GRADO EN INGENIERIA INFORMATICA
Ingeniería de Computadores

IMPLEMENTACIÓN DE UNA APLICACIÓN DE SEGUIMIENTO DE AVES UTILIZANDO MOTAS IRIS CON EL SISTEMA OPERATIVO TINYOS

Alumno: Sergio Pardo Hervias

Directora: Iratxe Soraluze

Director: Luis Gardeazabal

Trabajo de Fin de Grado, febrero de 2015

RESUMEN

La Sociedad de Ciencias Aranzadi requiere de un equipo muy costoso para realizar uno de sus estudios sobre la gaviota patiamarilla del Cantábrico, y ha propuesto a la facultad de informática de Donostia-San Sebastián la creación de un dispositivo para el mismo fin, por un precio menor.

Los dispositivos elegidos para la realización del seguimiento son las motas IRIS a las que se incorpora un GPS para poder registrar la localización de las aves.

El objetivo de este proyecto consiste en implementar en NesC parte del código de una aplicación de seguimiento de aves. Además, se requiere diseñar un sistema de actualización del software que ejecutan las motas de forma inalámbrica mediante el sistema de radio.

Este documento analiza la implementación del software para el seguimiento de las gaviotas patiamarillas, asi como el análisis previo del sistema operativo utilizado, TinyOS y las funcionalidades que éste nos ofrece.

INDICE

1 INTRODUCCIÓN 7

1.1 Motivación 7
1.2 Descripción 7
1.3 Objetivos 8
1.4 Plan de trabajo 9
1.5 Planificación del proyecto 10
1.6 Organización del documento 10

2 TECNOLOGÍA UTILIZADA PARA LA IMPLEMENTACIÓN DEL SISTEMA REQUERIDO 11

2.1 Redes de sensores inalámbricas y las motas IRIS 11
2.2 Sistema operativo TinyOS y el lenguaje de programación NesC 12
2.3 Heramientas ofrecidas por TinyOs para el cumplimiento de las especificaciones del proyecto 13
2.3.1 Estructura de directorios de TinyOS 13
2.3.2 Herramientas para realizar pruebas 13

3 ACTUALIZACIÓN DE SOFTWARE DE FORMA REMOTA 15

3.1 Tosboot, el bootloader 15
3.2 La golden image y los volúmenes 16
3.3 Pasos a seguir para la reprogramación 16
3.4 Comprobación del funcionamiento 17
3.4.1 Prueba I (reprogramar varias veces con programas distintos) 17
3.4.2 Prueba II (reprogramar varias veces con programas distintos pero solo un volumen) 18
3.4.3 Prueba III (apagar la mota mientras se disemina) 18
3.4.4 Prueba IV (Apagar la mota mientras se reprograma) 19

4 OTROS COMPONENTES SOTFWARE 21

4.1 Lectura/escritura de datos en memoria Flash 21
4.1.1 Volúmenes 21
4.1.2 Componentes e interfaces 22
4.1.3 Desarrollo del programa de pruebas y comprobación del funcionamiento 23
4.2 Radio 25
4.2.1 Componentes e interfaces 25
4.2.2 Tipos de mensajes 26
4.2.3 Desarrollo del programa de pruebas y comprobación del funcionamiento 27

4.3 Medición de la posición ……………………………………………………...…. 28
4.3.1 Funcionamiento del GPS ……………………………………………….. 28
4.3.2 Datos recibidos del GPS ………………………………………………... 29
4.3.3 Componentes e interfaces ……………………………………………… 29
4.3.4 Desarrollo del programa de pruebas y comprobación del funcionamiento 30
4.4 Medición de la tensión de la batería ………………………………………… 30
4.4.1 Componentes e interfaces ……………………………………………… 30
4.4.2 Medidas y prueba del funcionamiento ………………………………… 31

5 DISEÑO DE LAS APLICACIONES ………………………..…………... 33
5.1 Diseño de la aplicación de la estación base …………………..…………….. 33
5.2 Diseño de la aplicación cliente ……………………………..……………. 34
5.2.1 Fuera del nido …………………………………………...……………. 34
5.2.2 Entrando al nido ……………………………………….…………... 35
5.2.3 Dentro del nido …………………………………………..…………… 36
5.2.4 Saliendo del nido ……………………………………………………… 37

