificador igual a 100. Al dispararse este evento, primero hay que analizar de qué tipo de mensaje se trata para hacerle un tratamiento adecuado: si es topología se llama la función enviar_topologia (); si es un cambio de configuración, se debe guardar los valores de la nueva configuración pero a su vez se reenvía el mensaje a la red. En cambio si se trata de una solicitud de sensado de pilas simplemente se reenvía el mensaje a la red.

- `event error_t RED_DATA.indicationsum(uint32_t DirOrigen, mensaje_mote_t* mensajemote, uint8_t codigo_msj)`: Este evento corresponde a la llegada de un mensaje desde la red, del tipo mensaje_mote. Ante un mensaje, primero se debe analizar si se trata del arribo de un mensaje de muerte y en ese caso invocar la tarea procesar_muerte(). En caso contrario se verifica si es uno de nacimiento y entonces se debe llamar la función procesar_nacimiento() Por otro lado si no se trata de ninguno de estos mensajes, se lo reenvía por el USB.

En los siguientes diagramas se muestra la secuencia de mensajes entre la PC o usuario y la capa de aplicación y ésta hacia la capa de red.

La tabla asociada describe el tipo de mensaje que es cada uno; y los campos que se utilizan con los valores cargados.

Solicitud de topología - por parte del usuario.

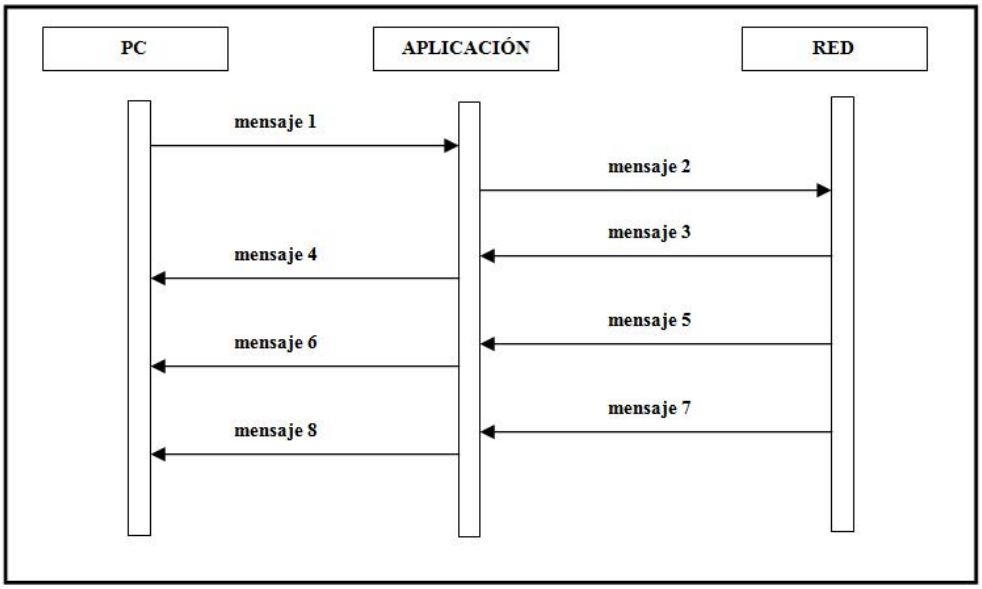


	mensaje 1	mensaje 2	mensaje 3	...	mensaje n-1	mensaje n
mensaje_pc_t	x					
tipo_mensaje	TOPOLOGIA					
sensor_id	—					
t_muestreo	—					
mensaje_mote_t		x	x	x	x	x
tipo_mensaje		TOPOLOGIA	TOPOLOGIA	TOPOLOGIA	TOPOLOGIA	TOPOLOGIA
mote_id		Mote_ID	Mote_ID	Mote_ID	Mote_ID	Mote_ID
campo1		Mote_ID	Mote_ID	Mote_ID	Mote_ID	Mote_ID
campo2		—	—	—	—	—
dato		—	—	—	—	—

Tabla 5.4: Campos utilizados en los mensajes que intervienen en solicitud de topología

Nota: En este caso se tiene n-1 motes “vivos” para los cuales se envía un mensaje de *TOPOLOGÍA* para cada uno con su número y el de su padre.

Solicitud de sensado de pilas - por parte del usuario.

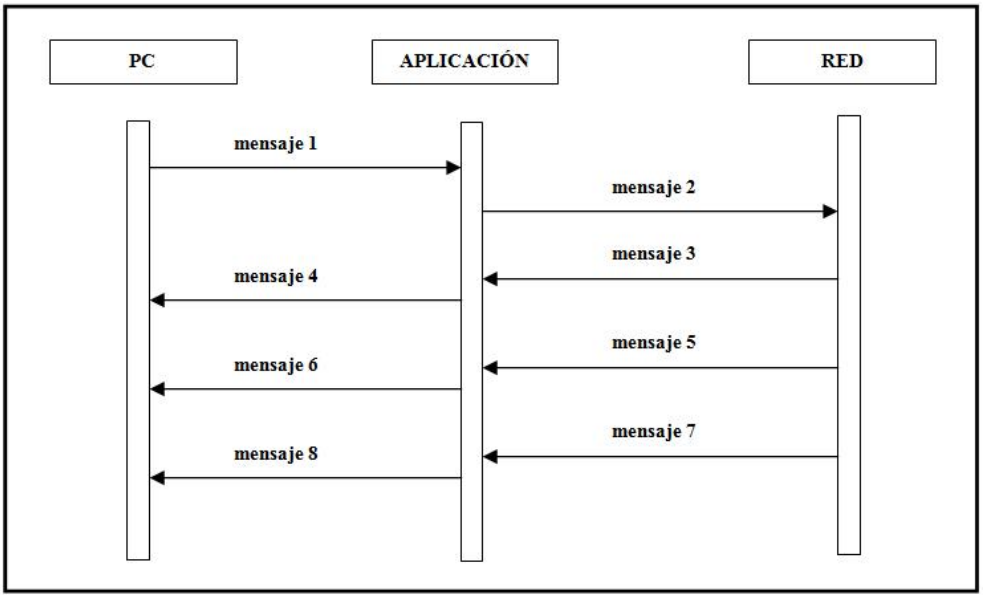


	mensaje 1	mensaje 2	mensaje 3	mensaje 4	mensaje 5
mensaje_pc_t	X	X			
tipo_mensaje	SENSADO_PILAS	SENSADO_PILAS			
sensor_id	—	—			
t_muestreo	—	—			
mensaje_mote_t			X	X	X
tipo_mensaje			SENSADO_PILAS	SENSADO_PILAS	SENSADO_PILAS
mote_id			Mote_ID	Mote_ID	Mote_ID
campo1			—	—	—
campo2			—	—	—
dato			valor	valor	valor

Tabla 5.5: Campos utilizados en los mensajes que intervienen en una solicitud de sensado de pilas

Nota:
1 - La secuencia continua hasta que lleguen todos los mensajes de todos los motes.
2 - En este caso la capa de aplicación hace un by-pass de los mensajes hacia un lado y hacia el otro; por lo tanto los pares de mensajes 1-2, 3-4, 5-6, 7-8 son el mismo.
3 - El mensaje 2 es un broadcast desde el SUMIDERO.

Cambio de configuración - por parte del usuario.



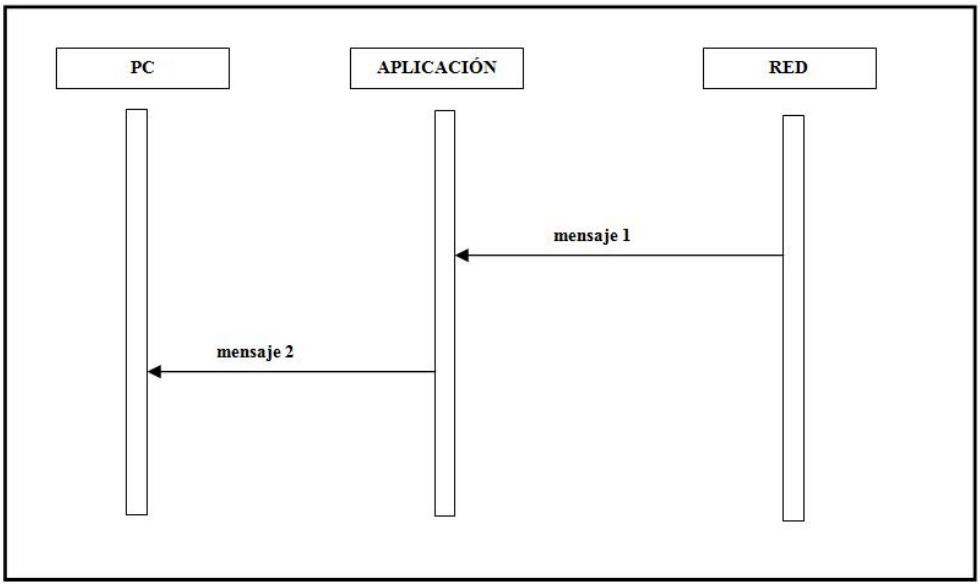
	mensaje 1	mensaje 2	mensaje 3	mensaje 4	mensaje 5
mensaje_pc_t	X	X			
tipo_mensaje	CAMBIO_CONF(*)	CAMBIO_CONF(*)			
sensor_id	SENSOR_ID	SENSOR_ID			
t_muestreo	minutos	minutos			
mensaje_mote_t			X	X	X
tipo_mensaje			ACK	ACK	ACK
mote_id			Mote_ID	Mote_ID	Mote_ID
campo1			—	—	—
campo2			—	—	—
dato			—	—	—

Tabla 5.6: Campos utilizados en los mensajes que intervienen en una solicitud de cambio de configuración

(*) - CAMBIO_CONFIGURACIÓN

Nota:
1 - El mensaje 2 es un broadcast desde el SUMIDERO.
2 - En este caso la capa de aplicación hace un by-pass de los mensajes hacia un lado y hacia el otro; por lo tanto los pares de mensajes 1-2, 3-4, 5-6, 7-8 son el mismo.

Nacimiento de un mote

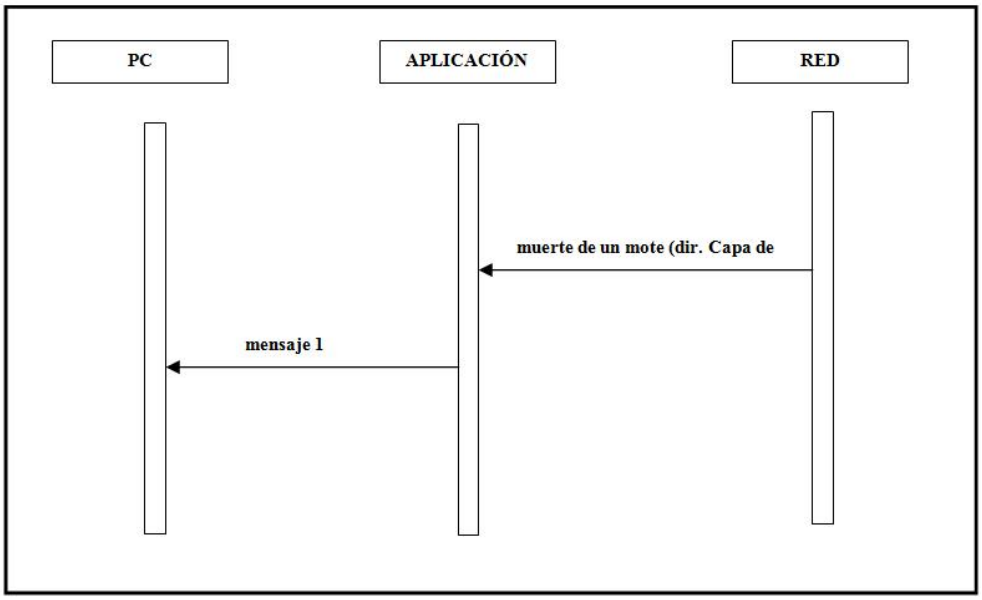


	mensaje 1	mensaje 2
mensaje_pc_t		
tipo_mensaje		
sensor_id		
t_muestreo		
mensaje_mote_t	X	X
tipo_mensaje	NACIMIENTO	NACIMIENTO
mote_id	Mote_ID	Mote_ID
campo1	—	Mote_ID_Padre
campo2	—	—
dato	—	—

Tabla 5.7: Campos utilizados en los mensajes que intervienen en el nacimiento de un mote

Nota:
1 - Entre el arribo del mensaje 1 y el envío del mensaje 2 la capa de aplicación debe actualizar el arreglo donde se almacenan los estados de los motes.
2 - El *MOTE_ID* del padre se toma del registro existente en el Sumidero.

Muerte de un mote

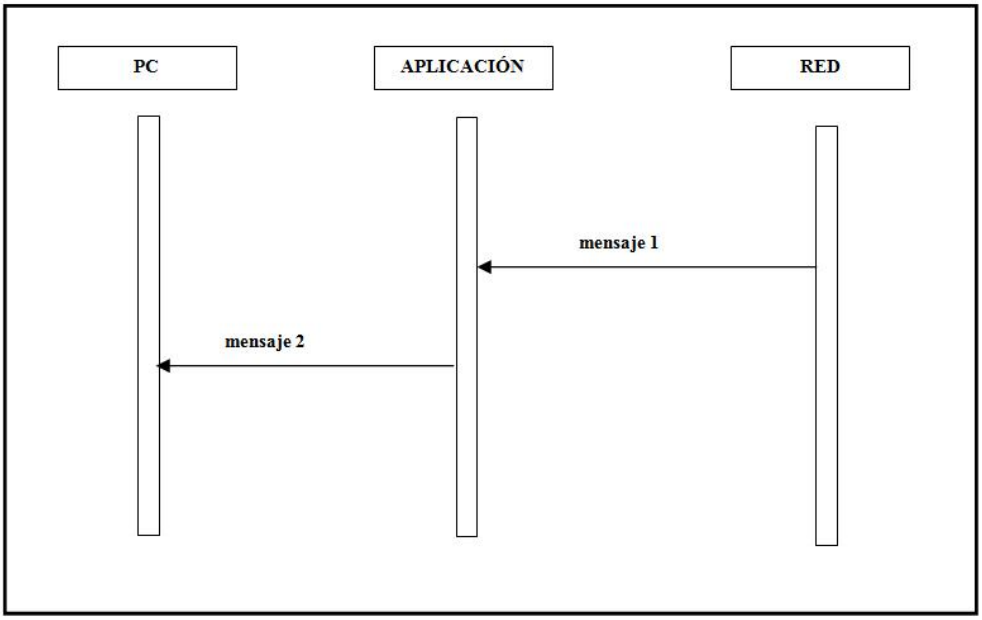


	mensaje 1
mensaje_pc_t	
tipo_mensaje	
sensor_id	
t_muestreo	
mensaje_mote_t	x
tipo_mensaje	MUERTE
mote_id	Mote_ID
campo1	—
campo2	—
dato	—

Tabla 5.8: Campos utilizados en el mensaje utilizado en la muerte de un mote

Nota: En el evento inicial no se intercambia un mensaje, solo es un evento donde la capa de red notifica la muerte de un mote indicando su dirección de capa de red. Luego la capa de aplicación lo busca en el arreglo motes (la búsqueda la hace según la dir. de capa de red) y envía a la PC el mensaje 1 pasando el número de mote asociado.

Dato - originado en un mote NO SUMIDERO



	mensaje 1	mensaje 2
mensaje_pc_t		
tipo_mensaje		
sensor_id		
t_muestreo		
mensaje_mote_t	X	X
tipo_mensaje	DATO	DATO
mote_id	Mote_ID	Mote_ID
campo1	SENSOR_ID	SENSOR_ID
campo2	$N^o Seq.$	$N^o Seq.$
dato	muestra	muestra

Tabla 5.9: Campos utilizados en los mensajes de datos de sensores

Nota: Ante los mensajes del tipo *DATO*, la aplicación del SUMIDERO actúa como by-pass.

5.4.1.3. sumidero.h

En este archivo se definen todas las constantes utilizadas en la aplicación Sumidero y todas las estructuras de datos que son específicas de la aplicación.

Las constantes son las que ya se mencionaron en las tablas 5.2 y la 5.2. También se declaran dos direcciones de interés que son:

- `DIRSUMIDERO:` constante igual a 0, que es la dirección de red del Sumidero.
- `BROADCAST:` constante igual a 0xFFFFFFFF, la cual es la dirección de red de broadcast.

Se declaran también las estructuras ya descriptas anteriormente: mensaje_pc_t y mensaje_mote_t en la sección 5.3; pero además se incluye la estructura paridad_t. Este tipo de dato fue creado para tener un registro de cada mote de la red, su estado y su dirección de red:

- `motepadre_id:` Es el id del mote padre del nodo en cuestión.
- `estado:` Indica el estado del mote; un 1 indica que está conectado y un 0 indica que está desconectado.
- `dir_red:` Es la dirección de red del mote.

5.4.2. No Sumidero

Se utilizan tres archivos:

- NoSumidero.nc
- NoSumideroM.nc
- nosumidero.h

El primero es la configuración donde se hace los cableados de los componentes. El segundo es nuestro módulo, donde se programan todas las funcionalidades. Por último nosumidero.h es donde se definen las diferentes constantes y estructuras a utilizar.

5.4.2.1. NoSumidero.nc

A continuación se detallan los componentes a utilizar: TimerMilliC(), DemoSensorC(), VoltageC() y Red.

El componente TimerMilliC maneja los tiempos; el DemoSensor() maneja el sensor de temperatura interno del microprocesador y VoltageC() es el sensor de Voltaje.

Los cableados de los mencionados componentes se pueden ver en la siguiente imagen:

```
// Interfaces
NoSumideroM.Boot -> MainC;
NoSumideroM.SplitControl -> Red;
NoSumideroM.RED_DATA -> Red;
NoSumideroM.Timer0 -> Timer0;
NoSumideroM.Lectura -> SensorT;
NoSumideroM.ReadV -> SensorV;
NoSumideroM.AMSend -> SerialActiveMessageC.AMSend[100];
```

Figura 5.13: Cableado de interfaces en aplicación de no Sumidero

La interfaz RED_DATA provee los comandos y eventos relacionados con el flujo de datos.

La interfaz SplitControl está relacionada con la inicialización y detención del componente.

La interfaz Lectura, al igual que la interfaz ReadV proveen los comandos y eventos necesarios para realizar las lecturas de los respectivos sensores.

El componente SerialActiveMessageC es incluido solamente a efectos de utilizar el comando getPayload que provee la interfaz AMSend que implementa. Este comando es

utilizado para extraer el campo payload de los mensajes del tipo message_t.

En la siguiente figura se ilustra gráficamente el cableado de la aplicación del no sumidero, descripto líneas atrás.

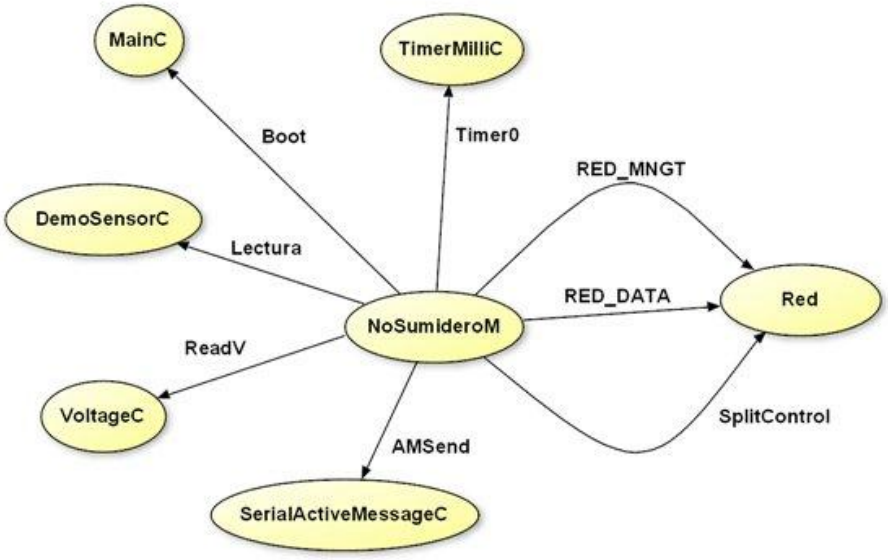


Figura 5.14: Componentes, sus interfaces y el cableado de la aplicación

5.4.2.2. NoSumideroM.nc

Se comenzará nombrando y describiendo las *variables globales* que se utilizarán en la aplicación:

- `uint8_t t_muestreo`: Variable que almacena el valor del período de muestreo en minutos. Esta variable se actualiza cada vez que llegue un mensaje de *CAMBIO_CONFIGURACIÓN*.

- `uint8_t minutos`: Variable que se actualiza cada minuto, y se resetea frente a la llegada de un mensaje de *CAMBIO_CONFIGURACIÓN*. Lleva la cantidad de minutos transcurridos desde el último instante de muestreo.

- `uint8_t seq_num`: Número de secuencia que se incrementa y se incluye en cada mensaje de dato enviado al sumidero. Es una forma de identificar los mensajes.

- `uint8_t sensor_id`: Valor entero que identifica el sensor que se debe muestrear. Esta variable se actualiza cada vez que llegue un mensaje de *CAMBIO_CONFIGURACIÓN*.

- `mensaje_pc_t* mensaje_recibido`: Puntero para manejar los mensajes recibidos desde el sumidero.

- `mensaje_mote_t respuesta`: Estructura que se utiliza para enviar los reconocimientos y los mensajes de nacimiento y muerte.

- `mensaje_mote_t muestra`: Estructura utilizada para almacenar los mensajes que contienen los datos del sensado de voltaje y del sensor que se está utilizando en ese momento.

- `message_t mensaje_salida`: Estructura del tipo message_t que encapsula los mensajes salientes con destino el nodo SUMIDERO.

- `message_t mensaje_ack`: Estructura del tipo message_t que encapsula los mensajes salientes con destino el nodo SUMIDERO y corresponden a los mensajes de reconocimiento que se envían solamente en los casos de *CAMBIO_CONFIGURACION*.

La secuencia de inicialización de la capa de red es:

1. Ante el evento Boot.booted (): llamada de inicialización de capa de Red.

2. Ante el evento de capa de red iniciada con éxito: se envía mensaje del tipo *NACIMIENTO* hacia el SUMIDERO y se dispara el timer con el período de muestreo que por defecto es de 10 minutos.

3. El timer que se utiliza expira cada medio minuto, y cada vez que lo hace se verifica si el tiempo transcurrido desde la última muestra equivale al tiempo de muestreo; en caso afirmativo se invoca la interfaz que toma la lectura del sensor de temperatura.

4. Cuando esta lectura finaliza con el evento Lectura.readDone(), se envía un mensaje del tipo *DATO* al Sumidero.

5. Si la lectura que finaliza es la lectura del sensor de voltaje: ReadV.readDone(), se envía al Sumidero un mensaje del tipo *SENSADO_PILAS*.

6. Cuando se dispara el evento *RED_DATA*.indicationnosum; significa que ha llegado un mensaje desde el Sumidero. Este mensaje puede ser de 2 tipos: solicitud de sensado de pilas, a lo que se invoca el comando read() de la interfaz ReadV. El otro tipo de

mensaje puede ser un cambio de configuración, a lo que se guardan los valores de interés y se contesta con un mensaje de reconocimiento.

5.4.2.3. nosumidero.h

En este archivo se definen las constantes a utilizar, en donde se pueden ver todos los valores de los tipos de mensajes y sensor_id.

También está definido el valor de *T_MUESTREO* por defecto. Cada vez que un mote nace carga en su variable de t_muestreo este valor, por eso es necesario enviarle un mensaje de configuración a todo mote recién nacido para informarle de la configuración actual.

La constante tal vez más importante es el *MOTE_ID*, la cual hay que cambiar cada vez que se programe un nuevo mote.

5.5. Desarrollo de Aplicación PC desarrollada para testeo

Como fue mencionado en párrafos anteriores, fue necesario crear una aplicación que corriera sobre el PC, la cual fuera encargada de monitorear el flujo y secuencia de mensajes que transitan por la interfaz USB entre el mote Sumidero y el PC respetando el protocolo definido para dicha interfaz.

De esta manera se podría gestionar la red enviando órdenes que se ejecutaran sobre la red y recibiendo mensajes provenientes de la misma.

Como primer paso antes de comenzar a desarrollar la aplicación de prueba, se debió estudiar los casos de uso ya definidos para la red y luego diseñar una primera estructura de la interfaz gráfica.

A modo de resumen, los comandos u órdenes enviadas por el usuario deben ser las siguientes:

- Cambio de configuración
- Solicitud de sensado de pilas
- Solicitud de topología

Pero a su vez el usuario debe ser capaz de visualizar todos los mensajes recibidos. Resaltando en este punto que esta aplicación tiene como único fin validar las pruebas de funcionamiento de la red, ya que la aplicación definitiva en la cual se incluye además una base de datos está a cargo del grupo SIMSI.

Los mensajes que el usuario recibe son del tipo:

- Dato
- Reconocimiento
- Respuesta a una solicitud de sensado de pilas
- Respuesta a una solicitud de topología
- Nacimiento de un nuevo mote
- Muerte de un mote

Luego de estudiados todos los casos de uso que la aplicación debería ser capaz de contemplar, se llegó a que para una versión de pruebas, la interfaz de usuario siguiente cubre todos los requisitos necesarios:

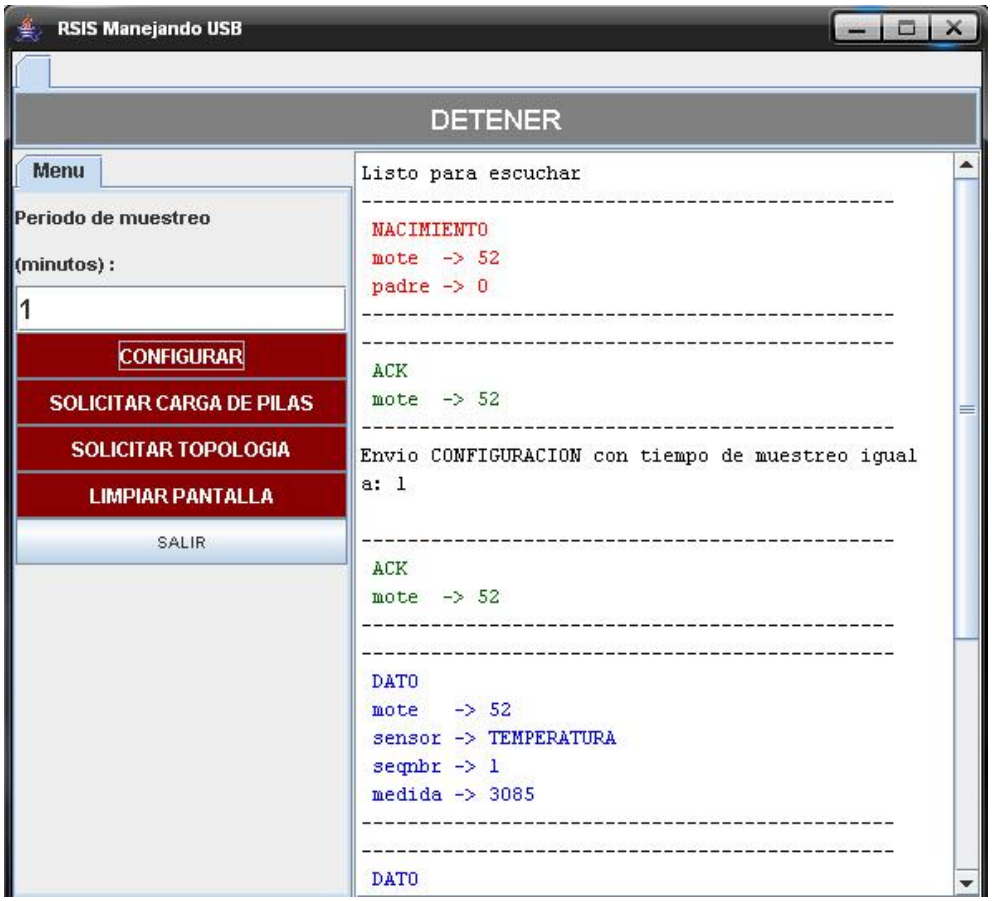


Figura 5.15: Vista de la aplicación de pruebas

Dicha aplicación fue desarrollada en JAVA ya que se contaba con una clase que nos permite controlar el puerto USB. Dicha clase se llama MoteIF[3].

[3] http://www.tinyos.net/dist-2.0.0/tinyos-2.0.0/doc/javadoc/net/tinyos/message/MoteIF.html

En la figura anterior se puede apreciar un área de texto ubicada a la derecha, la cual es utilizada para desplegar los mensajes recibidos.

El menú está conformado por 1 text box y 5 botones, a lo que se agrega el botón de la parte superior que permite detener o reanudar la adquisición de datos cuando el usuario lo desee sin la necesidad de cerrar el programa y perder los datos ya recogidos.

El primero de los 5 botones se llama"*CONFIGURAR*" y su función es enviar un mensaje de *CAMBIO_CONFIGURACIÓN* al mote Sumidero con el tiempo de muestreo indicado en el text-box, el cual es ingresado por el usuario y representa la cantidad en minutos que debe existir entre dos muestras consecutivas de un mismo mote.

No fue necesario crear un campo en donde el usuario pudiera elegir el sensor, ya que en los motes el único sensor que tenían en común es el de temperatura interna del microprocesador, por lo tanto siempre se realizará este sensor.

Cada mote responderá con un mensaje de reconocimiento.

El segundo de los botones llamado "*SOLICITAR CARGA DE PILA*" crea un mensaje en donde solicita un sensado del nivel de voltaje actual de las pilas.

Cada mote responderá con un mensaje del mismo tipo pero con el valor de la medida realizada.

El tercero de los botones llamado "*SOLICITAR TOPOLOGÍA*" envía una solicitud del estado actual de la red. A esto, el Sumidero responde con un mensaje por cada mote que hay en la red, indicando su *MOTE_ID* y el de su padre.

Los restantes 2 botones son intrascendentes a interés de las pruebas a realizar ya que uno limpia el text-área de los mensajes recibidos y el otro botón finaliza la ejecución del programa.

Capítulo 6

Diseño entidad de red

En este capítulo se tratarán todos los temas referidos al proceso de diseño de la red. Se comienza explicando la primera etapa de desarrollo que consistió en el estudio de los casos de uso. Luego se explica el proceso de selección de la topología de red, y por último el desarrollo de la aplicación, explicando la estructura de datos y la implementación en sí.

6.1. Estudio de casos de uso y funcionalidades principales

El primer paso en el diseño de la capa de red fue el estudio de los casos de uso y las primitivas que esta capa iba a implementar para brindar servicio a su capa superior: la capa de aplicación.

El estudio realizado de los casos de uso llevó a identificar tres grandes situaciones que debía contemplar la red. Las mismas son el envío de datos, el ruteo de paquetes y el nacimiento o muerte de un hijo. Dentro del envío de datos, la red se debe encargar de poder enviar los paquetes provenientes de la aplicación hacia el mote destino o recibir los que vengan hacia la aplicación y entregarlos en forma correcta. Por otro lado, debe realizar el ruteo de paquetes, el cual es el alma de la capa de red, debe poder rutear los paquetes en forma Broadcast o hacia un nodo específico de la red. Por último, cada vez que un nuevo nodo ingresa a la red o cuando un nodo muere, este evento debe ser indicado a la capa de aplicación del nodo sumidero.

Ampliando el estudio de los casos de uso, se logró un modelo más refinado, en donde se individualizaron ciertos eventos. Estas nuevas funciones que debe cumplir la capa de red, son las siguientes: inicializar la red, asignar la nueva dirección de red y MAC de un nuevo hijo asociado, realizar el ruteo de paquetes, el procesado de paquetes y por último notificar la muerte de un hijo.

A continuación se describe la funcionalidad de cada uno de lo casos descriptos anteriormente. La función de inicializar la red, se encarga de la inicialización de toda la red, esto comprende la inicialización de todos los componentes necesarios para el correcto funcionamiento a nivel de capa de red, así como también la orden de inicialización de la capa inferior. Por otro lado, la red de un nodo cualquiera, debe al momento de obtener un nuevo hijo, asignarle una nueva dirección tanto de red como de MAC, de acuerdo al direccionamiento empleado (El direccionamiento será explicado en la sección 6.2), por tal

motivo este punto forma también parte de la inicialización.

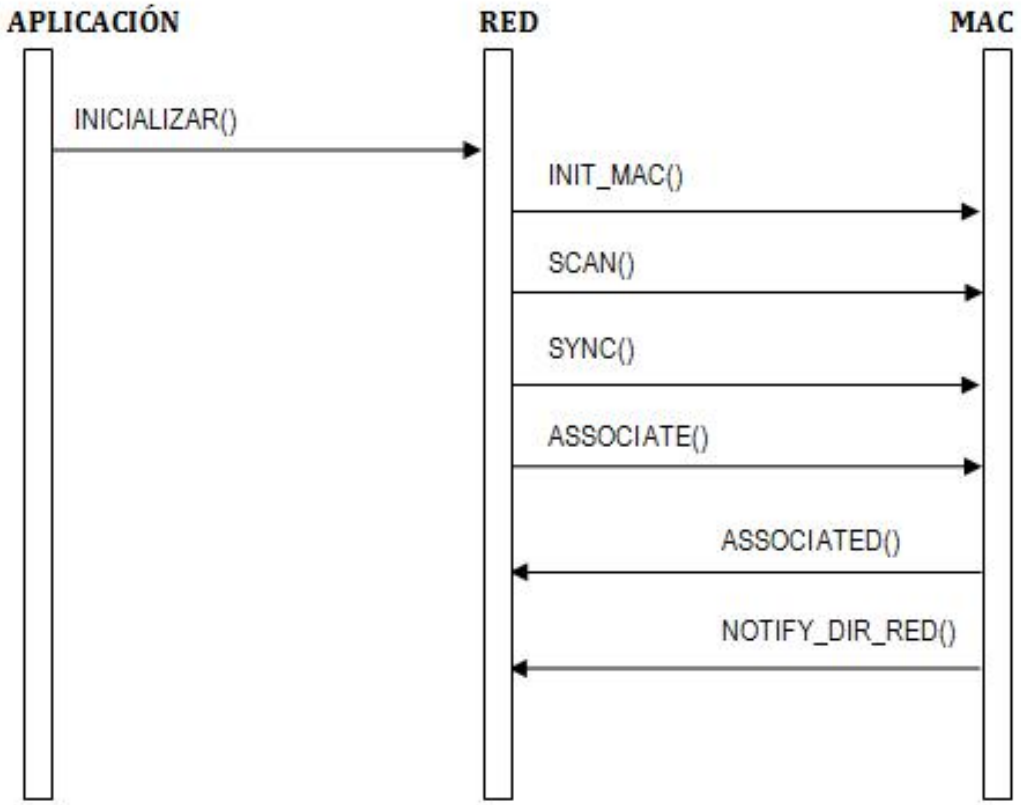


Figura 6.1: Diagrama de secuencia para el caso de uso de Inicialización de la Red, en el caso de un nodo No-Sumidero.

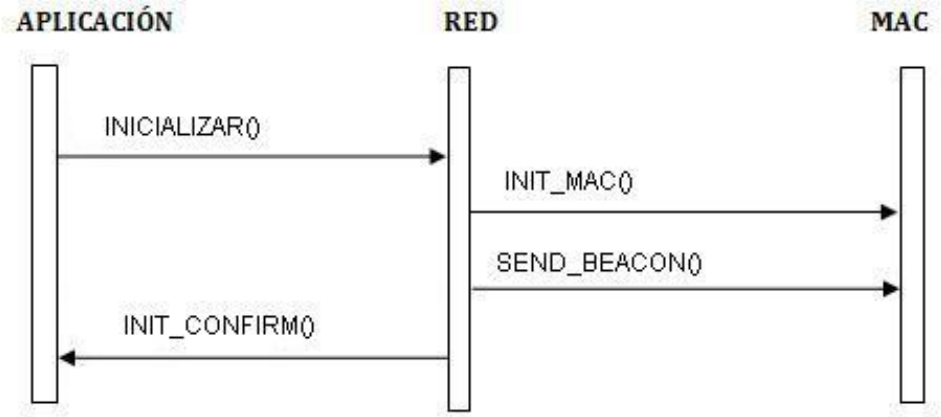


Figura 6.2: Diagrama de secuencia para el caso de uso de Inicialización de la Red, en el caso del nodo Sumidero.

El ruteo de paquetes, implica el poder decidir a nivel de capa de red, si al recibir un nuevo paquete, el mismo es para el nodo que recibe, es para uno de sus hijos en particular, o es broadcast. En este último caso, será enviado a todos los hijos. Cuando uno de los paquetes recibidos es para un nodo en particular, el mismo deberá procesar el paquete a nivel de capa de red y proceder dependiendo del tipo de paquete que sea.

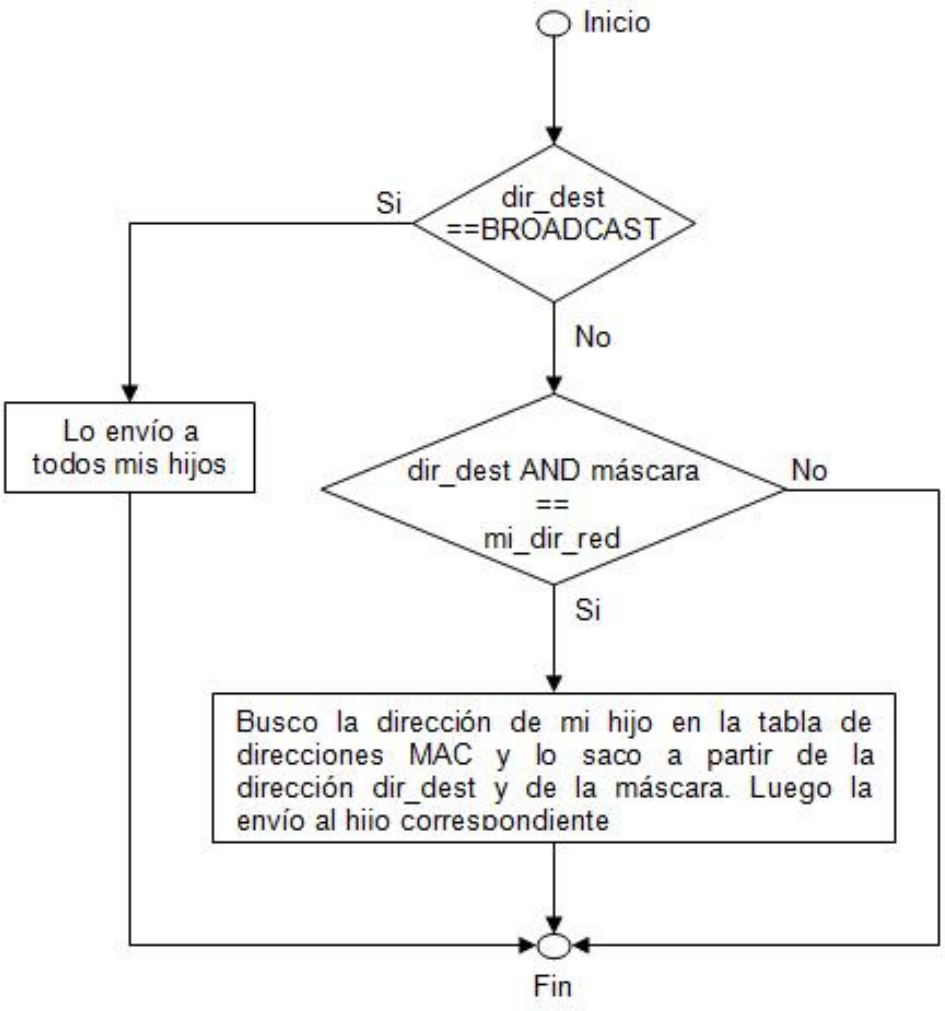


Figura 6.3: Diagrama de flujo para el caso de uso del Ruteo de paquetes.

Si a un nodo se le asocia un nuevo hijo, éste deberá avisarle al nodo sumidero el evento mediante un mensaje de nacimiento. Esto se debe a que el sumidero llevará el control de toda la topología de la red, sabiendo qué nodos están vivos, qué dirección de red tienen y qué número de mote. Por último se deberá indicar al sumidero el caso contrario, o sea la muerte de un hijo, para que éste pueda reordenar su tabla de topología.

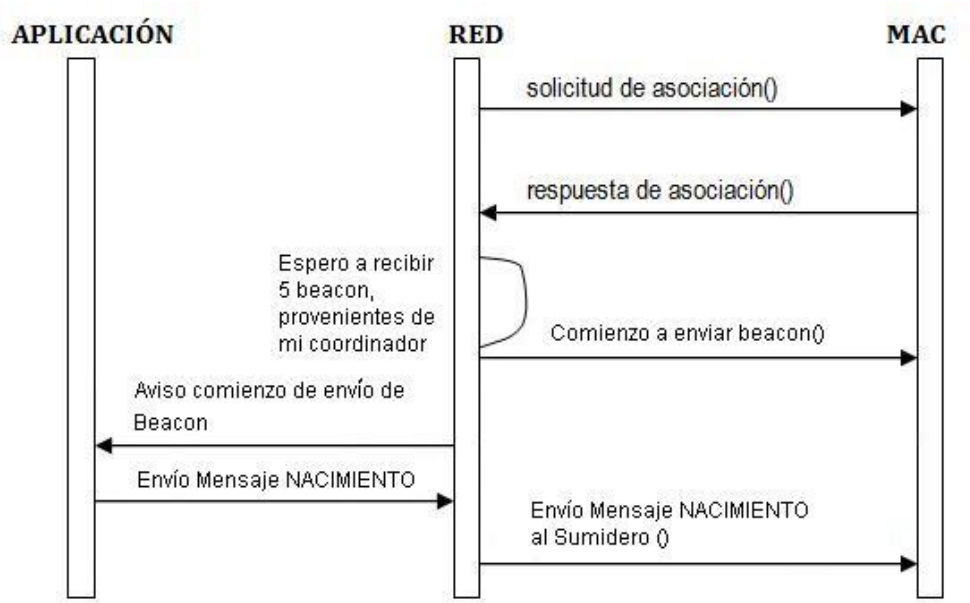


Figura 6.4: Diagrama de secuencia para el caso de uso del Nacimiento de un nuevo nodo.

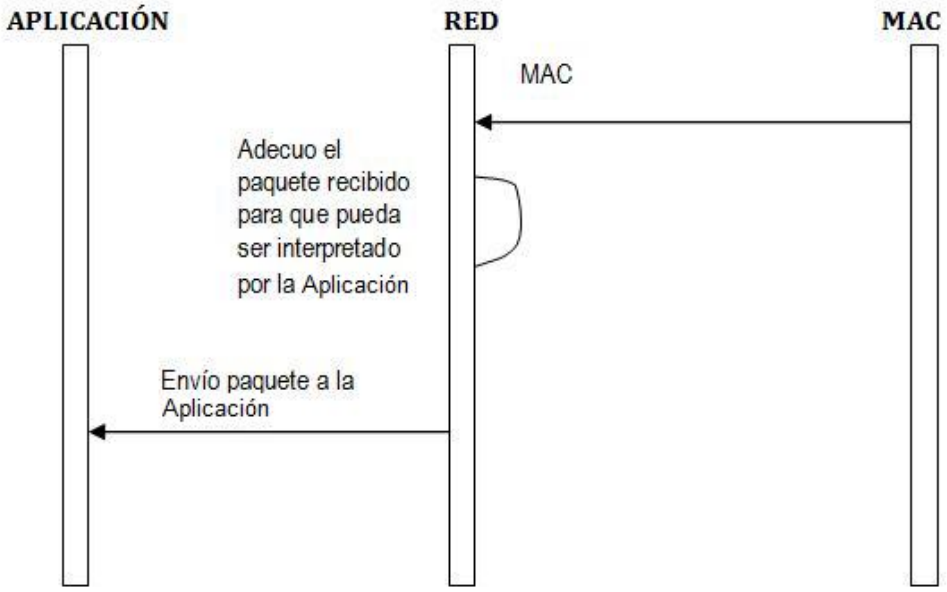


Figura 6.5: Diagrama de secuencia para el caso de uso de la Muerte de un nodo en la red para el caso de un Sumidero.