6 IMPLEMENTACIÓN DE LAS APLICACIONES ……..………...……….. 39
6.1 Elementos comunes ………………………………….…………...…….. 39
6.1.1 Estructuras de datos ……………………….....…………...……. 39
6.1.2 Variables de entorno ……………………….…………...……...……. 40
6.2 Implementación de la aplicación base ……………...…………..……… 40
6.2.1 Makefile de la aplicación base …………………….…...………..….…. 40
6.2.2 volumes-at45db de la aplicación base ………………….……....…..…… 40
6.2.3 Archivo de cabecera de la aplicación base …………………….…....…… 41
6.2.4 Implementación de la configuración de la aplicación base ……………… 41
6.2.5 Implementación del módulo de la aplicación base ……………............ 42
6.3 Implementación de la aplicación cliente …………………………………... 43
6.3.1 Makefile de la aplicación cliente …………………………………….. 43
6.3.2 volumes-at45db de la aplicación cliente …………………………………… 43
6.3.3 Archivo de cabecera de la aplicación cliente ……………………………… 43
6.3.4 Implementación de la configuración de la aplicación cliente …………….. 44
6.3.5 Implementación del módulo de la aplicación cliente ……………………... 45

7 PRUEBA DE LAS APLICACIONES ……………………………..…………. 49

7.1 Preparación de las pruebas ……………………………………..…..…..….. 49
7.2 Prueba de la base ……………………..…………………………..……..…. 49
7.3 Prueba de la aplicación cliente ………………….…………………..…..….... 50

8 CONCLUSIONES Y TRABAJO A REALIZAR ……………………………. 53

BIBLIOGRAFÍA …………………………………………………...…………. 54

1 INTRODUCCIÓN

1.1 Motivación

La Sociedad de Ciencias de Aranzadi (http://www.aranzadi-zientziak.org/category/ornitologia) desea estudiar los hábitos de la gaviota patiamarilla del Cantábrico. Este departamento realiza sus estudios en gran parte sobre las tres colonias ubicadas en la provincia de Guipúzcoa, dos en San Sebastián y una en Getaria. Su objetivo actual es monitorizar el comportamiento diario de estas aves, para identificar sus rutas de vuelo principales y uso del hábitat (áreas de descanso y de alimentación, tiempos de estancia, etc.). El problema reside en que los dispositivos electrónicos y servicios comerciales disponibles para la monitorización de la avifauna que se requieren para este estudio tienen un coste muy elevado.

Bajo la premisa de conseguir dichos dispositivos a un coste menor, la Sociedad de Ciencias Aranzadi ha propuesto un proyecto en colaboración con la Dep. de Arquitectura y Tecnología de Computadores de la Fac. de Informática de la UPV/EHU de Donostia-San Sebastián para el desarrollo y la construcción de dichos dispositivos así como del software necesario para su funcionamiento.

1.2 Descripción

Existe un trabajo realizado por otro alumno de la Facultad de Informática de la UPV/EHU, Rubén García Hernández, que precede a este proyecto. En dicho trabajo, se realiza un estudio de la tecnología existente para el seguimiento de aves y un diseño inicial de una aplicación que resuelva las necesidades de este proyecto.

Según el trabajo de este alumno, la tecnología a utilizar es una red de sensores inalámbrica (*Wireless Sensor Networks* o WSN). Estas redes están compuestas por varios nodos, que reciben el nombre de motas, provistos de sensores que recogen datos del exterior y que mediante las comunicaciones entre ellos consiguen realizar una tarea común. Estos nodos o dispositivos tienen un tamaño muy reducido, lo que implica ciertas limitaciones de memoria y procesamiento. Además, para este proyecto existe otra gran limitación, la alimentación, que al utilizar paneles solares, se tiene que controlar la carga de la batería al realizar ciertas operaciones.