Por último queda el caso de uso del procesado de paquetes. La red debe procesar los datos recibidos y saber qué hacer con ellos, la figura 6.7 muestra cómo será el procesado de paquetes.

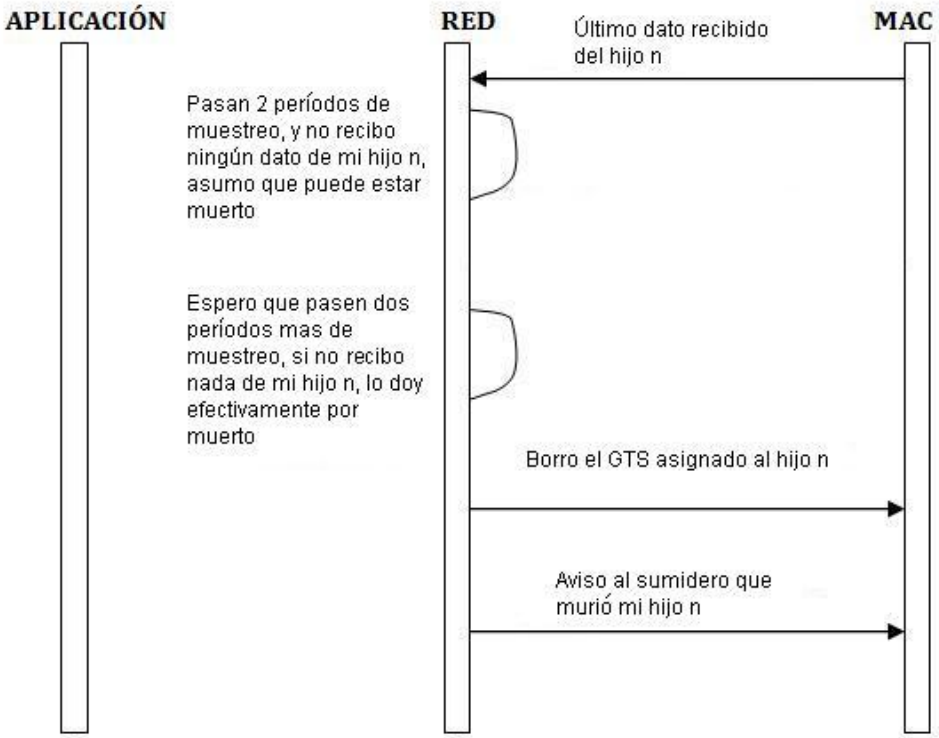


Figura 6.6: Diagrama de secuencia para el caso de uso de la Muerte de un nodo en la red para el caso de un No Sumidero.

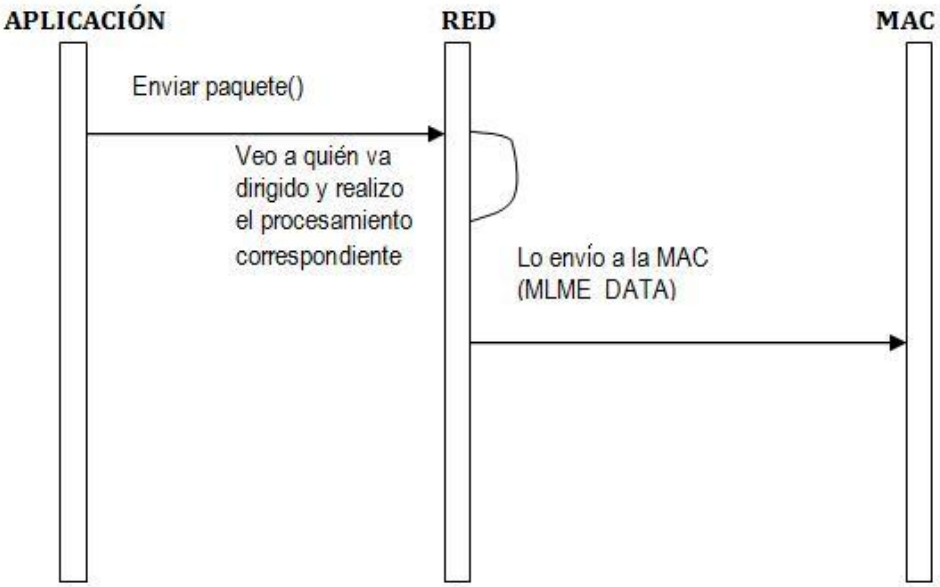


Figura 6.7: Diagrama de secuencia para el caso de uso del procesado de un paquete enviado desde la aplicación.

Debido a que la red debe proporcionar funcionalidades diferentes en el caso de que sea

sumidero o no, se realizaron dos instancias diferentes de la red, una llamada RedSumideroM.nc y otra RedM.nc. La necesidad de estas dos instancias radica en que el sumidero funcionará siempre como padre, enviando beacon, por lo tanto, no tendrá que escuchar los diferentes canales en busca de beacon, solicitar una asociación y pedir dirección de MAC y red. En cambio los dispositivos no sumideros si deberán realizar todas estas tareas.

La red implementará las primitivas que proporciona y a su vez utilizará las primitivas que le proporcionan las capas MAC como coordinador y como end_device. En el punto 7.4 se explica cómo se implementa cada una de las funcionalidades que provee la red.

6.2. Elección de la topología de la red

A la hora de elegir la topología de red, se estudiaron las posibles configuraciones que se podrían implementar. Estas configuraciones son:

- Clúster tree
- Mesh
- Star

Para evaluar cada una de ellas, con el fin de definir la más adecuada, hay que analizar las necesidades y la particularidad de nuestra red.

El mayor flujo de mensajes se originan en los motes de la red y tienen como destino el nodo central conectado al router. Los casos en que los mensajes viajan en sentido contrario son muy escasos, y se trata solamente ante una orden del usuario. Como sabemos nuestra red estará funcionando continuamente y la intervención del usuario se efectúa en muy pocas oportunidades ya que el principal objetivo es que los motes recaben datos y los envíen a los motes centrales.

De las observaciones anteriores se concluye que hay que privilegiar el flujo de mensajes en un sentido (up-link).

Otro punto a tener en cuenta es que en ningún caso hay necesidad de intercambiar mensajes entre nodos distantes en la red que no involucren al nodo central.

Por parte de la implementación, la topología ideal es que cada mote, sin considerar el mote central sería mantener un enlace por cada mote; de esta manera simplificaría mucho la programación, reduciendo el tamaño del programa y de esta manera ahorrar recursos de memoria que son muy escasos en nuestro hardware.

Considerando que los motes tienen un alcance entre 50-70 metros, será necesario implementar el multi-hop para poder ofrecer una cobertura mayor, ya que un círculo de 70 metros es muy reducido para la aplicación. Esta condición obliga a que cada mote tenga que mantener como mínimo dos enlaces y así descartar la topología star (estrella).

Por otro lado, el standard IEEE 802.15.4 define que cada mote, en un enlace es "hijo" ó es "padre" lo que significa que es end_device ó coordinador respectivamente. Cada mote es definido como hijo ó como padre en cada interfaz, pero no puede tener dos interfaces en las que sea hijo.

A partir de esta restricción y la particularidad que no hay necesidad de comunicación entre dos end_device's se concluye entonces que la topología más adecuada es cluster-tree.

Luego de definida la topología, se comienza estudiar la red: en particular el direccionamiento y el ruteo.

Primero se planteó si es necesaria la modalidad de envío de mensajes *UNICAST*, ya que en principio los mensajes desde un end_device siempre van al nodo central y los mensajes desde el nodo central tienen como destino toda la red, o sea *BROADCAST*.

Finalmente se descubrió que en algunos casos, por ejemplo ante la falta de un mensaje de reconocimiento por parte del end_device, habría que volver a intentar enviarle un mensaje y en ese caso no es conveniente volver a enviar un *BROADCAST* sino un *UNICAST* a ese destino en particular.

Lo anterior obligó a pensar en una capa de red donde existiera un adecuado direccionamiento para poder realizar el ruteo.

El direccionamiento concluido es el que mostrado en la imagen siguiente:

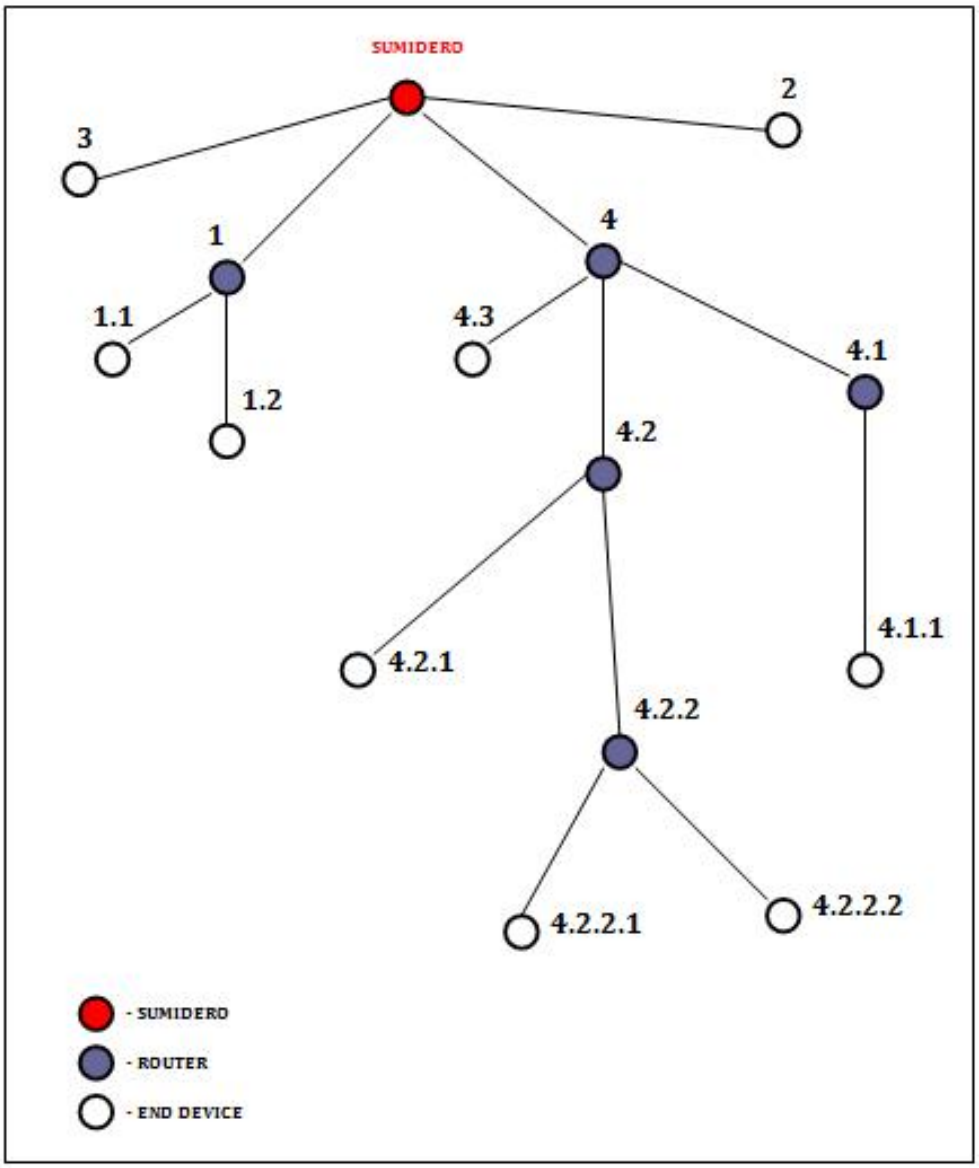


Figura 6.8: Esquema de topología utilizada a nivel de capa de red.

Es necesario que la capa de red maneje una variable que indique la distancia (en saltos) al sumidero.

D: Distancia, en saltos, del nodo al sumidero.

La dirección de red va a estar formada por un "array" en donde cada entrada está definida por el número de hijo que tienen sus padres. Este número como máximo puede valer 7, ya que este es el límite de hijos que puede tener un padre en nuestra implementación. El "0" está prohibido como número de hijo.

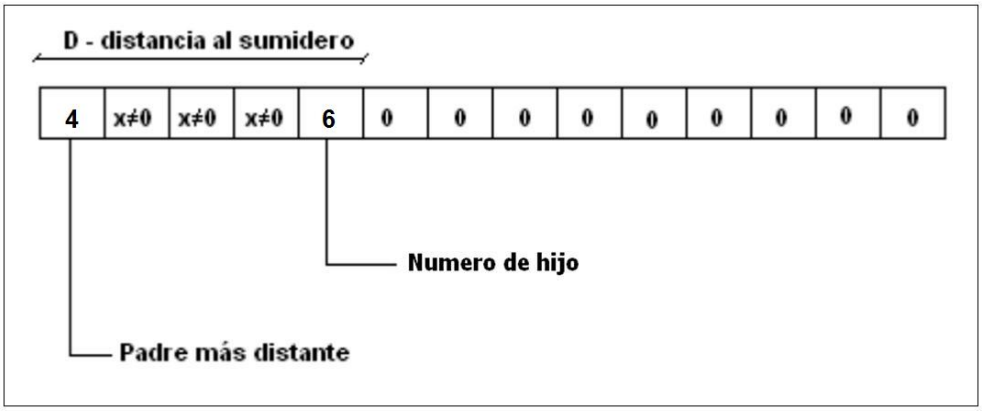


Figura 6.9: Formato de la dirección de red.

También se utiliza una máscara:

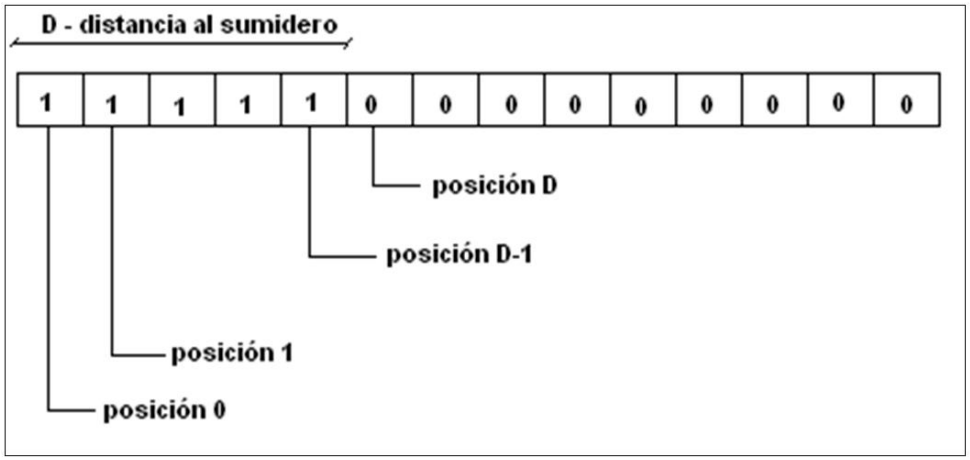


Figura 6.10: Máscara asociada al ejemplo anterior.

Ya definido el direccionamiento se pasó a ver el tema del ruteo, el cual será necesario únicamente en los casos en que el origen sea el nodo central (Sumidero) y sea en modalidad *UNICAST*.

ALGORITMO PARA RUTEAR PAQUETES

DirDest - Dirección de red del destino.
MiDir - Mi dirección de red.
M - Máscara

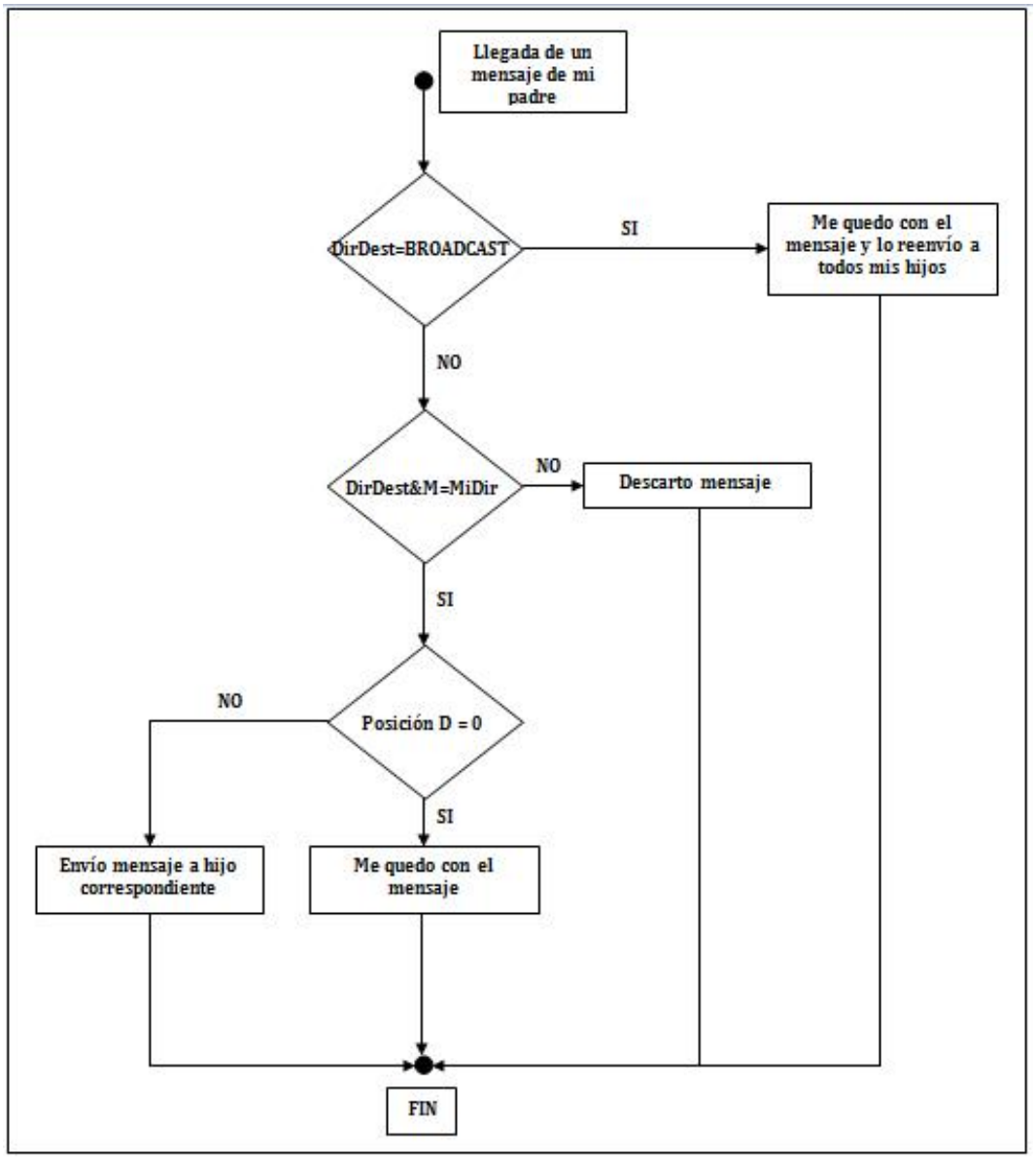


Figura 6.11: Diagrama de flujo de ruteo de paquetes.

La asignación de direcciones es dinámica y se realiza al momento de asociación de un nuevo hijo.

Si se tienen 7 hijos asociados, se debe rechazar el nuevo pedido de asociación, en caso contrario se debe asignar el número de hijo disponible; para ello la capa de red deberá llevar una tabla de asociación:

Número de hijo	Dirección de MAC
1	Dir_MAC 1
2	Dir_MAC 2
–	–
4	Dir_MAC 4
–	–
–	–
7	Dir_MAC 7

Tabla 6.1: Tabla de asociación que mantiene la capa de red.

6.3. Estructura de datos

Para estructurar los mensajes a nivel de capa de red, se comenzó por dimensionar la carga que se debe transportar. Para ello hay que analizar el tamaño de los mensajes de la aplicación.

Los mensajes de aplicación como máximo pueden tener largo 6 bytes.

Otro punto a tener en cuenta es que la interfaz ya existente con la capa MAC, permite como máximo pasar una carga de largo 120 bytes.

Con este par de requerimientos, se comenzó a trabajar en las diferentes estructuras posibles para los paquetes de red y es aquí necesario analizar los posibles tipos de paquetes existentes.

En la capa de red se manejan tres diferentes tipos de paquetes:

- MUERTE
- DATO
- GESTIÓN

Los paquetes de muerte tienen como fin notificar al Sumidero de la muerte de algún mote. Los paquetes del tipo dato son exclusivamente para transportar mensajes de la capa de aplicación; en cambio los de gestión, no alcanzan la capa superior sino que se mantiene a nivel de red para intercambiar información relacionada exclusivamente con la gestión de

la red.

El caso en que el paquete adquiere el mayor tamaño es obviamente en los datos. Además de la carga, un dato debe tener la dirección de destino que ocupa 4 bytes. Estos dos campos comprenderían un dato: dirección de red y carga.

Para la capa de red, también es útil conocer el origen del dato, por ello se le agrega un campo que es la dirección de red de origen, a lo que se le suma el campo tipo de mensaje y formamos así la unidad de red denominada paquete.