Otra cosa a tener en cuenta es que estos dispositivos, una vez montados en las aves, no podrán volver a ser manipulados manualmente, por lo que si se desea reprogramar el dispositivo con otra versión de software, tendrá que hacerse de forma remota. Para esto se necesitarán dos aplicaciones distintas a implementar, por un lado una para los dispositivos que recogen la posición de las gaviotas y otra para la estación base que recoja esos datos y se encargue de enviar los comandos pertinentes para reprogramar los demás nodos.

La estación base se colocará cerca de la zona de anidamiento de las aves, de forma que la descarga de los datos recogidos por los nodos instalados en la motas se realizará cuando estén en el radio de acción de la base. Esto, según se estima, será durante las horas nocturnas, mientras que por el día se recogerán los datos deseados. En el caso de este proyecto, los nodos o motas de la WSN no se comunicaran entre ellos, únicamente con la estación base.

1.3 Objetivos

La meta de este proyecto es implementar parcialmente un software para el seguimiento de la gaviota patiamarilla adecuado a las necesidades de la Sociedad de Ciencias de Aranzadi. Esta implementación será una primera versión que se irá completando según los datos obtenidos y otras especificaciones futuras.

El enfoque propuesto se basa en la utilización de motas de la familia IRIS (de MEMSIC-Crossbow) con el sistema operativo TinyOS y de algunas de las aplicaciones que trae implementadas este sistema operativo. Estas aplicaciones o componentes software son incluidos en el código fuente de TinyOS para crear la aplicación requerida. En el apartado 2.1 se ven en detalle los componentes necesarios. Bajo este enfoque, el proyecto completo requiere del desarrollo de dos elementos, varios nodos equipados con un sistema de localización global, que son los que portarán las aves, y una estación base, encargada de descargar el historial de las posiciones del nodo que porta el ave y de diseminar y reprogramar los nodos con las posibles versiones que en un futuro se puedan implementar.

El objetivo principal de este documento es analizar la implementación de las dos aplicaciones, una para los nodos puestos en las gaviotas y la otra para la base, necesarias para poder hacer el seguimiento de aves.

A continuación se listan los sub-objetivos que se han requerido realizar para cumplir con el objetivo general.

- Llevar a cabo un estudio de las herramientas a utilizar, con el fin de aprender el funcionamiento de TinyOS y del lenguaje de programación NesC con los que se realiza la implementación de las aplicaciones para este proyecto.

- Estudiar y probar el sistema de actualización de software por medio de la radio para poder actualizar el software en cualquier momento sin interceptar las aves.

- Determinar los componentes software que se utilizarán en las aplicaciones a desarrolar, probando cada uno por separado y posteriormente juntarlos para la creación de las dos aplicaciones.

- Crear una serie de pruebas que simulen el entorno en el que las motas actuarán, de forma que se pueda determinar su correcto funcionamiento.

1.4 Plan de trabajo

El trabajo realizado en este proyecto está dividido en cuatro tareas. Cada tarea se explica indicando los objetivos y la descripción del procedimiento para cumplirlos.

1.4.1 Tarea 1. Estudio de las tecnologías a utilizar

Objetivos. Se pretende conocer en detalle la tecnología elegida y conocer qué elementos proporciona que puedan ser utilizados para el software a implementar.

Descripción. El estudio afecta fundamentalmente a las herramientas y el código fuente ofrecido por el sistema operativo TinyOS. Es decir, cómo se puede aprovechar el código proporcionado por el sistema operativo para el beneficio de la aplicación requerida.

1.4.2 Tarea 2. Exploración de un sistema de actualización de software de forma remota

Objetivos. Se pretende analizar en detalle la herramienta que trae TinyOS implementada para la actualización de software por medio de la radio.

Descripción. TinyOS proporciona una aplicación software que permite transmitir una aplicación a las motas y programar las motas con dicha aplicación. El estudio de esta aplicación se realizará para probar su correcto funcionamiento, simulando varias situaciones que se pueden dar cuando el dispositivo esté montado en las aves.

1.4.3 Tarea 3. Exploración de otros componentes provistos por el sistema operativo TinyOS

Objetivos. Se pretende analizar los componentes principales requeridos para implementar las dos aplicaciones, realizando pruebas para determinar si cumplen las especificaciones.