Luego de tener un paquete, se debe solicitar el servicio de la capa MAC para que ésta se encargue de transferirla. Lo ideal en este paso es aprovechar al máximo la capacidad de la trama MAC y enviar varios paquetes a la vez, pero esto conlleva un problema difícil de solucionar con los recursos limitados de memoria y procesamiento. Para juntar varios paquetes se debe tomar la decisión de en qué momento, aunque no llenemos la trama, solicitar el envío. Si se espera hasta completar la trama en todos los casos , se estaría agregando retardo a los paquetes y no favorece los intereses del usuario.

En la implementación actual, en cada trama se envía un solo paquete pero igualmente se incluye un campo que indica la cantidad de paquetes que viajan en la trama, el cual obviamente está fijo en 1.

La figura siguiente ilustra la estructura para el caso de un paquete del tipo dato:

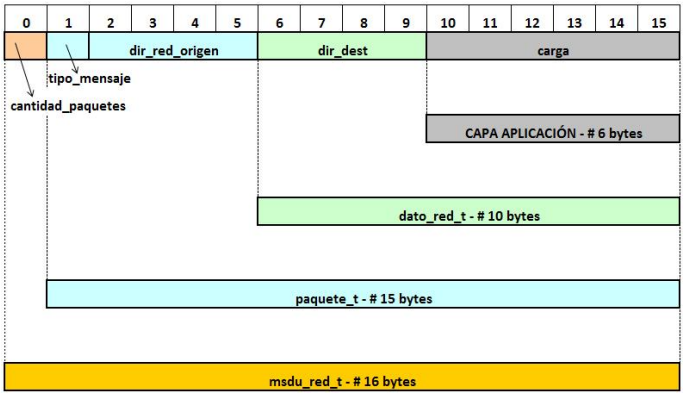


Figura 6.12: Estructura de un mensaje de tipo *DATO*.

Cabe notar que cuando se trata de un paquete de muerte, en el campo dir_dest se guarda la dirección de red del mote que acaba de des asociarse de la red.

Para los mensajes del tipo gestión, existe una estructura diferente y se ilustra en la siguiente figura:

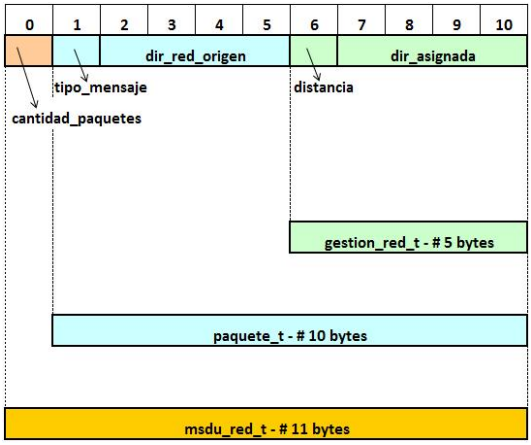


Figura 6.13: Estructura de un mensaje de tipo *GESTION*.

Para los tres tipos de mensajes, se utiliza la siguiente tabla de identificadores:

tipo_mensaje	identificador
GESTION	0x01
MUERTE	0x04
DATO	0x05

Tabla 6.2: Tabla de identificadores de mensajes de red.

6.4. Implementación: Funciones, eventos y comandos

En esta sección, se detalla el funcionamiento de los módulos RedSumideroM.nc y RedM.nc. Se explicará la función de cada uno de los comandos y eventos que se implementan en los mismos, junto con las variables usadas.

6.4.1. RedSumideroM.nc

Variables Globales:

- `uint16_t my_coord_address`: En esta variable se setea la dirección corta de capa MAC que tendrá el sumidero. Es utilizada para identificar a nivel de capa MAC al sumidero y para asignar la dirección de capa MAC de los hijos del sumidero. Esta última se forma en función de la dirección del sumidero y el número de hijo.
- `uint16_t my_coord_pan_id`: Esta variable indica el identificador de la PAN en la cual se va a comenzar la red.
- `uint32_t my_coord_address_long[2]`: Es usada para enviar los datos desde la red hacia la MAC, se necesita porque a pesar de que se utilizaran las direcciones cortas

es necesario pasar una dirección de 64 bits. Se carga la misma dirección corta en los dos componentes del array.

- `dir_MAC_t dir_MAC_hijos[7]`: Este array es usado como tabla para almacenar la dirección MAC de cada uno de los hijos del sumidero, de forma de poder saber a que hijo enviar un dato o desde que hijo provino. La estructura de datos `dir_MAC_t`, fue creada para la aplicación.

```
typedef struct {
  bool ocupado;
  nx_uint8_t direccion_MAC[2];
  bool vivo;
} dir_MAC_t;
```

Figura 6.14: Tipo de variable `dir_MAC_t`.

El campo ocupado, es utilizado para indicar si el elemento del array está en uso, es decir si ya fue asignado a un nuevo hijo. En direccion_MAC[2], almaceno la dirección MAC del nuevo hijo. Por último, el campo vivo es utilizado para controlar la actividad del hijo, esté se pone en uno cuando se recibe un dato nuevo desde el hijo correspondiente y en cero cuando expira el tiempo de control.

- `uint8_t cantidad_hijos`: Utilizada para indicar la cantidad de hijos que tiene el sumidero. Es necesario saber el número de hijos para no pasarse del máximo permitido, el cual es de siete hijos.

- `uint32_t mi_dir_red`: Almacena la dirección de capa de Red del sumidero, la misma será cero para el sumidero, es así para poder individualizarlo a nivel de red.

- `uint8_t distancia`: Indica la distancia al sumidero en cantidad de saltos, en este caso será cero.

- `uint8_t t_muestreo`: Se almacena el tiempo de muestreo de la red, es decir cada cuanto tiempo cada uno de los nodos toma las medidas, la misma está contemplada en minutos. A pesar de que el sumidero no adquiere medidas esta variable es necesaria para controlar si algún hijo murió y para poder setear el tiempo de muestreo de la red.

- `uint8_t control_minutos`: Variable utilizada para el control de la muerte de los hijos. Lleva el control de la cantidad de minutos que pasaron desde el último control de actividad.

- `msdu_red_t msdu_salida`: Es utilizada para formar la trama de red a ser enviada a la MAC.La estructura `msdu_red_t`, fue implementada para manejar los paquetes de red.

```
typedef struct {
  nx_uint8_t cantidad_paquetes;
  paquete_t paquete;
} msdu_red_t;
```

Figura 6.15: Tipo de variable `msdu_red_t`.

El funcionamiento de cada campo se encuentra explicado en la sección 6.3.

- `uint8_t auxiliar[120]`: Variable auxiliar utilizada para almacenar el msdu de red a ser enviado.
- `bool PrimerHijo`: Es utilizada para indicar el nacimiento del primer hijo asociado al sumidero. Es necesaria para habilitar el timer de control de actividad de los hijos.

Funciones:

- `void init_red()`: Funcion utilizada al momento de arrancar la red. La misma se encarga de habilitar la capa MAC e inicializar sus variables, setear la dirección de MAC y la PAN. Por ultimo comienza a enviar Beacon utilizando la interfaz de capa MAC `MLME_COORD_START.request`.

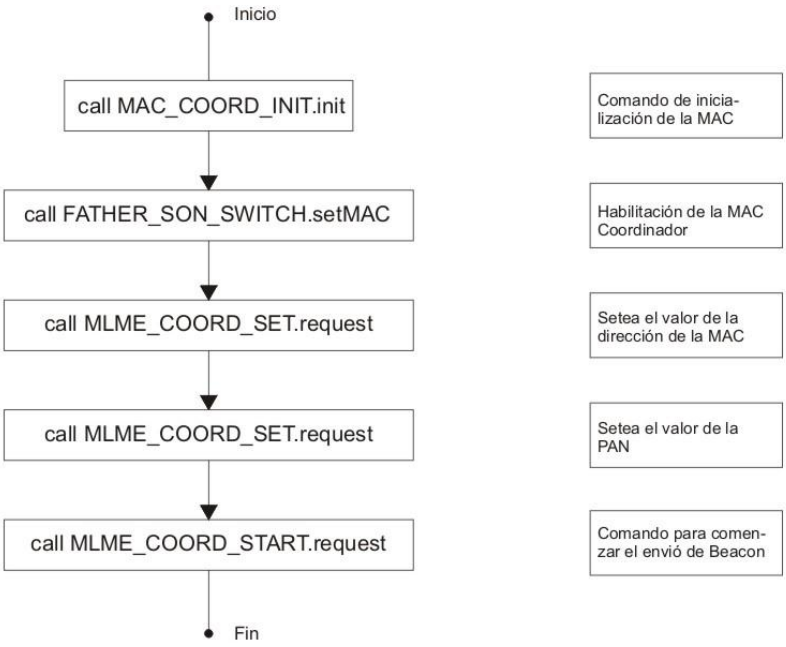


Figura 6.16: Diagrama de flujo de la función `init_red`.

- `void armar_direccion_de_red(uint8_t nro_de_hijo)`: Función a la cual se le pasa como argumento el número de hijo. En función de este parámetro la función se encarga de generar y enviar la dirección de red del nuevo hijo.

Tareas:

- `task void controlar_hijos()`: Esta es una tarea que se encarga de controlar la actividad de los hijos que tiene asociados. Se postea periódicamente con un timer, cada medio tiempo de beacon. En la tarea se incrementa un contador y cuando el mismo llega a dos veces el tiempo de muestreo, controla la actividad de todos los hijos. Al morir un hijo se manda un mensaje de muerte hacia la aplicación disparando el evento `RED_DATA.indicationsum` y también se llama al comando `MLME_COORD_GTS.request` para eliminar el GTS asignado al hijo muerto.

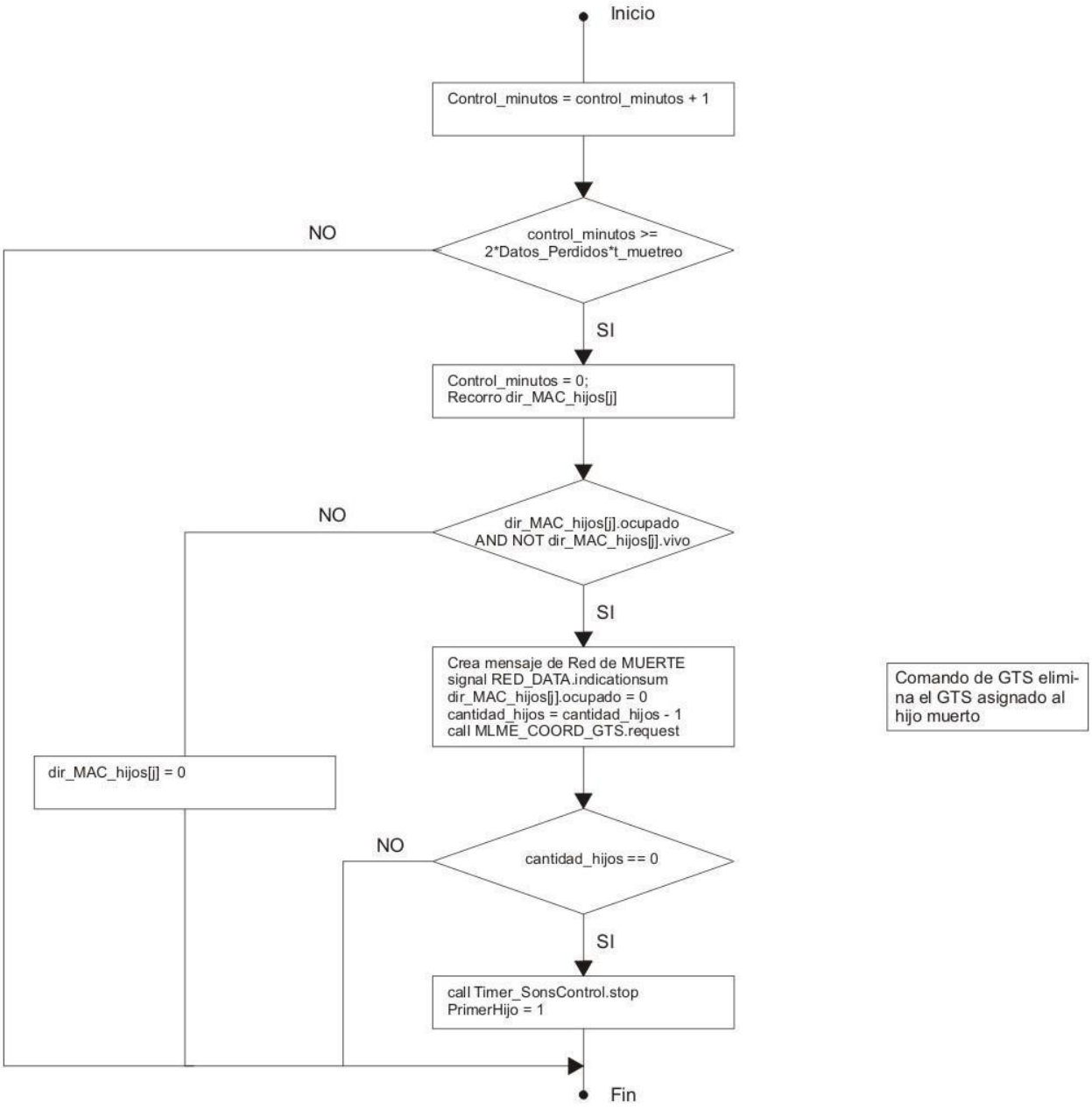


Figura 6.17: Diagrama de flujo de la tarea `controlar_hijos`.

Comandos:

El modulo RedSumideroM, provee tres interfaces, las cuales son `RED_DATA`, `RED_MNGT` y `SplitControl`. Estas tres interfaces poseen comandos y eventos, por tal motivo los comandos son implementados en la capa de red y lo eventos disparados desde la red e implementados en la aplicación.

- `command error_t RED_DATA.request(uint32_t DirDestino, message_t* carga)`: Este comando se encarga de generar el paquete de red, para enviarlo en forma broad-

cast o a un hijo en particular dependiendo del argumento `DirDestino`, la carga proveniente de la aplicación será el msdu del paquete de red. Una vez pronto el paquete se llama al comando `MCPS_COORD_DATA.request` y se envía al destino correspondiente.

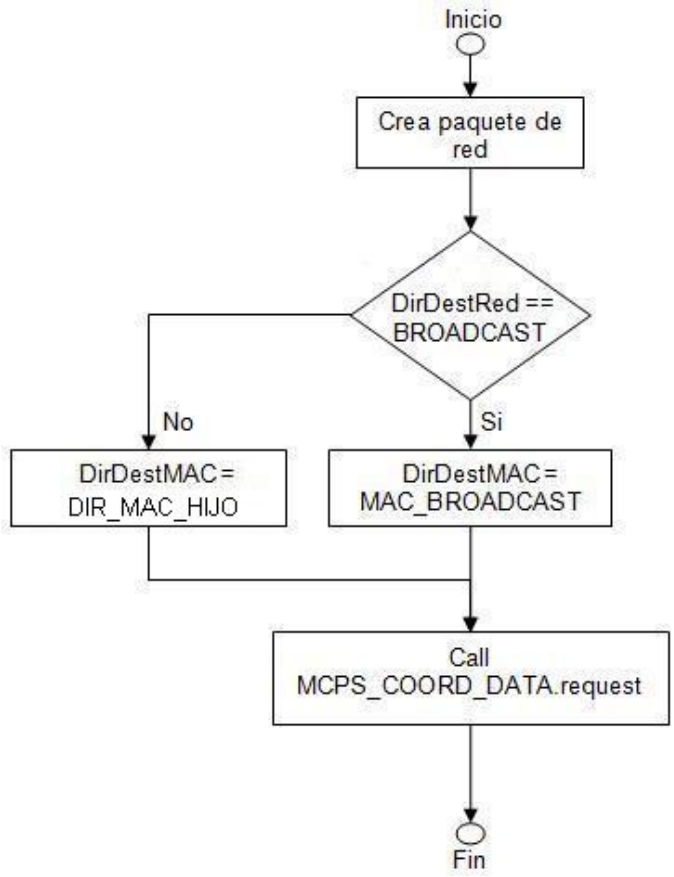


Figura 6.18: Diagrama de flujo del comando `RED_DATA.request`.

- `command error_t RED_MNGT.set_TMUESTREO(uint8_t periodo_muestreo)`: Este comando se encarga de setear el tiempo de muestreo a nivel de capa de red. Es decir carga en la variable global `t_muestreo` el valor del argumento pasado al comando. Esto es necesario para conocer la cadencia con que llegarán los datos provenientes de los hijos y de esta forma poder controlar la actividad de los mismos.
- `command error_t SplitControl.start()`: Este comando se encarga de llamar a la función init_red().
- `command error_t SplitControl.stop()`: Comando implementado para poder utilizar la interfaz `SplitControl`, no realiza ninguna función.

Eventos:

- `event error_t MLME_COORD_GTS.indication(uint16_t DevAddress, uint8_t GTSCharacteristics, bool SecurityUse, uint8_t ACLEntry)`: Al dispararse este

evento la red llama a la función `armar_direccion_de_red`, debido a que cuando se dispara este evento el hijo ya se encuentra asociado y listo para recibir su dirección de red.

- `event error_t MLME_COORD_START.confirm(uint8_t status)`: Cuando llega este evento se dispara el evento `SplitControl.startDone`, con el argumento `SUCCESS`. Se dispara este evento hacia la aplicación para confirmar a la misma que terminó la inicialización.
- `event error_t MLME_COORD_SET.confirm(uint8_t status, uint8_t PIBAttribute)`: No realiza ninguna acción, se necesita para poder utilizar la interfaz `MLME_COORD_SET`.
- `event error_t MLME_COORD_GET.confirm(uint8_t status, uint8_t PIBAttribute, uint8_t PIBAttributeValue[])`: No realiza ninguna acción, se necesita para poder utilizar la interfaz `MLME_COORD_GET`.
- `event error_t MLME_COORD_ASSOCIATE.indication(uint32_t DeviceAddress[], uint8_t CapabilityInformation, bool SecurityUse, uint8_t ACLEntry)`: Este evento se dispara cuando llega un pedido de asociación por parte de un nuevo hijo. Antes de autorizar la asociación, se verifica que no se halla pasado el número máximo de hijos. Se llama al comando `MLME_COORD_ASSOCIATE.response`, para indicar si es posible o no la asociación. En caso de que sea el primer hijo se activa el timer periódico `Timer_SonsControl`, para realizar el control de hijos, el mismo se dispara cada 30 segundos. Se utiliza 30s debido a que 60s sobrepasa el tamaño válido máximo admitido como argumento por el timer(el timer recibe como parámetro el tiempo en milisegundos).

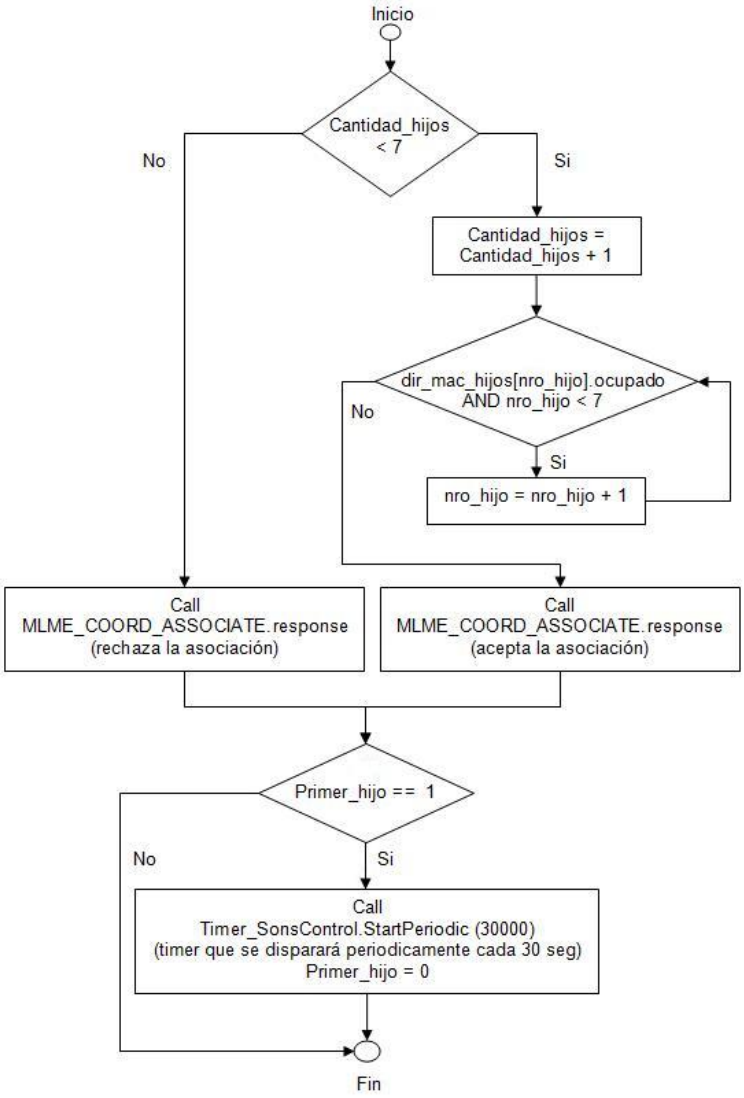


Figura 6.19: Diagrama de flujo del evento `MLME_COORD_ASSOCIATE.indication`.

- `event error_t MLME_COORD_DISASSOCIATE.indication(uint32_t DeviceAddress[], uint8_t DisassociateReason, bool SecurityUse, uint8_t ACLEntry)`: No realiza ninguna acción, se necesita para poder utilizar la interfase `MLME_COORD_DISASSOCIATE`.

- `event error_t MCPS_COORD_DATA.confirm(uint8_t msduHandle, uint8_t status)`: No realiza ninguna acción, se necesita para poder utilizar la interfase `MCPS_COORD_DATA`.

- `event error_t MCPS_COORD_DATA.indication(uint16_t SrcAddrMode, uint16_t SrcPANId, uint32_t SrcAddr[2], uint16_t DstAddrMode, uint16_t DestPANId, uint32_t DstAddr[2], uint16_t msduLength, uint8_t msdu[120], uint16_t mpduLinkQuality, uint16_t SecurityUse, uint16_t ACLEntry)`: Este evento se dispara cuando llega un nuevo dato a la red proveniente de la MAC. Se encarga de enviar el dato hacia la aplicación, esto lo lleva a cabo disparando el evento `RED_DATA.indicationsum` y extrayendo la carga útil del paquete de red.

- `event void Timer_SonsControl.fired()`: Cada vez que se dispara este evento se postea la tarea `controlar_hijos`, para verificar el estado de cada uno de sus hijos.

6.4.2. RedM.nc

Variables Globales:

El siguiente grupo de variables cumplen la misma función que en RedSumideroM.nc, por tal motivo serán únicamente mencionadas, el resto de las variables serán explicadas.

- `uint32_t my_coord_address_long[2]`
- `dir_MAC_t dir_MAC_hijos[7]`
- `uint8_t cantidad_hijos`
- `uint8_t distancia`
- `uint8_t t_muestreo`
- `uint8_t control_minutos`
- `msdu_red_t msdu_salida`
- `uint8_t auxiliar[120]`

El siguiente grupo de variables solo se encuentran en RedM.nc, por tal motivo será explicada su función.

- `PANDescriptor pan_des`: En esta variable se guardan los valores elegidos para canal, dirección del padre, etc. una vez terminado el escaneo de los diferentes canales en busca de uno para asociarse.
- `uint16_t my_device_address`: Es el valor de la dirección de capa MAC del nodo como hijo. Inicialmente tiene el valor de 0xFFF0. Cuando se asocia con un padre, se cambia esta variable al valor asignado por el mismo.
- `uint16_t father_coord_address`: Utilizada para almacenar la dirección MAC del padre con el cual se asocia.
- `uint16_t father_pan_id`: Se le carga el valor de identifiacor de la PAN del padre con el cual se asocia.
- `my_coord_address`: Esta variable es utilizada para cargarle el valor de la dirección MAC, que va a tener el dispositivo cuando se padre y mande beacon. Será igual al valor de `my_device_address`.
- `uint16_t my_coord_pan_id`: Se inicializa con el valor de PAN en la cual va a mandar beacon.

- `uint8_t received_beacon_count`: Variable que lleva la cuenta de la cantidad de beacon recibidos.
- `bool allow_channel_change`: Este booleano permite el cambio de canal, entre los canales de recepción de beacon y transmisión de beacon.
- `uint8_t go_associate`: Variable que habilita el pedido de asociación con un padre una vez sincronizado con el mismo.
- `uint32_t mi_dir_red`: Almacena la dirección de capa de Red asignada por el padre una vez asociado con el mismo.
- `uint32_t mascara`: Variable en la cual se almacena la máscara para el ruteo, la misma se forma a partir de la distancia al sumidero.
- `uint8_t canales_ocupados[15]`: Este array almacena el canal al cual se está intentando asociar.
- `uint8_t canal_nueva_PAN`: Se le carga el canal en el cual se comenzara la nueva PAN, es decir en el cual se comunicara el dispositivo como padre.
- `uint32_t DeviceAddress1[2]`: Variable utilizada para almacenar la dirección MAC del nodo que pidió la asociación.
- `uint8_t SyncState`: Variable que indica el estado del sincronismo. Su valor dependerá de si está o no sincronizado.
- `uint8_t nro_de_hijo_Armar_red`: Almacena el número de hijo que pidió la asignación de los GTS.

Funciones:

- `void init_red()`: Esta función al igual que en `RedSumideroM` se encarga de inicializar la capa MAC. En lugar de comenzar a mandar beacon una vez inicializada la MAC, procede a realizar un escaneo de todos los canales en busca de actividad llamando al comando `MLME_DEVICE_SCAN.request`.
- `void armar_mascara(uint8_t distancia_al_sumidero)`: En función del argumento `distancia_al_sumidero`, calcula la máscara correspondiente para poder realizar el ruteo de paquetes de red. La máscara es calculada a partir de la distancia al sumidero.
- `bool verificar_canal_libre(uint8_t canal_a_verificar)`: Esta función devuelve `TRUE` si el canal que se le pasa como argumento se encuentra en el array global `canales_ocupados`, en caso contrario devuelve `FALSE`.
- `void ocupar_canal(uint8_t canal_a_ocupar)`: Función que se encarga de guardar en el array `canales_ocupados` el canal que se le pasa como argumento a la función.
- `void liberar_canales_ocupados()`: Elimina todos los canales guardados en el arrray `canales_ocupados` y lo pone en 0.

Tareas:

- `task void armar_direccion_de_red()`: Esta tarea se encarga de generar la dirección de red del nuevo hijo asociado. Una vez generada es enviada al hijo mediante el comando `MCPS_COORD_DATA.request`. Su funcionalidad es similar a la de `RedSumideroM`.
- `task void start_sending_beacons()`: Esta tarea se postea una vez asociado el dispositivo con un coordinador y se encuentra listo para comenzar a mandar beacon. Comienza a mandar beacon llamando al comando de MAC, `MLME_COORD_START.request`.
- `task void controlar_hijos()`: Cumple la misma función que la tarea descripta en `RedSumideroM`, la única diferencia es el momento en que se postea la tarea, la misma se postea cada vez que recibe un beacon.
- `task void associate()`: Se encarga de procesar un pedido de asociación por parte de un nuevo hijo. Su funcionamiento es igual al del comando `MLME_COORD_ASSOCIATE.indication` del modulo `RedSumideroM`.

Comandos:

- `command error_t RED_DATA.request(uint32_t DirDestino, message_t* carga)`: Este comando se cumple la misma función que en `RedSumideroM`. La única diferencia es que en siempre envía el paquete hacia el sumidero y utiliza el comando de MAC `MCPS_DEVICE_DATA.request`.

Los siguientes comandos realizan la misma función que su simétricos en el modulo `RedSumideroM`, por tal motivo solo serán mencionados.

- `command error_t RED_MNGT.set_TMUESTREO(uint8_t periodo_muestreo)`.
- `command error_t SplitControl.start()`.
- `command error_t SplitControl.stop()`.

Eventos:

- `event error_t MLME_DEVICE_SCAN.confirm(uint8_t status, uint8_t ScanType, uint16_t UnscannedChannels, uint8_t ResultListSize, uint8_t EnergyDetectList[], SCAN_PANDescriptor PANDescriptorList[])`: Este evento es disparado una vez terminado el escaneo de canales. Cuando termina de escanear todos los canales por energía, genera una máscara indicando en los canales en los cuales encontro actividad, y en el último que no encontró actividad lo almacena en `canal_nueva_PAN`. Terminado esto realiza un escaneo en dichos canales en busca de beacon, una vez terminado se dispara nuevamente el evento y se selecciona el canal con el cual se sincronizará.

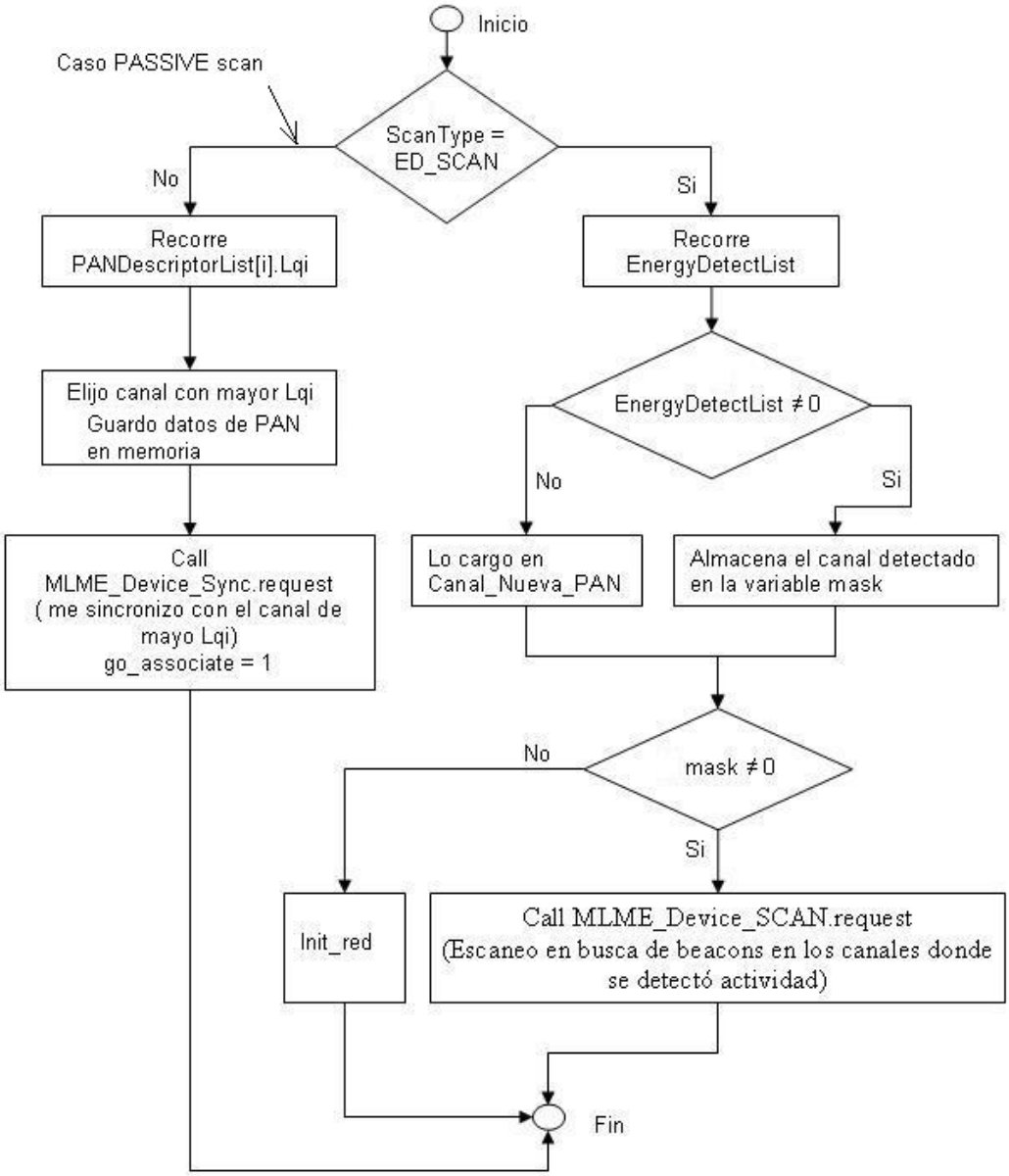


Figura 6.20: Diagrama de flujo del evento `MLME_DEVICE_SCAN.confirm`.

- `event error_t MLME_DEVICE_SYNC_LOSS.indication(uint8_t LossReason)`: Este evento se dispara cuando se pierde sincronismo, el mismo analiza si es una pérdida de sincronismo simple o si se produjo la muerte del coordinador. En el primer caso se trata de sincronizar nuevamente en el mismo canal, en el segundo caso realiza un escaneo nuevamente. Este evento es fundamental para el correcto funcionamiento de la red, ya que es necesario que cada nodo de la red se pueda resincronizar en forma automática en caso de pérdida de sincronismo. En la figura 6.21 se puede observar como es el funcionamiento del evento.

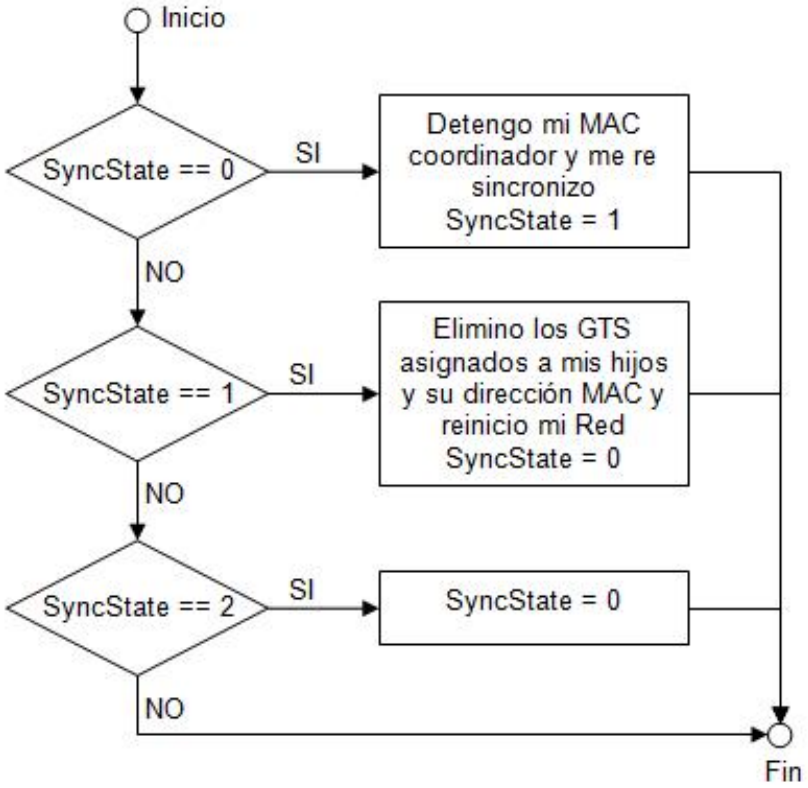


Figura 6.21: Diagrama de flujo del evento `MLME_DEVICE_SYNC_LOSS.indication`.

Por otro lado para poder completar la resicronización es necesario contemplar algunos elementos más. Juntamente con el disparo por primera vez del evento `MLME_DEVICE_SYNC_LOSS.indication` desde la capa MAC, se inicia un timer (T_ScanDuration, el cual es usado para controlar el tiempo que se escanea cada canal, y es usado solo en ese momento; por tal motivo como forma de ahorrar espacio en memoria se reutiliza este timer) para controlar la muerte del coordinador. Como se puede observar en la figura 6.22, una vez que se está en el estado 1, debido a que se perdió sincronismo, se trata de resincronizar con el coordinador que tenía. Si se resincroniza, esto se debe a que recibe nuevamente un beacon del mismo coordinador, se pasa al estado 2 y espera a que expire el timer lanzado para volver al estado 0 de sincronizado. En caso contrario, si no logra resincronizarse y expirara el timer, como todavía estará en el estado 1, se pasa al estado 3. En el estado 3 el nodo se reinicia y comienza una nueva búsqueda de un coordinador con el cual asociarse.

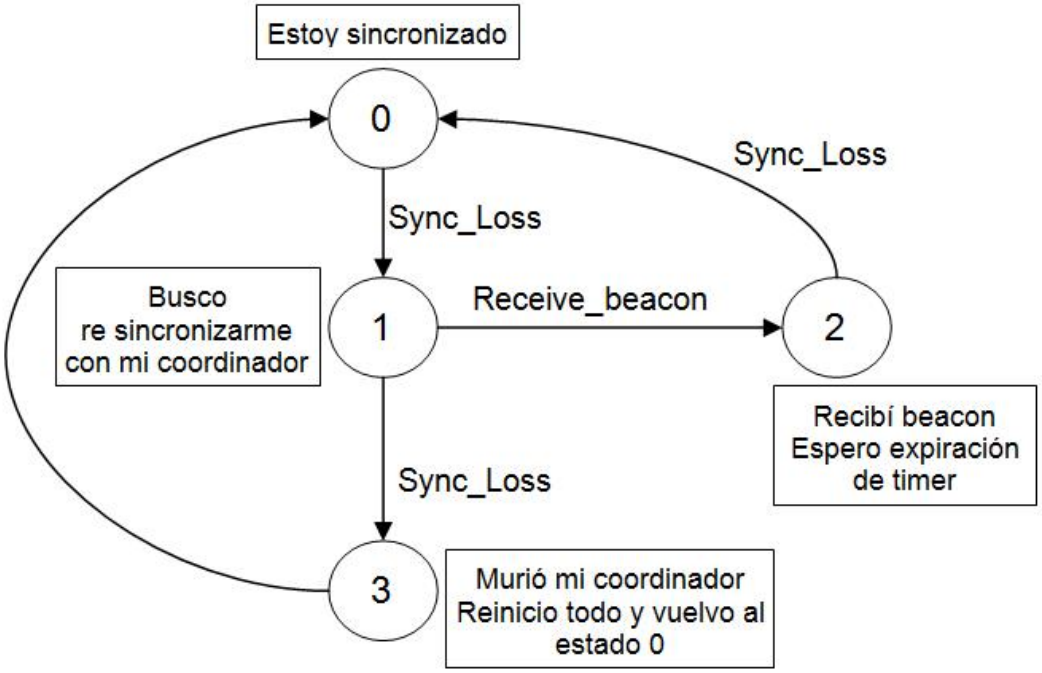


Figura 6.22: Máquina de estados que representa el funcionamiento de la implementación de la resincronización de un nodo.

- `event error_t MLME_COORD_GTS.indication(uint16_t DevAddress, uint8_t GTSCharacteristics, bool SecurityUse, uint8_t ACLEntry)`: Se dispara con el pedido de asignación de un GTS por parte de un nuevo End-Device. En este evento se postea la tarea para armar la dirección de red de un nuevo hijo.
- `event error_t MLME_DEVICE_GTS.confirm(uint8_t GTSCharacteristics, uint8_t status)`:Este evento se dispara en respuesta a un comando `MLME_DEVICE_GTS.request`, en caso de que el resultado no sea `SUCCESS`, se vuelve a llamar al mismo comando.
- `event error_t MLME_DEVICE_BEACON_NOTIFY.indication(uint8_t BSN,PANDescriptor pan_descriptor, uint8_t PenAddrSpec, uint8_t AddrList, uint8_t sduLength, uint8_t sdu[])`: Este evento es disparado cada vez que se recibe un beacon. Comienza a dispararse una vez sincronizado con el canal elegido, se utiliza para llamar a los diferentes comandos necesarios para iniciar la comunicación con un coordinador.

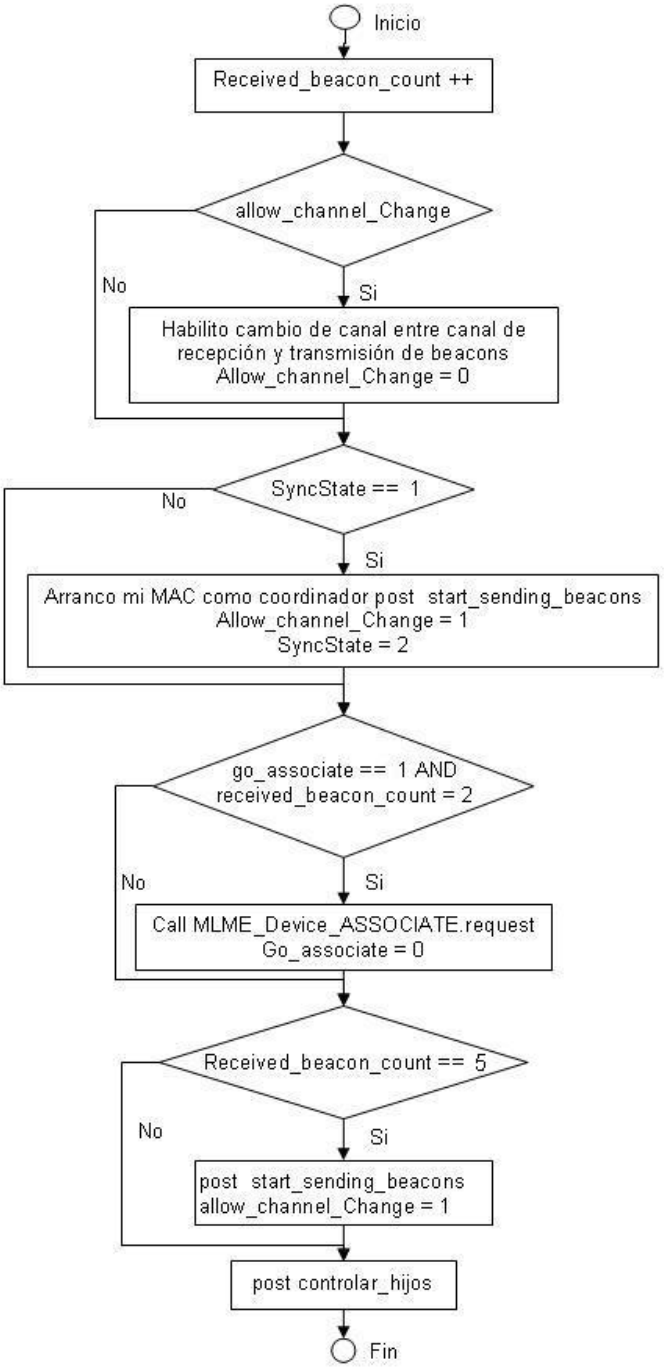


Figura 6.23: Diagrama de flujo del evento `MLME_DEVICE_BEACON_NOTIFY.indication`.

- `event error_t MLME_COORD_START.confirm(uint8_t status)`: Este evento es disparado una vez terminado el inicio del envío de los beacon, si el valor de `status` es `SUCCESS` dispara el evento `SplitControl.startDone` con el argumento `SUCCESS`.

- `event error_t MLME_COORD_ASSOCIATE.indication(uint32_t DeviceAddress[], uint8_t CapabilityInformation, bool SecurityUse, uint8_t ACLEntry)`: Se dis-

para cuando viene un pedido de asociación de un End-Device, este evento únicamente se encarga de postear la tarea `associate`.

- `event error_t MLME_DEVICE_ASSOCIATE.confirm(uint16_t AssocShortAddress, uint8_t status)`: Este evento se dispara una vez que llega la respuesta del coordinador, en caso de que `status` sea `SUCCESS` se setea la dirección de MAC asignada y se llama al comando `MLME_DEVICE_GTS.request`. En caso de que `status` no sea `SUCCESS` se realiza un nuevo escaneo en busca de beacon en los canales restantes.