Descripción. Los componentes ya implementados están creados para un propósito específico, de forma que se tendrá que analizar si funcionan de la misma manera para la aplicación a programar que para las que fueron creadas. Para ello se crearán programas de prueba sobre los que se realizarán varios tests.

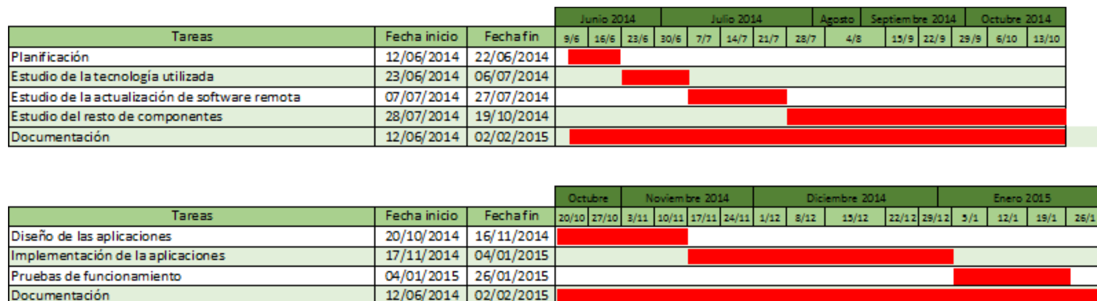
1.4.4 Tarea 4. Creación de las aplicaciones y prueba de su correcto funcionamiento

Objetivos. Se pretende diseñar una lógica de estados que describe el funcionamiento de la aplicación y con la que se optimice el uso de los recursos disponibles.

Descripción. Se requiere la creación de una lógica para la unión de todos los componentes para que se cumplan los requerimientos. Además, se documentarán las pruebas a realizar para determinar el correcto funcionamiento de la aplicación, junto a los resultados de dichas pruebas.

1.5 Planificación del proyecto

En este apartado se analiza cuando se han llevado a cabo las tareas a realizar a lo largo del tiempo, para lo que se hará uso del diagrama de Gantt. Para realizar una planificación más precisa del proyecto, a las tareas comentadas en el apartado anterior se le añaden otras dos, la planificación y la documentación que recoge los avances realizados a lo largo del proyecto. Además, la realización de las aplicaciones se desglosa en tres sub-tareas, diseño, implementación y pruebas de funcionamiento.



Tareas	Fecha inicio	Fecha fin
Planificación	12/06/2014	22/06/2014
Estudio de la tecnología utilizada	23/06/2014	06/07/2014
Estudio de la actualización de software remota	07/07/2014	27/07/2014
Estudio del resto de componentes	28/07/2014	19/10/2014
Documentación	12/06/2014	02/02/2015

Tareas	Fecha inicio	Fecha fin
Diseño de las aplicaciones	20/10/2014	16/11/2014
Implementación de la aplicaciones	17/11/2014	04/01/2015
Pruebas de funcionamiento	04/01/2015	26/01/2015
Documentación	12/06/2014	02/02/2015

Figura 1. Diagrama de Gantt.

A continuación, se muestra una tabla con las horas necesitadas para cada una de las tareas.

Tareas a realizar	Horas dedicadas
Planificación	10 h
Estudio de la tecnología utilizada	20 h
Estudio de la actualización de software remota	30 h
Estudio del resto de componentes	65 h
Diseño de las aplicaciones	40 h
Implementación de la aplicaciones	60 h
Pruebas de funcionamiento	30 h
Documentación	50 h
Total	305 h

Tabla 1. Horas dedicadas a cada tarea.

1.6 Organización del documento

El resto del documento se organiza de la siguiente forma. En el **segundo** capítulo, se realiza una introducción a las motas que se utilizan a lo largo de proyecto. Además, se habla del sistema operativo que utilizan y del contenido que nos ofrece dicho sistema operativo para cumplir las especificaciones técnicas y de funcionamiento que necesita la Sociedad de Ciencias Aranzadi. El **tercero**, analiza el comportamiento de la aplicación más crítica del proyecto, el encargado de realizar la actualización de software de forma remota, cerciorándose de que ésta funcione según lo esperado. El **cuarto** capítulo, contiene un análisis de los demás componentes software que se utilizan en la implementación de las aplicaciones. Este análisis se realiza sobre cada componente individualmente, analizando su funcionamiento y documentando las pruebas realizadas para cerciorarse de que dicho componente funciona tal y como se espera de él. En el **quinto** capítulo, se realiza el diseño de las dos aplicaciones a realizar, la de la base y la de los nodos. El siguiente capítulo, el **sexto**, recoge la implementación de las dos aplicaciones mencionadas. El **séptimo** capítulo, detalla las pruebas realizadas para comprobar si el sistema funciona como se desea. En el **octavo** se detallan las conclusiones y el trabajo futuro.

2 Tecnología utilizada para la implementación del sistema requerido

Este capítulo se centra en realizar una investigación de las tecnologías que se usan para lograr los objetivos de este proyecto. Primeramente se estudia la plataforma en la que se desarrolla la implementación a desarrollar, tanto los dispositivos hardware como el sistema operativo y el lenguaje de programación sobre el que se trabajará. Una vez realizada la introducción del hardware y del software, se analiza más específicamente las herramientas que brinda el sistema operativo TinyOS para lograr desarrollar una aplicación para la localización de aves.

2.1 Redes de sensores inalámbricas y las motas IRIS

Una red de sensores inalámbrica (Wireless Sensor Networks, WSN) es una red de dispositivos muy pequeños, comúnmente conocidos por el nombre de “nodos” o “motas”, que colaboran en una tarea común. Estas redes son desatendidas, es decir, funcionan de forma autónoma, sin necesidad de intervención humana. Cada una de estas motas tiene varios sensores o la posibilidad de conectarle sensores de forma que puedan recoger datos del entorno, además de la capacidad de procesar, y transmitir dichos datos. Estos datos serán recogidos por una estación base, la cual es la encargada de proporcionar la información a otro dispositivo como puede ser un ordenador o a otra red, por ejemplo, Internet.

Las motas tienen unos recursos físicos muy limitados, debido a las metas de tamaño pequeño, bajo costo y bajo consumo de energía. Es por lo que ha sido necesario un estudio de mercado para determinar qué marca y modelo son los más adecuados para el desarrollo del proyecto. Como ya se ha comentado con anterioridad, este estudio ha sido realizado por otro alumno de la facultad, que analizo los dispositivos que hay en el mercado para el seguimiento de aves y los distintos modelos de motas, destacando sus ventajas y desventajas.

Después de realizar dicho estudio, se ha decidido usar las motas IRIS como nodos para la red de sensores. Los aspectos por los que se ha elegido este dispositivo son su ligereza, pesando únicamente 18 g (sin batería), su transmisor de radio, que no consume mucho y tiene una gran sensibilidad en la recepción y la ranura de 51 pines, que facilita el añadir componentes. Las principales especificaciones de la mota se muestran en la Tabla 2.

Especificaciones del procesador			Especificaciones técnicas		Especificaciones de la radio	
Procesador	Chip	Atmel ATMega 1281	Rango de funcionamiento	2,7 V - 3,3 V	Rango de frecuencia	2405 MHz-2480 MHz
	Frecuencia	8 MHz	Dimensiones	3.2x5.8x0.7 cm	Data rate	250 Kbit/s
Memoria	Flash de programa	128 KB	Peso	18 g (sin baterias)	Rango en exterior	> 300 m
	Flash de datos	512 KB	Interfaz de usuario	3 LEDs	Rango en interior	> 50 m
	RAM	8 KB	Conector de expansión	51 pin		
	EEPROM	4 KB				
Consumo	Activo	8 mA				
	Sleep	8 µA				

Tabla 2. Especificaciones de la mota IRIS.

2.2 Sistema operativo TinyOS y el lenguaje de programación NesC

TinyOS tiene un modelo de ejecución basado en eventos y un modelo de programación basado en componentes, codificado por el lenguaje nesC, un dialecto de C. TinyOS no es un sistema operativo en el sentido tradicional; sino que se trata de un marco de programación para sistemas embebidos y un conjunto de componentes que permiten la construcción de un "sistema operativo específico" para cada aplicación.