- `event error_t MCPS_COORD_DATA.indication(uint16_t SrcAddrMode, uint16_t SrcPANId, uint32_t SrcAddr[2], uint16_t DstAddrMode, uint16_t DestPANId, uint32_t DstAddr[2], uint16_t msduLength, uint8_t msdu[100], uint16_t mpduLinkQuality, uint16_t SecurityUse, uint16_t ACLEntry)`: Este evento es disparado cuando un nodo recibe un dato proveniente de un hijo, lo que hace es enviarlo hacia su coordinador mediante el comando `MCPS_DEVICE_DATA.request`.

- `event error_t MCPS_DEVICE_DATA.indication(uint16_t SrcAddrMode, uint16_t SrcPANId, uint32_t SrcAddr[2], uint16_t DstAddrMode, uint16_t DestPANId, uint32_t DstAddr[2], uint16_t msduLength, uint8_t msdu[120],uint16_t mpduLinkQuality, uint16_t SecurityUse, uint16_t ACLEntry)`: Este evento se dispara cuando se recibe un dato proveniente del coordinador, en este caso se debe realizar el ruteo del paquete hacia el correspondiente destino. El paquete puede ser para el propio nodo, para uno de sus hijos o en broadcast.

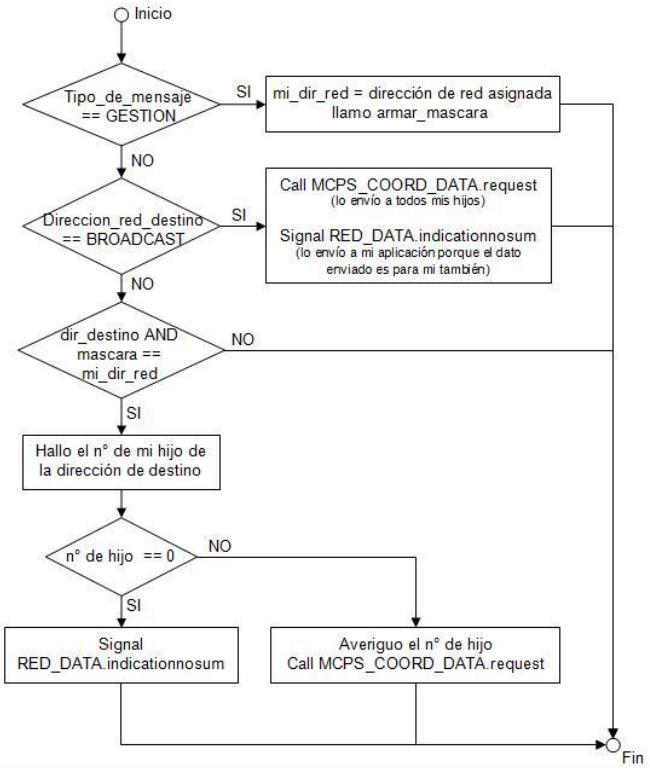


Figura 6.24: Diagrama de flujo del evento `MCPS_DEVICE_DATA.indication`.

El siguiente grupo de eventos no realizan ninguna función, fueron implementados para poder utilizar las interfaces correspondientes.

- `event error_t MLME_COORD_SET.confirm(uint8_t status,uint8_t PIBAttribute).`
- `event error_t MLME_DEVICE_SET.confirm(uint8_t status,uint8_t PIBAttribute).`
- `event error_t MLME_COORD_GET.confirm(uint8_t status,uint8_t PIBAttribute, uint8_t PIBAttributeValue[]).`
- `event error_t MLME_DEVICE_GET.confirm(uint8_t status,uint8_t PIBAttribute, uint8_t PIBAttributeValue[]).`
- `event error_t MLME_COORD_DISASSOCIATE.indication(uint32_t DeviceAddress[], uint8_t DisassociateReason, bool SecurityUse, uint8_t ACLEntry).`
- `event error_t MLME_DEVICE_DISASSOCIATE.indication(uint32_t DeviceAddress[], uint8_t DisassociateReason, bool SecurityUse, uint8_t ACLEntry).`
- `event error_t MLME_DEVICE_DISASSOCIATE.confirm(uint8_t status).`
- `event error_t MCPS_COORD_DATA.confirm(uint8_t msduHandle, uint8_t status).`
- `event error_t MCPS_DEVICE_DATA.confirm(uint8_t msduHandle, uint8_t status).`

Capítulo 7

Análisis de consumo

El objetivo de este capítulo es mostrar un análisis de consumo en base a los diferentes parámetros de la red para mostrar la dependencia de la duración de las pilas con los mismos.

7.1. Influencia de Standard IEEE 802.15.4 en el consumo

Como fue mencionado desde un principio, la red se ajusta al Standard IEEE 802.15.4 la cual define tiempos de inactividad con el propósito de ahorrar energía. Dichos períodos de inactividad van acompañados por períodos de actividad en donde todos los nodos se comunican e intercambian información. La conmutación entre un estado y otro es un ciclo periódico. Todos estos tiempos tienen una forma particular de representación, la cual se describe a continuación:
Cada ciclo está definido entre dos tramas de beacons consecutivos, en donde se presentan ambos períodos mencionados. La figura 3.2 del capítulo 3 ilustra lo mencionado anteriormente. El Standard define el símbolo como la unidad atómica de representación del tiempo y su valor es igual a $16\mu s$. Todo período de tiempo debe ser un múltiplo entero de símbolos. Como se puede observar esta unidad es muy pequeña en comparación con los tiempos de actividad e inactividad. Por eso es que los tiempos son medidos por las variables:

- BO - Beacon Order
- SO - Superframe Order

Para representar el tiempo activo se lo hace haciendo referencia a SO, el cual quiere decir[1]:

$$Tiempo\ supertrama\ (en\ símbolos) = aBaseSuperframeDuration * 2^{SO}\ símbolos$$

donde

$$aBaseSuperframeDuration = aBaseSlotDuration * aNumSuperframeSlots$$

[1] Ecuaciones definidas en la Standard IEEE 802.15.4 [4]

con:

aBaseSlotDuration = 60 símbolos;

aNumSuperframeSlots = 16 slots;

El BO está definido de la misma manera, pero en este caso, la cantidad de símbolos representa el total que caben en un ciclo completo. Por lo tanto SO nunca puede ser mayor que BO.

Se hace aquí un aclaración muy importante: la aplicación desarrollada no utiliza cada símbolo como es definido en el Standard, sino que tiene una pequeña diferencia en su valor.

El reloj utilizado es de 32768 Hz, lo que da como resultado un ciclo de *30.5* μs, y no podría medir símbolos. La solución que se encontró (solución implementada por Hurray) es trabajar con la igualdad: *30.5* μs = 2 símbolos.

Esta aproximación provoca que los tiempos no sean exactamente como se definen en el Standard, pero será perfectamente coherente en toda la red.

7.2. Cálculo del Beacon Order.

Ya descriptos los parámetros que intervienen en la duración de los tiempos, se pasará a definir los valores a utilizar en la red para responder adecuadamente al compromiso entre mínimo consumo y máximo desempeño de la red.

Por un lado, el máximo tiempo de actividad permite tener mayor tráfico y menor dificultad de sincronismo entre los motes, pero ello provoca por el contrario una reducción en la duración de las pilas. La dificultad de sincronismo se debe a que con grandes tiempos de inactividad y breves períodos de actividad, se reduce la probabilidad que los relojes no se desfasen de manera significativa y el beacon llegue en el corto lapso en el que el nodo asociado tiene el transceiver encendido. Por lo tanto para definir los parámetros involucrados, entiéndase BO y SO; primero se deben analizar las características particulares[2] de la red. El primer punto a tener en cuenta es no provocar retardos significativos en los datos, por lo que se definió que el BO debería ser tal que el tiempo entre beacons fuera aproximadamente de 1 minuto. También se tuvo en cuenta que el menor tiempo de muestreo admitido es también de 1 minuto; por lo tanto el mote debería ser capaz de enviar los datos a una cadencia mayor que con la que los genera.

Se concluye entonces que el tiempo debe ser igual a 1 minuto:

$$aBaseSuperframeDuration * 2^{BO} * t\_símbolo = 60\text{s}$$

Por lo tanto:

$$BO = log_2\ (60s\ /\ (aBaseSuperframeDuration * t\_símbolo))$$

Tomando los valores correspondientes:

$$t\_símbolo = 1/2\ *\ 1/32768Hz = 15.26\mu s$$

$$aBaseSuperframeDuration = 960$$

Tenemos:

$$BO = 12$$

[2]topología Cluster Tree, dimensión de la red, tamaño de los paquetes, etc

7.3. Cálculo del Superframe Order.

Una vez definido el Beacon Order, se procedió a calcular el SO mínimo necesario para cubrir las necesidades de tráfico. Se realizaron ciertas hipótesis que se tomaron como punto de partida, a saber:

- Debido a la topología elegida para la red (Cluster Tree), el nodo existente entre el nodo sumidero y el resto de la red será quien soporte el mayor tráfico ya que deberá reenviar los datos de la totalidad de la red o su rama.
- Se tratará con una red de 256 nodos máximos.
- El estudio se restringe a un caso de distribución uniforme de motes en el área de interés, con 40 nodos por rama[3].
- El período de muestreo típico es de 10 minutos.
- Los datos enviados hacia el sumidero se realizan enviando los paquetes hacia el coordinador en las ranuras de tiempo garantido (GTS).
- A modo de aplicación, no estará permitido tomar más de una medida en cada supertrama, es decir si el tiempo entre beacon es T_{beacon} y el tiempo entre muestras adquiridas es T_{adq}, entonces se cumple que $T_{adq} > T_{beacon}$. Esto lo que asegura es que la aplicación no generará mas de un paquete entre dos tramas beacon. Es decir, que en funcionamiento normal se enviará a lo sumo un paquete de aplicación con las medidas tomadas. Existe el caso en que haya pasado una supertrama entera y el dispositivo no tenga datos para enviar a su coordinador en el GTS.

Se debe encontrar el SO de la red con el propósito de que el sumidero pueda recibir todos los paquetes de aplicación (medida adquirida) de todos los motes de la red. Será necesario citar las siguientes constantes:

- 6 bytes es el largo de los mensajes de la capa de aplicación.
- 10 bytes es el largo del encabezado de la capa de red.
- 12 bytes de mac header, más 2 bytes de checksum; con un máximo de 120 bytes de carga (de los cuales utilizamos solamente 16).
- El header o preámbulo que define el Standard IEEE 802.15.4 en la capa física son 6 bytes.

En total suman 36 bytes.

Otros puntos a tener en cuenta:

[3]Se define como rama a todos los nodos de la red que dependen de un solo nodo asociado directamente al sumidero para enviar los datos al mismo

- Entre cada trama hay un tiempo mínimo que se debe respetar (que es el tiempo que se asume le lleva al software guardar estos datos en RAM y estar listo para recibir una nueva trama), y en el Standard se especifica como *macMinLIFSPeriod* y es de 20 bytes.
- Si se le suma que es una trama con reconocimiento, se le agrega el tiempo t_{ACK}; el cual es de 5 símbolos - 3 bytes.

Todo esto se resume en un total de 59 bytes por trama; lo que equivale a 118 símbolos.

Finalmente considerando que se utiliza GTS para enviar las tramas, y el sumidero tendrá que recibir datos de 40 motes por cada mote asociado a él cada 10 minutos, esto equivale en promedio, a 4 tramas tramas por time slot.

*118 símbolos/trama * 4 tramas/time slot = 472 símbolos/time slot*

para cubrir los requisitos mencionados anteriormente.

Son 16 time slots los que conforman el período activo; por lo tanto se debe cumplir con la siguiente condición:

n^o símbolos/time slotn^o time slots<aBaseSuperframeDuration*2^{SO} símbolos*

donde el parámetro a determinar es el SO.

$$\log_2((n^o símbolos/timeslot * n^o timeslots)/aBaseSuperframeDuration) < SO$$

$$2.98 < \text{SO}$$

Finalmente tomamos SO = 3, lo cual equivale a un tiempo de actividad de:

tiempo de supertrama = aBaseSuperframeDuration2^{SO} *t_símbolo*

tiempo de supertrama = 117 ms

7.4. Cálculo del consumo.

Con el SO y BO definidos se cumple con una parte del objetivo planteado, el de transportar adecuadamente el tráfico que se generará en el modo de funcionamiento normal de la red.

Resta cumplir con el consumo, alcanzar una durabilidad de la red de un año. Para hacer esto se necesitan nuevos datos que inciden directamente en el resultado.

Partiendo de la base que utilizamos dos pilas AA, a las cuales se le puede atribuir una capacidad de carga de 2700 mAh y se les puede extraer hasta el 50 % de la carga[14]. Por lo tanto la carga disponible total es aproximadamente 2700 mAh.

La carga de las pilas es la energía disponible, ahora resta definir el consumo para alcanzar la vida útil esperada.

De las mediciones realizadas y de la hoja de datos de los motes; los datos con los que se cuentan sobre la corriente consumida en cada período es:

- período activo - 21 mA
- período inactivo - 8.7 μA Aquí está contemplado el consumo del microprocesador en estado *STANDBY*.

Cabe aclarar que el período activo en cada mote se multiplica por dos ya que el nodo debe funcionar como end_device y también como coordinador. A esto hay que sumar el tiempo de guarda existente antes del inicio de cada supertrama para respetar el tiempo que insume el transceiver en iniciarse por completo. La guarda utilizada equivale a 2200 símbolos[4]:

*tiempo de guarda = n^o símbolos * t_símbolo*

*tiempo de guarda = 2200 * 1/2 * (1/32768)ms = 33.57 ms*

Considerando entonces el tiempo de la supertrama y el tiempo de guarda:

tiempo de actividad = tiempo de guarda + tiempo de supertrama = 150.9 ms

Veamos si estos valores en conjunto con los calculados para BO y SO en las secciones anteriores son coherentes con las aspiraciones.

A partir del tiempo de actividad y el tiempo de guarda, cada un minuto habrá menos de 0.5 segundos de actividad y el resto del tiempo es inactividad. Este tiempo de actividad se

[4]Esta cantidad de símbolos fueron valores provistos en la implementación Hurray, cabe aclarar que excede los tiempos necesarios que se especifican en el datasheet, pero que sirven para asegurarse completamente de que el transceiver se encuentre listo a la hora de recepción del beacon

da, tomando en cuenta que en un tiempo de beacon hay dos tiempos de supertrama y que para el SO elegido la duración del tiempo de actividad será de 0.15 segundos aproximados. Esto implica 0.3 segundos por minuto, pero tomando una guarda para cubrir errores se tomó 0.5 segundos. En el transcurso de una hora el período de actividad será menor que 30 segundos en total.

Menos de 30 segundos en una hora, equivale a una relación menor a 30/3600 = 1/120.

Ponderando los valores de corriente que consumen en ambos períodos se tiene:

$I_{MEDIA} = 21mA * 1/120 + 9*10^{-3}$ mA * 119/120 = 0.184 mA

Si en total se tiene 2700 mAh disponibles ⇒ 2700 mAh / 0.184 mA = 14674 horas

Considerando las horas que tiene un año:

365 días/año * 24 horas/día = 8760 horas/año.

Se concluye que con estos valores se alcanza teóricamente un año y ocho meses de duración de las pilas.

Resumiendo, si se cumplen los siguientes puntos:

- BO = 12, equivalente a un tiempo entre beacons de 1 minuto.
- SO = 3, equivalente a un tiempo de actividad de 0.3 segundos aproximadamente. No olvidar que este tiempo está multiplicado por dos porque el nodo funciona como end_device y como coordinador.
- $I_{MÁX}$ = 21 mA.
- I_{MIN} = 9 μA.

se alcanzaría el objetivo del proyecto.

Para finalizar también se puede observar que la cantidad de horas de rendimiento es mucho mayor que las necesarias para un año. Esto se hizo con la finalidad de tener un margen para el error y contemplar los instantes de toma de medidas de los sensores. Cabe aclarar que para el análisis se supuso una topología de 40 nodos por rama, este valor por rama es el máximo admisible para la no pérdida de datos. Este valor surge del SO elegido para lograr el bajo consumo, es decir se podrían tener 7 ramas conectadas al sumidero de 40 nodos con lo cual se llega a los 256 nodos pedidos.

7.5. Medidas en laboratorio

Una vez finalizada la etapa de diseño y desarrollo se procedió a tomar las medidas pertinentes para verificar si cumplía con lo mencionado en las secciones anteriores. Las medidas se efectuaron con todas las funcionalidades activas, con el propósito de hacerlas lo más fieles posible con la implementación utilizada en campo.

Se pudieron obtener los valores de la corriente de actividad e inactividad del hardaware en su conjunto. Estos se encontraron dentro de lo esperado [5]:
$I_{ACT} = 20.9$ mA.
$I_{INACT} = 8.4$ μA.
Para poder observar la forma de onda de la corriente se colocó una resistencia en serie, y se verficó que fuera acorde con lo esperado. En la siguiente figura se muestra la forma de onda obtenida:

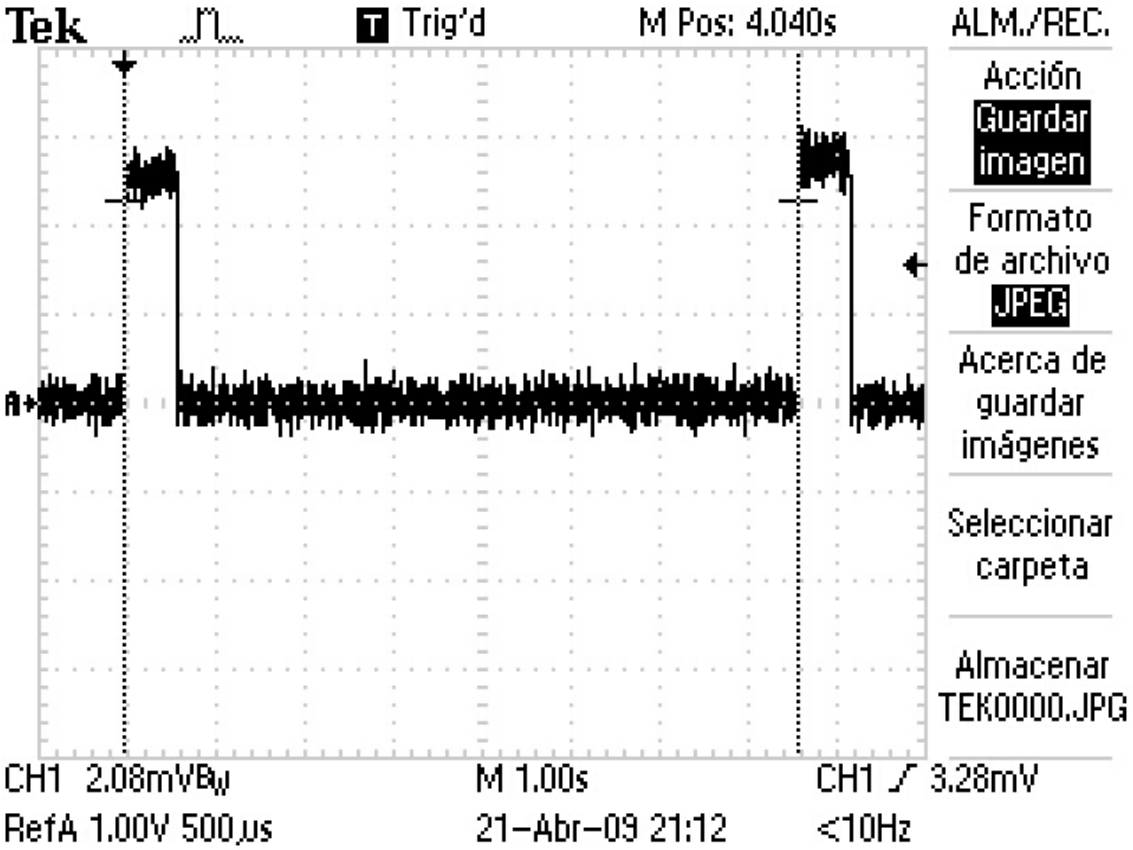


Figura 7.1: Formas de onda en la corriente del Tmote operando en régimen

En esta figura claramente se logra observar el período de la supertrama:
$T_{beacon} = 7.6$ s
el cual concuerda con el BO = 10(tomado así en para poder visualizar los dos escalones de corriente) seleccionado para esta prueba [6].