Un programa de TinyOS es un grafo de componentes, cada uno de los cuales es una entidad independiente con capacidad computacional, que expone una o más interfaces. Estas interfaces son bidireccionales y contienen tres abstracciones computacionales: comandos, eventos y tareas. Un comando es una función que esta implementada en el componente proveedor de una interfaz, mientras que un evento es una función que se implementa en el componente que usa dicho comando. Las tareas se utilizan para expresar concurrencia entre componentes.

NesC tiene dos tipos de componentes: módulos y configuraciones. Los módulos proporcionan código y están escritos en un dialecto de C con extensiones para llamadas e implementación de los comandos y eventos. Un módulo declara variables de estado privadas y buffers de datos, a los que sólo se puede hacer referencia desde la componente. Las configuraciones, a su vez, se utilizan para conectar componentes entre sí, conectando interfaces utilizadas por los componentes con interfaces proporcionadas por otros. Todas las aplicaciones TinyOS están descritas por una configuración de nivel superior que relaciona todos los componentes utilizados.

En la Figura 2 se muestra como es el funcionamiento de TinyOS con una aplicación de ejemplo. En el componente llamado *AppC*, se definen todos las conexiones entre las interfaces de todos los demás componentes, que son las flechas entre cada interfaz. El componente *MainC* es el encargado de iniciar algunos componentes. *AppM* es el módulo principal de esta aplicación de ejemplo, en ella se implementan los comandos y eventos que se usan. La interfaz Leds que provee el componente *LedC*, únicamente proporciona comandos; mientras que *Timer*, del componente *TimerC*, no solo implementa comandos sino que también señala eventos.

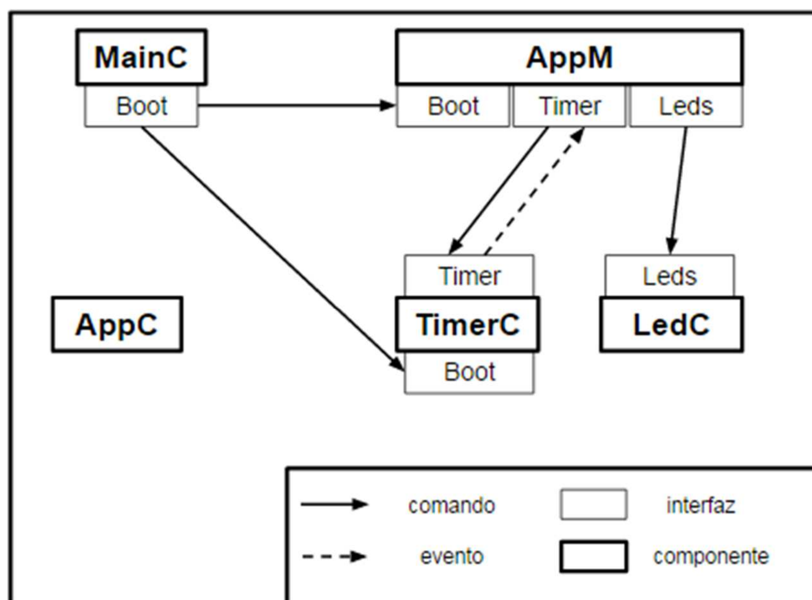


Figura 2. Relación entre componentes de una aplicación ejemplo

2.3 Herramientas ofrecidas por TinyOS para el cumplimiento de las especificaciones del proyecto

Como se ha comentado en el capítulo de introducción, el enfoque que se sigue se basa en utilizar parte de los componentes y aplicaciones implementadas por TinyOS y en caso de ser necesario, modificarlo para las necesidades de la aplicación a realizar. Además, el sistema operativo ofrece varias funcionalidades que hacen que en las pruebas que se realicen sea más fácil ver los resultados.

2.3.1 Estructura de directorios de TinyOS

TinyOS está organizado en varias carpetas, entre las que se encuentran *apps, licenses, tools, tos…* Algunas de estas traen código ya implementado del cual se puede echar mano en la programación de la aplicación que se quiera desarrollar.

Para ser más concretos, el directorio de apps, contiene varias aplicaciones de ejemplo que hacen uso de varios componentes, de forma que se pueden utilizar como guía de usuario o como punto de inicio de otra aplicación más compleja. En el directorio de *tos*, por su parte, están los archivos correspondientes a los componentes que con sus interfaces facilitan la tarea al programador. Y es que cuando se desea hacer uso de los sensores, leds o timers, entre otros, estos componentes nos proveen de comandos que nos darán los resultados sin tener que meternos en la implementación de los mismos.

Por otro lado, está la carpeta de *tools*, que proporciona varias herramientas para hacer ciertas tareas más fáciles a los programadores, de las cuales se habla en el siguiente apartado.

2.3.2 Herramientas para realizar pruebas

A pesar de que los componentes de TinyOS están implementados en NesC, este sistema operativo incorpora varias herramientas que pueden estar escritas en otros lenguajes, como puede ser Java o Python. A lo largo de este proyecto, hay dos de estas herramientas que son muy utilizadas, las cuales están programadas con Java. La primera es *PrintfClien*t, que como su nombre indica, es un cliente que imprime por pantalla los mensajes que las aplicaciones requieran. Entre los componentes que provee TinyOS tenemos el de *PrintfC*, que implementa varias funciones entre las que tenemos la conocida *printf.* El cliente de *Printf* será el encargado de visualizar los mensajes creados por esta función.

Otra herramienta muy utilizada es Listen. Gracias a ella se pueden ver todos los mensajes enviados y recibidos por una mota que está conectada al ordenador. Es muy útil ya que esto nos permite ver toda la comunicación que tiene la mota con el resto de la red.

3 Actualización de software de forma remota

Una vez instaladas las motas a las gaviotas, no se volverán a tocar dichas motas y si se requiriese una actualización o corrección del código del programa, no se dispondrá físicamente de los dispositivos. Es por ello que se necesita un mecanismo que permita programar el dispositivo con las nuevas versiones de software de forma remota y es donde TinyOS implementa una aplicación llamada Deluge.

Al ser tan importante, hay que asegurar que la versión del programa que se esté corriendo cuando se instalen las motas a las aves, sea compatible con Deluge. Además, hay que ver qué características ofrece por defecto Deluge en lo que a tratamiento de errores (desconexión, interferencias en la radio...) se refiere y ver qué soluciones se les puede dar, si no son tratadas.

A continuación se comentan varios puntos para entender el correcto funcionamiento de esta aplicación, que son TOSBoot, los volúmenes con las imágenes de programa y las funcionalidades de TinyOS para conseguir la programación del dispositivo de forma remota. Asimismo, se hace un exhaustivo banco de pruebas para comprobar que realmente cumple las expectativas de funcionamiento.

3.1 TOSBoot, el bootloader

La aplicación de TinyOS que se encarga del arranque de la mota, TOSBoot, proporciona un conjunto de mecanismos necesarios para programar el microcontrolador con una imagen del programa almacenado. TOSBoot es un programa independiente que se ejecuta cada vez que el microcontrolador sale del estado de reset. Al ser una aplicación autónoma significa que cualquier estado no volátil dejado atrás por las aplicaciones de TinyOS no afecta a la ejecución de TOSBoot. La instalación de TOSBoot en un nodo sólo se produce a través de una conexión física y nunca través de la red.

Los parámetros se pasan a TOSBoot a través de memoria no volátil. El uso de este tipo de memoria asegura un correcto funcionamiento incluso durante las interrupciones inesperadas o fallos de energía durante el proceso de inicio. Los parámetros incluyen el número de interrupciones consecutivas durante el arranque y la ubicación del código binario en la flash externa con el que programar el microcontrolador.

Si se solicita la reprogramación, TOSBoot borrará la flash de programa y escribirá el nuevo binario en la misma. Al terminar, el bit de programación se resetea y TOSBoot salta a la primera instrucción de la aplicación. Si no se solicita la programación, sólo se ejecuta el salto a la aplicación.