[5] Según datasheet de TmoteSky [8]
[6] No olvidar que una se corresponde con la supertrama escucha del macCoordinador, y la otra es la trama

También se realizó la medida del tiempo de guarda considerado necesario para el encendido del transceiver [7] el cual tiene una medida teórica de $T_{GUARDA} = 34.37ms$. Para poder medir este tiempo en el evento *bi.fired* se enciende un led hasta terminar la supertrama. De esta manera se puede observar en la siguiente figura un escalón más pequeño de la supertrama. El tiempo de diferencia entre el escalón principal y el otro más pequeño se corresponde con el *tiempo de guarda*

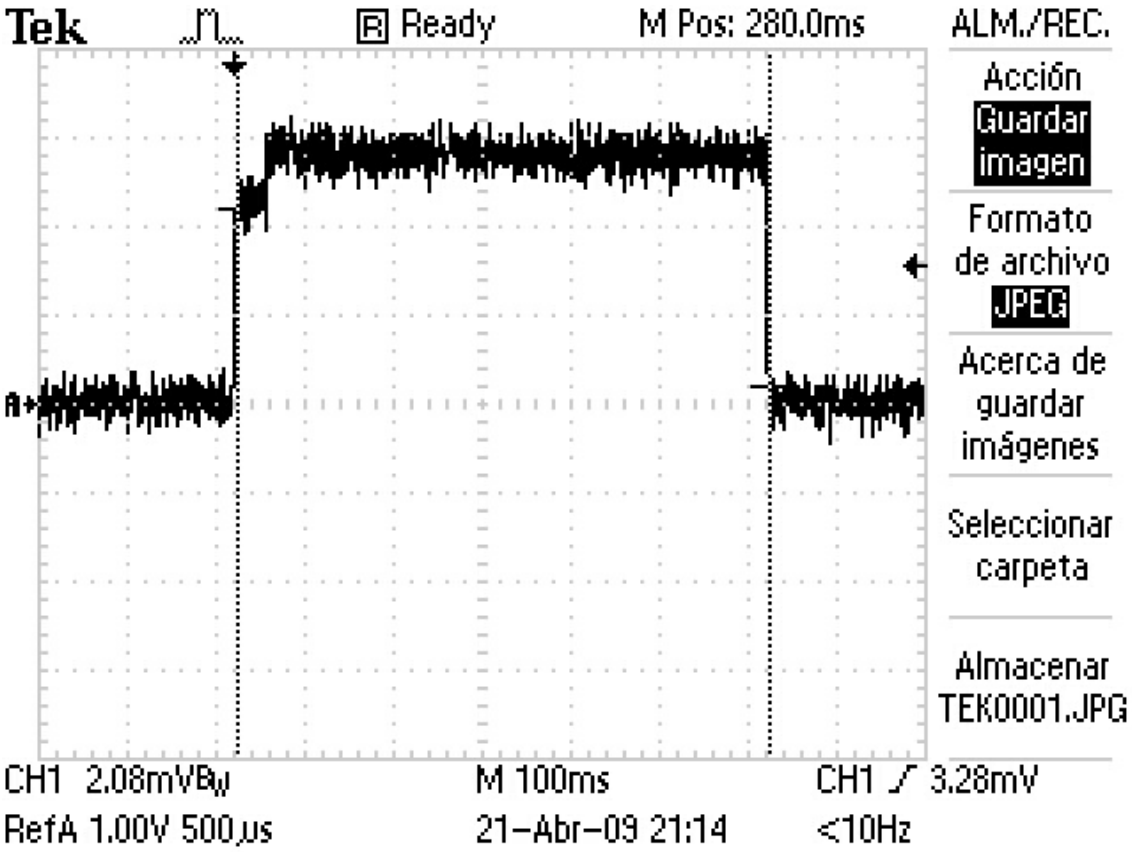


Figura 7.2: Direcciones de MAC y red utilizadas en las pruebas

Realizando el zoom en la zona del escalón se obtiene la medida

tiempo de guarda = 38ms

el cual se asemeja al valor teórico esperado.

a recibir, en el canal físico del device. Ambos períodos son simétricos en el tiempo, y uno se encuentra justo en la mitad del otro

[7] Este es el tiempo que transcurre entre que se disparan los eventos *before_bi.fired()* y el evento *bi.fired* y contempla los tiempos de demoras del oscilador interno del Chip CC2420 en estabilizarse y demás, cuya especificación completa se encuentra en su datasheet [9], medidos en ticks como se explica en 7.4

Capítulo 8

Pruebas generales

Se resume aquí el proceso de pruebas al cual fue sujeta la aplicación y la entidad de red diseñada. Se realiza una introducción sobre el ambiente de test, y los resultados obtenidos en cada etapa.
Cabe destacar que la mayoría de las funcionalidades Mac y física ya se encontraban con las modificaciones hechas y completamente probadas. El proceso de test aquí descripto, fue realizado en paralelo con el de implementación de estos propios componentes.
Una gran dificultad en esta etapa fue la limitante impuesta por la capacidad en memoria del mote, ya que la función de debug, printf[1], utiliza un amplio espacio de ROM del controlador (5KB de 48KB que posee el procesador, e incrementándolos según la cantidad de llamados a la función).

8.1. Pruebas componentes de alto nivel Sumidero y NoSumidero

En esta sección se explica la forma de testeo de los componentes SumideroM.nc y NoSumidero.nc.
Para poder testear si estos responden con lo acordado frente a los eventos que la capa de red le señala, se debió de poder simular este componente. Esto es porque en esta etapa del proyecto no se contaba con su implementación. En la figura se muestra como fueron implementados estos componentes de test. Para el componente SumideroM ya se contaba con la interfaz USB que maneja los mensajes de aplicación definidos, y se podían ver los mensajes en la aplicación de escritorio desarrollada en java[2]. La siguiente figura muestra el esquema de test de los dos archivos finales.

[1] Por más detalles dirigirse al anexo C

[2] Cabe mencionar que la función printf nunca fue dejada de lado por su sencillez a la hora de visualización de las variables de interés

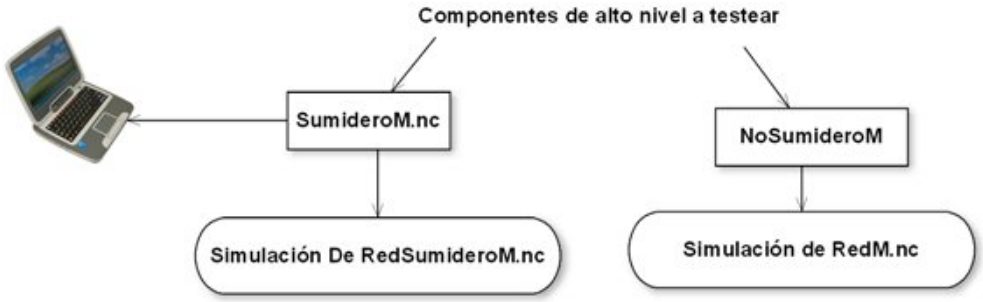


Figura 8.1: Esquema de test de dispositivo de alto nivel

Se fueron testeando uno por uno los casos de uso sin mayor dificultad, primero para el caso del Sumidero, y luego para el NoSumidero.

- Envío de mensajes de nacimiento y muerte.
- Solicitud del sumidero de cambio de parámetros.
- Envío de datos de sensores.

8.2. Pruebas componente RedM

Para poder testear correctamente este componente, se debieron simular los comandos desde la capa de aplicación (solicitud de servicios de envío de datos y demás) y se observaron como estos comandos repercutían a en la interacción de RedM con las dos interfaces de enlace MAC [3]. También se simularon los eventos de MacDevice y MacCoord simulando el arribo de datos del transceiver o de solicitudes de servicios, y se observó como estos mensajes interactuaban hacia la aplicación. Es decir, en el sentido contrario al caso antes mencionado. En la figura 8.3 se muestra este esquema:

[3]de las cuales se quitó su interacción con la física para poder debugguear sin tener la limitante en ROM

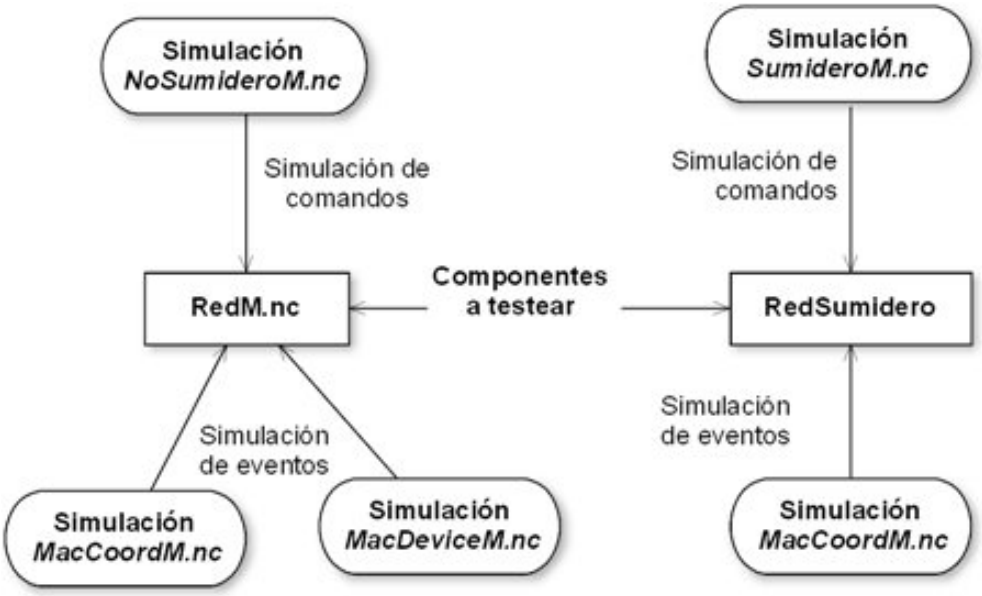


Figura 8.2: Esquema de test de entidad de red

A continuación se listan algunas de las casuísticas principales probadas:

- Sincronismo con los dos enlaces MAC MacCoord y MacDevice. [4]. Respuesta al desincronismo con el coordinador.
- Creación de dirección de red y selección de parámetros MAC para inicializar la Red Coordinador.
- Solicitudes de servicios en el enlace en que funciona como device y respuesta a las solicitudes del lado que funciona como coordinador (Asociación y gestión de GTS)
- Procesado y generación de tramas de ambas interfaces.
- Multisalto
- Ruteo[5]

8.3. Pruebas de la integración entre todos los componentes

Estos fueron los resultados de las pruebas de todos los componentes interactuando en tiempo real, y el sumidero funcionando con la interfaz java desde la cual se fueron probando las secuencias de los casos de uso de la aplicación.
Al igual que en los otros procesos, fue de gran utilidad el uso de los leds, ya que estos servían como indicadores del estado en que se encontraba el sistema (tiempo activo de la supertrama del device, tiempo activo de la supertrama del coordinador, y tiempo de inactividad). A continuación se muestra en la siguiente tabla las direcciones utilizadas para las direcciones a nivel de MAC y red.

[4]Esto es tener un dispositivo escuchando tramas del coordinador y el poder postularse en otro canal

	mote Sumidero		mote 1		mote 1.1	
Direcciones	Como device	Como Coordinador	Como device	Como Coordinador	Como device	Como Coordinador
PAN	n/a	0x0001	0x0001	0x0082	0x0082	0x0183
MAC	n/a	0x0034	0x3401	0x3401	0x0101	0x0101
RED	0x00000000		0x20000000		0x24000000	

	mote 2		mote 2.1	
	Como device	Como Coordinador	Como device	Como Coordinador
	0x0001	0x0083	0x0083	0x0184
	0x3402	0x3402	0x0201	0x0201
RED	0x40000000		0x44000000	

	mote 2.2	
	Como device	Como Coordinador
	0x0083	0x0185
	0x0202	0x0202
RED	0x48000000	

Figura 8.3: Direcciones de MAC y red utilizadas en las pruebas

con el coordinador

[5]Es decir, elegir la correcta dirección del "*Hijo*"a cual le corresponde el paquete en función de la dirección de red de destino en el paquete recibido por el "*Padre*"

Capítulo 9

Balance y conclusiones

En este capítulo se exponen resultados y reflexiones referentes al desarrollo del proyecto, lo cumplido y lo no cumplido respecto a lo propuesto en primera instancia. También se describen mejoras a realizarse en la aplicación que no se implementaron porque o bien excedían el alcance de este proyecto, o por no contarse con los recursos necesarios en el hardware.

9.1. Logrado y no logrado a la fecha de finalización

En general se ha logrado cumplir con los objetivos principales del proyecto, los cuales son un correcto funcionamiento de la red y un tiempo de vida estimado de la misma superior a un año. Con respecto al primer objetivo; si bien existen pérdidas de sincronismo que podrían influir en la duración de las pilas de los motes, se han logrado testear con éxito todos los casos de uso, y se han contemplado los problemas de sincronismo, a tal punto que frente a la caída de un nodo de la red, éste puede recuperarse (él y el resto de los hijos o motes de los cuales este depende) y volver a asociarse.

Se probó la correcta recepción de datos con una red mínima y se evaluó la no pérdida de los mismos para un máximo de 40 motes por rama. Esta restricción se debe al superframe order elegido, que para lograr un adecuado consumo fue fijado en 3. Aquí se hace evidente el claro compromiso entre el consumo y el desempeño de la red.

Respecto al tema de consumo, si bien no se contó con la instrumentación adecuada para realizar medidas de carga o corriente acumulada por un tiempo prolongado y así estimar la duración de las pilas, se realizaron medidas de tiempo y corriente que permitieron extrapolar el consumo de las pilas en un año. Se ha obtenido un resultado satisfactorio, ya que se han logrado valores de corriente tales que permitirían un tiempo de vida de las pilas que supera el año de duración. Esto es así debido al análisis teórico realizado, el cual contempla el peor caso de funcionamiento[1].

Como pendiente quedaría realizar la medida de consumo correspondiente una vez que se encuentre disponible el dispositivo previsto para este fin, el cual se contaba que estaría a disposición al comienzo del desarrollo del proyecto. Por razones ajenas a este grupo de

[1] Esto abarca la pérdida seguida de sincronismo, el aumento de los tiempos actividad y el consumo del sensor

trabajo no se pudo contar con el mismo en tiempo y forma.

9.2. Posibles mejoras, soluciones propuestas

9.2.1. Mejor uso del tiempo de actividad para transmitir datos

Esta mejora surge en el caso en el que se tiene que mandar datos de hijos, nietos y varios nodos de la red en un sólo GTS. Si bien existe la solución de si al no tener espacio en los GTS, se espere la siguiente supertrama, existe la posibilidad de agrupar en una misma trama MAC varios paquetes de red. Se estaba en la disyuntiva de saber cuándo enviar la trama en los casos que ésta no se encuentre con su capacidad de carga agotada.
Debido a la escasez de tiempo y de memoria (RAM, ROM) no se pudo implementar esta funcionalidad. Por un lado era requisito disponer de más memoria para poder realizar una solución rápida y poco eficaz; en el otro extremo con más tiempo se podría desarrollar una solución que aprovechara en mejor medida el espacio de memoria disponible.

9.2.2. Agregado de marca de tiempo en los mensajes

Actualmente los datos tomados de los sensores son generados y enviados en mensajes que únicamente se diferencian por un número de secuencia. La implementación actual está tomando el instante de arribo del mensaje al sumidero, como la hora que identifica esa medida, claro está que no es lo más conveniente ya que los datos pueden tener retardos de hasta 5 minutos[2] en el peor de los casos.

La solución a este problema es que cada nodo cuente con un registro vinculado con la hora, y a cada mensaje que contenga la información de la medida de los sensores se agregue el valor de este registro. Esto implica un protocolo a nivel de aplicación que involucre la PC quien sería la referencia de tiempo. En varias instancias se evaluó el impacto que el desarrollo de esta funcionalidad podría tener sobre el plan de proyecto y la decisión fue dejarlo para la etapa final, a la cual no se llegó con suficiente tiempo como para poder implementarla.

9.2.3. Algoritmo de selección de canal de nueva PAN

El algoritmo de selección de canal para una nueva PAN implementado en esta aplicación no contempla el caso de estación oculta. Este es un problema que se da cuando dos estaciones se pueden comunicar con otra intermedia, pero no entre ellas. Lo que sucede es que las estaciones que no se comunican entre sí no podrán ver la actividad que genera cada una de ellas, y al querer comunicarse con el nodo intermedio simultáneamente, se produce la colisión de los datos.
En la red de sensores implementada se puede dar el caso de que un mote al realizar un scan, no encuentre actividad en el canal en el que escucha las tramas beacon el coordinador con el cual se va a asociar. Por el funcionamiento del algoritmo, el mote seleccionará el último canal en el que escuchó actividad para crear la nueva PAN, éste puede coincidir con el canal en el cual escuchaba el coordinador. Durante las pruebas no se presentó el

[2]Debido a los valores supuestos de 10 saltos máximo, a un BO tal que permita medio minuto por salto

problema por que las distancias eran acotadas, al igual que el número de nodos, y los canales la mayoría de las veces fueron fijados en valores conocidos.

Por motivos de tiempo no se pudo corregir este problema, una posible solución (la más viable) sería realizar un algoritmo centralizado en la PC para la selección del canal en el cual se comenzará la nueva PAN. Esto se considera así debido a la potencia de cálculo que posee una PC en comparación con un mote.

9.2.4. Algoritmo de cálculo de SO y BO dinámicos en la red

En el capítulo de análisis de consumo, fue muy difícil estimar el peor caso de congestión para calcular el BO y SO que optimizan el consumo. Esto es debido a que son muchas variables en juego, y que el evento de adquisición de datos, y el tiempo de la supertrama son asíncronos. Además, el período de muestreo es otra variable que juega un papel crucial, ya que de ésta depende el flujo de datos que circula en la red.
Por otro lado, los datos que el Sumidero puede envía al PC (mediante el uso de mensajes de configuración) son suficientes para poder armar el grafo de la red. De esta manera, conociendo el BO y SO(fijos en nuestra aplicación) de la red y el período de muestreo (el cual es seteado desde la propia aplicación del PC) se podría ejecutar un algoritmo en el PC que calcule en función de estos parámetros el SO y el BO óptimos de la red, es decir, que permitan el correcto envío de datos sin congestionar la red, y que optimicen los tiempos de actividad para minimizar el consumo. El resultado podría ser informado a los nodos en la red para que ajusten su SO y su BO[3] optimizando la performance de la red.

[3]El caso de BO requiere una implementación extra ya que cambios en el BO repercuten en la sincronización con los hijos

Apéndice A

NesC: Introducción al lenguaje

El lenguaje NesC (Network Embedded Systems C), es un lenguaje orientado a componentes, que mantiene la sintaxis de C y está especialmente diseñado para trabajar en redes de sensores. Por este motivo está pensado para ser ejecutado en plataformas de bajos recursos.

Toda aplicación implementada en NesC, se ejecutará sobre el sistema operativo de tiempo real TinyOS. Por este motivo, la aplicación podrá utilizar los componentes implementados en TinyOS para control del hardware sobre el cual se correrá la aplicación. Con los componentes implementados en TinyOS se podrá tener control sobre el microcontrolador, la UART, los Timer, el chip de radio, etc..

Necesariamente cada componente estará compuesto por dos archivos, un archivo donde se encuentra la configuración y la implementación y otro donde se encuentra el módulo. Es posible agregar archivos con encabezados y estructuras de datos ".h".
La *Configuración* es utilizada para crear una librería, no se utiliza para aplicaciones comunes.
La *Implementación* es donde se realiza el cableado (o wiring) de la aplicación, es decir se dice que componentes y que interfaces implementadas por otro componente que se va a usar.
El *Módulo* es en donde se realiza la implementación de cada una de las interfaces que maneja. Un módulo puede proveer o usar diferentes interfaces. Las interfaces que se proveen deben estar completamente implementadas tal cual su definición, y las que se usan se pueden llamar cuando se necesario. A su vez los módulos poseen la particularidad de poder ser declarados genéricos o no. En caso de que sea genérico cada vez que se haga referencia a este módulo se podrá crear una instancia nueva del mismo, es decir que serán dos bloques independientes de código [1] y no el mismo compartido.

[1] Esto conlleva a duplicar el tamaño del código ya que son dos bloques de código diferentes

```
module BlinkC
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
  uses interface McuSleep;
}
implementation
{
int contador = 0;
  event void Boot.booted()
  {
    call Timer0.startOneShot( 100 );
  }

  event void Timer0.fired()
  {
  call Timer1.startOneShot( 10000 );
  }

  event void Timer1.fired()
  {
    call Timer1.stop();
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC, McuSleepC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;


  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
  BlinkC.McuSleep -> McuSleepC;
}
```

Figura A.1: Ejemplo del código de un módulo y de un archivo de implementación.

Apéndice B

Lineamientos para la puesta en marcha de la red

En esta sección se explican los procedimientos necesarios a la hora de realizar el despliegue de la red. Se tratan temas referentes a cómo programar el hardware, los cambios en los archivos (.h) necesarios y demás.

B.1. Programación del mota vía USB

Para programar el mote desde el PC se requiere tener instalado el sistema operativo tinyOS. Para ello existen varios pasos a seguir, que se detallan en [13]. Una vez instaladas las librerías allí mencionadas, y realizadas todas las configuraciones necesarias, se debe copiar la carpeta *Archivos fuente del software de RSIS* que se encuentra disponible en el CD, dentro de la carpeta *cygwin/opt/tinyos-2.x*. Luego desde la consola de cygwin [1] posicionarse en la carpeta *cygwin/opt/tinyos-2.x/repositorio/rsis/apps*. Allí se observan los dos proyectos principales, diferenciados para el caso del Sumidero y para el caso del No sumidero, como se menciona en el capítulo 5.

B.2. Configuración de los diferentes nodos en la red

Para programar el Sumidero no es necesaria ninguna configuración extra; simplemente posicionarse dentro de la carpeta *Sumidero* de *apps*, y ejecutar el comando *make tmote install*.
Como se mencionó en el capítulo 5, cada nodo diferente al Sumidero consta de un identificador que lo diferencia del resto de los motes dentro de la red. Para poder hacer esta distinción a la hora de hacer el despliegue de los nodos en el predio, es que se debe de cambiar este identificador a la hora de programarlos, y utilizar uno diferente en cada nodo NoSumidero. Este es un identificador de 8 bits que se encuentra accesible en uno de los archivos de configuración. Para cambiarla, abrir el archivo *NoSumidero.h* y renombrar la constante *MOTE_ID* con el valor deseado. Luego, al igual que en el caso del Sumidero, posicionarse en la carpeta NoSumidero de app, y ejecutar el comando *make tmote install*.
Es importante notar aquí, que la completa inicialización de un nodo NoSumidero puede

[1] Cygwin es una consola Linux requerida para ejecutar TinyOS en Windows: www.cygwin.com/

insumir un tiempo aproximado a 30 minutos. Esto se debe a que cada mote que es encendido, realiza una ED_SCAN con una duración de 1 minuto por canal. Una vez finalizado el primer SCAN, se realiza el PASSIVE_SCAN en los canales que se escuchó actividad. Considerando el peor caso (que se escuche actividad en los 16 canales) y que el transceiver maneja 16 canales se concluye que son 30 minutos de inicialización.

Apéndice C

Herramienta para depuración: Función `Printf`

En este anexo se pretende hacer una descripción superficial del componente nesC utilizado para depurar el código.

Al comienzo del desarrollo de la programación no era conocido este componente, y todo el testing de las aplicaciones consistía en cambiar de estado las luces de los leds del mote en función de los valores que tomaban las variables o eventos que se producían.

Como ya fue mencionado en el capítulo 2, los motes cuentan con tres leds que son para uso del programador. Estos leds son de color rojo, verde y azul, numerados como led0, led1 y led2 respectivamente.

Cabe mencionar que hay incluidos también dos leds más junto al conector USB, pero la finalidad de éstos es indicar cuándo está llegando o se está enviando información por el USB. El led rojo indica que está enviando información, mientras que el verde indica que la está recibiendo.

En la siguiente figura se puede apreciar la ubicación de los leds en el mote:

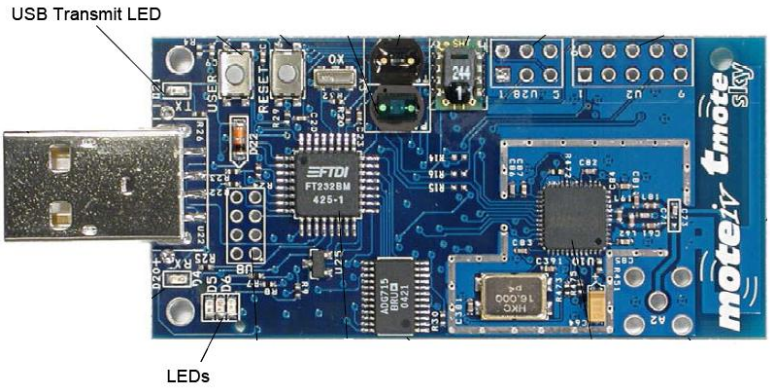


Figura C.1: Ubicación de los leds en el mote.

Un ejemplo de esta depuración se puede ver en el siguiente bloque de código:

```
if (variable == 5){
    call Leds.led0On(); // Si la variable es igual a 5 encender el led rojo.
} else if (variable == 7){
        call Leds.led1On(); // Si la variable es igual a 7 encender el led verde.
    }else{
        call Leds.led2On(); // En su defecto encender el led azul.
    }
```

Figura C.2: Código de depuración con leds.

No es muy difícil de suponer que esta modalidad de testing es prácticamente inviable, por lo que hubo que recurrir a otra forma de debugear el código.

Esta nueva forma de hacerlo es con la ayuda del componente PrintfC.

Este componente provee 2 interfaces que nos interesan:

- SplitControl
- PrintfFlush

La interfaz SplitControl está relacionada con la inicialización del componente:

- `command SplitControl.start()`: Arranca el componente; dispara el evento startDone(error_t error)

- `command SplitControl.stop()`: Detiene el componente; dispara el evento stopDone(error_t error)
- `event void startDone(error_t error)`: Notificación del resultado de la ejecución del comando start()
- `event void stopDone(error_t error)`: Notificación del resultado de la ejecución del comando stop()

En cuanto a la interfaz PrintfFlush, está vinculada al envío de los datos:

- `command error_t flush()`: Con la ejecución de este comando, se envían todos los mensajes guardados en el buffer de salida.
- `event void flushDone(error_t error)`: Notificación del resultado de la ejecución del comando flush()

A continuación se describirá cómo utilizar este componente para enviar los mensajes por el USB. La función principal es la función printf(), a la cual se debe pasar la cadena de caracteres que se desea imprimir en pantalla. Esta función admite también la impresión de valores de variables, siendo ésta la mayor utilidad de printf().

Un ejemplo del uso de printf:

variable = 9;

printf("Valor de la variable: % x", variable);

esto imprimirá:

Valor de la variable: 9

A lo que se puede concluir que es mucho más simple que utilizar leds.

Algunos detalles que restan mencionar:

- El símbolo de % es para indicar como se debe imprimir el valor: si es x es en hexadecimal (minúsculas); X es en hexadecimal también pero en mayúsculas; i es decimal; etc.
- Se pueden incluir varios parámetros en el mismo printf, los cuales deben ir separados con ",": *printf("Valor de las variables: % x % i", variable1, variable2);*
- La función printf solamente guarda en el buffer de salida los mensajes que se le solicita enviar, es necesario invocar luego el comando flush() de la interfaz PrintfFlush para que efectivamente se envíen los mensajes.
- No se pueden incluir más de un comando PrintfFlush.flush() dentro de cada evento.

- El punto anterior obliga a utilizar un timer el cual cada 500 ms ejecuta el comando flush().

Todo lo descripto anteriormente es necesario para enviar los mensajes, pero hace falta aún, una aplicación del lado del PC que "levante" todos los mensajes enviados por el mote y los despliegue en pantalla.

La aplicación que se encarga de esta tarea es la aplicación JAVA PrintfClient, la cual escucha en el puerto que se le indique y toma los mensajes que vengan formateados como se define en la clase JAVA PrintfMsg.

La clase PrintfMsg es autogenerada, y se crean sus métodos y variables a partir del archivo printf.h.

Para crear esta clase automáticamente se debe utilizar el comando java MIG [1], el cual se incluye en el Makefile de la aplicación.

Un ejemplo de llamada a la aplicación PrintfClient desde líneas de comandos es:

java PrintfClient -comm serial@COM5:115400

donde *COM5* es el puerto en donde se desea escuchar y *115400* es el baudrate específico de los motes tipo ***telosb***.

[1] http://www.tinyos.net/tinyos-1.x/doc/nesc/mig.html

Apéndice D

Plan de proyecto

D.1. Plan de proyecto inicial

Para la creación del plan de proyecto, se debieron definir las tareas por las cuales estaría conformado el proyecto en su totalidad.

Se crearon cuatro grupos de tareas y los mismos son los siguientes:

1. **Capacitación:** Este grupo comprende toda la etapa de estudio y familiarización con la tecnología a usar, esto agrupa al hardware, al software y a los protocolos.
2. **Diseño:** Como lo indica el nombre en este punto se busca con los conocimientos adquiridos lograr llegar a un diseño que cumpla con las condiciones de éxito ya prefijadas.
3. **Implementación:** Implica comenzar a llevar las ideas del diseño a la realidad, se comenzará con la programación, manipulación del hardware, etc.
4. **Validación:** Se realizarán las pruebas necesarias para corroborar que el sistema cumpla efectivamente con todos los requisitos.

D.1.1. Capacitación

D.1.1.1. Normas del Standard IEEE 802.15.4

Estudio del documento de la IEEE que define el Standard para la comunicación entre sensores inalámbricos(“ IEEE Standard for Information technology–Telecommunications and information exchange between systemsLocal and metropolitan area networks Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)”) - versión 2006 [4].

D.1.1.2. Elementos de diseño

Introducción al sistema operativo TinyOS, hardware (TmoteSky o similar), lenguaje de programación nesC. Esta etapa termina en conjunto con el seminario que se llevará a cabo al inicio del proyecto. Una vez completado éste, se deberá de haber comprendido cómo programar, compilar, crear un proyecto y hacer las pruebas pertinentes en el hardware.

D.1.1.3. Estudio de Implementaciones de capas superiores

Estudio de implementaciones de capas superiores y topologías de red existentes usadas en redes de sensores inalámbricos, tomando las encontradas como posibles soluciones, sin la necesidad en principio de elegir una definitiva, tarea que se dejará para la etapa de diseño. Incluimos la búsqueda de aplicaciones similares que puedan aportar a nuestra necesidad.

D.1.1.4. Estudio y prueba de implementaciones de capa MAC

Estudio de la implementación del grupo Hurray y prueba de funcionalidades ya existentes a nivel de capa MAC, las cuales son de todas las existentes, las que se acercan más al Standard deseado.

D.1.2. Diseño

D.1.2.1. Topología de red

Diseño de la topología de la red, el mismo se realizara basándose en el entregable referente al estudio de implementaciones de capa de red.

D.1.2.2. Protocolo de capa de red

Selección de protocolo de capa de red. En base a los estudios realizados a nivel de capa de red, se deberá seleccionar la implementación más adecuada para el proyecto, la elección del protocolo de ruteo, y además completar todas las funcionalidades que no se encuentren implementadas o que sea necesaria su modificación para adaptarse a los requerimientos.

D.1.2.3. Casos de Uso

Previo a la implementación de estos protocolos, y ya interiorizados en el tema, se describirán en detalle los casos de uso para la futura implementación de las funcionalidades listadas en la primera etapa. Estos casos de uso se especificarán para cada una de las entidades de la red: Mac, Red y Aplicación

D.1.3. Implementación

D.1.3.1. Implementación de la red

Programación (utilizando lenguaje nesC) de nuestra implementación. En una primera etapa implementaremos la capa de enlace (ajustada al Standard 802.15.4 de la IEEE que es una de nuestras restricciones) para la cual disponemos de gran parte del código, que deberemos entender en profundidad para hacer las modificaciones necesarias con el fin de adaptarlo a nuestros requerimientos.

D.1.3.2. Programación de la capa de red

Con las funcionalidades y características ya definidas nos enfocaremos en el desarrollo de esta capa.

D.1.3.3. Integración de las implementaciones antes mencionadas

Esta es una etapa que se asume se llevará a cabo en paralelo con las dos anteriores. Consiste en adaptar las funcionalidades que una capa provee a la otra, y verificar que estas funcionalidades cumplan con los requerimientos de la capa superior.

D.1.3.4. Desarrollo de la aplicación

Interfaz entre el nodo sumidero, en donde se colectan los datos de toda la red, y el PC. Se deberá de implementar esta interfaz acorde a especificaciones definidas por los tutores en un futuro. Nos podremos basar en posibles aplicaciones ya elaboradas en algún proyecto similar que habremos buscado en la etapa de capacitación.

D.1.3.5. Instalación de la implementación

Luego de que las anteriores etapas fueron finalizadas, y se realizo el compilado de todo el código, se procederá a instalar el software en los diferentes elementos de nuestro sistema.

D.1.3.6. Medición del consumo

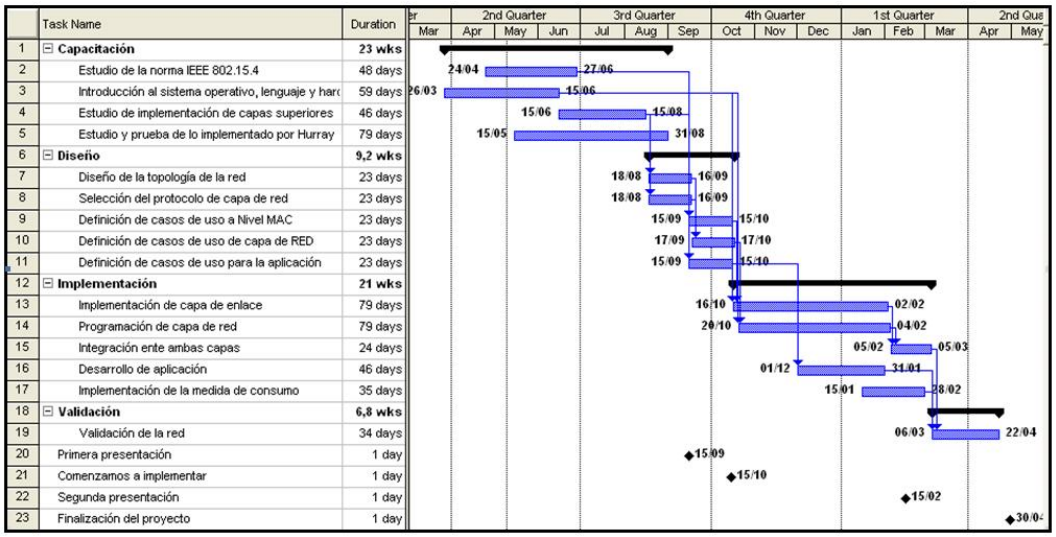
Implementación de la modalidad de medición de consumo.

D.1.4. Integración y Validación de la red

D.1.4.1. Validación de los criterios de éxito y aceptación

Aquí se utilizaran diferentes métodos para probar el cumplimiento de cada uno de los criterios de éxito. Los mismos serán: prueba de campo, donde se verificara el correcto funcionamiento de la comunicación entre los nodos; medición del consumo, se usara el método de medida de consumo pensado en la etapa anterior.

Para las tareas mencionadas, se definió un cronograma para las mismas, concluyendo en el siguiente diagrama de Gantt:



	Task Name	Duration
1	**Capacitación**	**23 wks**
2	Estudio de la norma IEEE 802.15.4	48 days
3	Introducción al sistema operativo, lenguaje y harc	59 days
4	Estudio de implementación de capas superiores	46 days
5	Estudio y prueba de lo implementado por Hurray	79 days
6	**Diseño**	**9,2 wks**
7	Diseño de la topología de la red	23 days
8	Selección del protocolo de capa de red	23 days
9	Definición de casos de uso a Nivel MAC	23 days
10	Definición de casos de uso de capa de RED	23 days
11	Definición de casos de uso para la aplicación	23 days
12	**Implementación**	**21 wks**
13	Implementación de capa de enlace	79 days
14	Programación de capa de red	79 days
15	Integración ente ambas capas	24 days
16	Desarrollo de aplicación	46 days
17	Implementación de la medida de consumo	35 days
18	**Validación**	**6,8 wks**
19	Validación de la red	34 days
20	Primera presentación	1 day
21	Comenzamos a implementar	1 day
22	Segunda presentación	1 day
23	Finalización del proyecto	1 day

Figura D.1: Diagrama de Gantt original.

A cada tarea se le asignó un responsable y se estimó la cantidad de horas hombre que consumiría cada una de ellas. En la siguiente tabla se detallan estos valores:

Tarea	Fecha de inicio	Fecha de finalización	Horas hombre	Recurso
Capacitación				
Estudio del Standard *IEEE 802.15.4*	24-04	30-06	154	Todos
Introducción al sistema operativo, lenguaje y hardware	25-03	15-06	154	Todos
Estudio de implementación de capas superiores	16-06	14-08	154	Todos
Estudio y prueba de lo implementado por Hurray	01-07	01-09	154	Todos
Diseño				
Diseño de la topología de red	15-08	15-09	26	L.P.
Selección del protocolo de capa de red	15-08	15-09	52	P.M. y F.P.
Definición de casos de uso a nivel MAC	15-09	14-10	18	L.P.
Definición de casos de uso de capa de Red	16-09	15-10	18	F.P.
Definición de casos de uso para aplicación	15-09	15-10	18	P.M.
Implementación				
Implementación de capa de enlace	15-10	12-02	256	Todos
Programación de capa de red	16-10	13-02	170	L.P. y F.P.
Integración entre ambas capas	15-02	16-03	43	P.M.
Desarrollo de aplicación	15-12	12-02	170	L.P. y P.M.
Implementación de la medida de consumo	31-01	11-03	37	P.M.
Validación				
Validación de la red	17-03	14-04	154	Todos

Tabla D.1: Tabla de asignación de tareas.

Considerando la tabla anterior, el total de horas hombre que se estimó que consumiría el proyecto era 1578 hs; un promedio de 526 hs por integrante.

D.2. Real desarrollo de tareas

En esta sección analizaremos a modo general la asignación inicial de tareas, el cumplimiento de los tiempos y la dedicación horaria.

Se pudo observar a lo largo del transcurso del proyecto, que las tareas fueron adecuadamente definidas con una distribución temporal que se adecuó al proyecto en la fase inicial, pero en la fase final se notó un leve desfasaje. El momento que separó estos dos períodos fue después de la primera presentación que se llevó a cabo el 15 de setiembre de 2008. Luego de este hito, se procedió a hacer las pruebas pertinentes en laboratorio para verificar el correcto funcionamiento del código que debería cumplir con el standard IEE 802.15.4. Se constató que no cumplía con el requerimiento de apagado y encendido del radiotransmisor, lo que provocó que se tuviera que dedicar más tiempo a la tarea de "*Implementación de capa de enlace*" y tuvo como consecuencia el retardo de las tareas que dependían directamente de este resultado.

Debido a la intensa labor en la capa de enlace por parte de todos los integrantes luego de la resultados negativos en laboratorio, se postergaron las tareas que se preveía se desarrollaran en paralelo: "*Programación de capa de red*" y "*Programación de aplicación*". Las tareas a partir de la fecha se retrasaron entre uno y dos meses aproximadamente. Para disminuir el impacto que esto podría provocar en nuestro diagrama de Gantt, se procedió a intensificar la dedicación horaria obteniendo buenos resultados, ya que las posteriores tareas se vieron perjudicadas pero de manera insignificante.

En rasgos generales la tarea que insumió mayor dedicación horaria fue "*Implementación de la capa de enlace*", la cual aproximadamente fue el triple pero básicamente se cumplió con el cronograma en tiempo y forma.

Se subestimó la tarea de documentación que no se definió, pero en cuanto a la tarea relacionada con la "*Implementación de la medida de consumo*", que en realidad no se debió incluir ya que se suponía que se contaba con el dispositivo adecuado para dicha medida pero no se logró.

Apéndice E

Contenido del CD

E.1. Estructura del CD

A continuación se presenta el árbol de directorios correspondiente al CD que se adjunta con la presente documentación.

```
/
├── RSIS.pdf
├── Archivos fuente del software de HURRAY
├── Documentación de HURRAY
└── Archivos fuente del software de RSIS
```

E.2. Descripción de los archivos

En esta sección se describe el contenido de cada uno de los archivos contenidos en las diferentes carpetas del CD.

Viendo la estructura del CD en la sección anterior se puede ver que en el primer nivel (junto con los directorios) se encuentra el archivo que contiene el presente documento, llevando el nombre de `RSIS.pdf`.

Se incluye también la carpeta `Archivos fuente del software de HURRAY` en donde se encuentra la estructura original del código implementado por el grupo HURRAY, el cual se tomó como punto de partida para el presente proyecto. Dentro de dicha carpeta está `hurray2x`, carpeta dentro de la cual se ubican las carpetas `apps`, `support` y `tos`. Dentro de apps se encuentran las aplicaciones: SimpleRoutingExample, GTSManagementExample, DataSendExample y AssociationExample. Para cada ejemplo existe una carpeta con los respectivos archivos ".h", configuración e implementación. Los archivos con extensión .h contienen las definiciones de las estructuras y constantes, mientras que el archivo de configuración contiene el cableado y el de implementación todo el código que implementa las funcionalidades.

La otra carpeta que resulta relevante para este proyecto es `tos`, la cual contiene los archivos relacionados con los componentes hardware principales como son CC2420 (transceiver) y MSP430 (microprocesador) y el código que implementa el Standard IEEE 802.15.4. además de archivos que son propios del sistema operativo TinyOS. La carpeta `chips` es

la que contiene los archivos de los chips mientras que la `ieee802154` contiene los archivos que implementan el Standard.

Por otra parte tenemos la carpeta `Archivos fuente del software de RSIS`, la cual contiene una estructura de carpetas igual a `Archivos fuente del software de HURRAY` pero con las modificaciones hechas durante el desarrollo de este proyecto. Por ello es que existen diferencias principalmente en las carpetas de `apps`, en donde se encuentran los archivos que implementan las capas superiores (aplicación y red) de la red, aquí se encuentran: `Interfaces`, `Red`, `Sumidero` y `NoSumidero`. También existen muchas diferencias en los archivos contenidos en la carpeta `ieee802154` y se agregó una carpeta llamada `rsis` dentro de la estructura: `hurray2x/support/sdk/java/net/tinyos` en donde se implementa la aplicación de la PC que se encarga de desplegar los datos en pantalla y enviar mensajes por el USB.

E.3. Requerimientos del sistema

- **Sistema operativo:** Windows 98/XP o superior
- **Memoria RAM (mínima):** 64 MB
- **Resolución de pantalla:** 800 x 600 o superior
- **Velocidad de lectora de CD-ROM:** 16x o superior
- **Visor de PDF:** Acrobat Reader 5 o superior

Bibliografía

[1] Jesús Serna Sanchis, Trabajo de ampliación de redes de sensores inalámbricas

[2] Hurray es un grupo de investigadores en arquitecturas de sistemas computacionales para tiempo real, parte de su desarrollo fue utilizado como punto de partida en este trabajo
http://www.hurray.isep.ipp.pt/

[3] Anis Koubâa, Mário Alves y Eduardo Tovars; Lower Protocol Layers for Wireless Sensor Networks: A Survey
http://www.hurray.isep.ipp.pt/asp/show_doc2.asp?id=229

[4] Standard de la IEEE en el cual se centra este trabajo:
Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks WPANs)
http://standards.ieee.org/getieee802/802.15.html

[5] Documentación del código de partida, An IEEE 802.15.4 protocol implementation in nesCTinyOS Reference Guide v1.2 se encuentra disponible en :
http://www.hurray.isep.ipp.pt/privfiles/tr-hurray-061106.pdf

[6] Anis Koubaa, André Cunha y Mário Alves; A Time Division Beacon Scheduling Mechanism for IEEE 802.15.4/Zigbee Cluster-Tree Wireless Sensor Networks
http://www.cister.isep.ipp.pt/asp/show_doc2.asp?id=366

[7] Zigbee es un protocolo que utiliza la interfaz IEEE 802.15.4 y especifica una capa superior de red y aplicación www.zigbee.org/
http://es.wikipedia.org/wiki/ZigBee_(especificaci%C3%B3n)

[8] Hoja de datos TmoteSky:
http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf

[9] Fabricante del chip CC2420, actualmente Texas Instruments compró la firma.
Hoja de datos del componente principal:
http://www.alldatasheet.com/datasheet-pdf/pdf/125399/ETC/CC2420.html

[10] Hoja de datos del microcontrolador
www.datasheets.org.uk/datasheets/MSP4301611.html

[11] Sitio principal del sistema operativo http://www.tinyos.net/

[12] Philip Lewis, TinyOS programming
http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf

[13] Pasos para la instalación de tinyOS
http://docs.tinyos.net/index.php/Getting_started

[14] Leonardo Barboni, Maurizio Valle, "Experimental Analysis of Wireless Sensor Nodes Current Consumption", sensorcomm, pp.401-406, 2008 Second International Conference on Sensor Technologies and Applications, 2008
http://www2.computer.org/portal/web/csdl/doi/10.1109/SENSORCOMM.2008.14