3.2 La Golden Image y los volúmenes

La Golden Image es una aplicación de TinyOS completa y de confianza que se almacena en una ubicación de sólo lectura en la flash externa. Utilizando el hardware de protección contra escritura, la Golden Image sólo se puede instalar a través de una conexión física. El propósito de la Golden Image es tener un mecanismo para poder traer el nodo a un estado recuperable en caso de que sucediese un error. Por lo tanto, la Golden Image proporciona funcionalidad mínima para apoyar Deluge, permitiendo al nodo la reprogramación inalámbrica.

Este mecanismo es útil cuando, por ejemplo, ocurre un error en la lectura del programa binario mientras escribía para programar la flash. En este tipo de error, el nodo se reinicia y se realiza otro intento de programar. Cuando se hacen varios intentos, la Golden Image se carga, haciendo posible inyectar un nuevo programa sin la necesidad de una conexión física con un ordenador.

Aparte de tener esta imagen para poder volver a un estado recuperable, necesita tener espacio para almacenar las imágenes que la base le transmite. Por ello, se necesita reservar parte de la memoria flash y esto se hace definiendo volúmenes, como se verá en el apartado 4.1.1. Deluge requiere un volumen para la Golden Image y tres para diversas versiones de software.

De todas formas, como en este proyecto solo se utiliza uno de esos tres volúmenes, a los otros dos volúmenes que se tienen que definir únicamente se le asignará el tamaño mínimo posible, ya que se necesita el espacio máximo para poder guardar los datos de localización del GPS.

3.3 Pasos a seguir para la reprogramación

Una vez programadas las motas con algún programa que tenga Deluge habilitado, si se desean reprogramar, será necesario conectar otra mota a un ordenador para que haga de base. Esta base será la encargada de mandar los comandos necesarios para que las motas inicien la reprogramación. Además, diseminará el código binario de las nuevas versiones a las demás motas que estén a su alcance.

La facilidad de uso es uno de los objetivos principales, tratando de hacer que la gestión de la programación por la red sea tan simple y transparente como sea posible. Una simple línea de comandos de Python Toolchain es utilizada para todas las operaciones. La cadena de herramientas de Python puede ser utilizada para hacer ping a la red y recuperar información acerca de las imágenes que están en la red, inyectar nuevas imágenes, e inyectar comandos para programar la red con una nueva imagen. El script para interactuar con las motas está en la carpeta tinyos-2.x/tools/tinyos/misc y se llama tos-deluge y también habrá que especificarle el puerto por el que realizará la comunicación.

A pesar de tener varios parámetros para este script, sólo se analizan los tres más importantes y los que se usarán en este proyecto. Para empezar, está la opción -i que pasándole como parámetro la imagen creada al compilar la aplicación, guardará (inyectará) dicha imagen en el volumen que especifiquemos de la mota base. Un ejemplo de un comando completo es el siguiente:

```
> tos-deluge serial@/dev/ttyS1:57600 -i 1 tos_image.xml
```

Por otro lado, está el comando -dr, el cual hará que la base disemine la imagen guardada en el volumen especificado a las motas y les mande que se reprogramen con dicha imagen. Este comando estará activo en todo momento, de forma que si una mota entra en alcance, se "infectará" también. Por ello se necesita el comando -s, que hace que pare la "infección". Siguiendo con el ejemplo, estos serían los comandos:

> > tos-deluge serial@/dev/ttyS1:57600 -dr 1
> > tos-deluge serial@/dev/ttyS1:57600 -s

3.4 Comprobación del funcionamiento

Para conocer si Deluge actúa como se necesita, se han creado dos programas nuevos, llamados prueba1 y prueba2. Estos no son más que dos modificaciones de la aplicación Blink incluida en la carpeta *tinyos-2.x/apps*, que fue creada para realizar una introducción a Deluge. Con estas tres aplicaciones, se ha simulado el comportamiento normal de Deluge y varias situaciones en las que el componente podría fallar.

3.4.1 Prueba I (Reprogramar varias veces con programas distintos)

Comandos/pasos:

1.- Diseminar y reprogramar con prueba1 (comando -dr 1)
2.- Diseminar y reprogramar con prueba2 (comando -dr 2)
3.- Inyectar Blink en el volumen 1 (comando -i 1)
4.- Diseminar y reprogramar con Blink (comando -dr 1)

Desarrollo de las pruebas:

Durante la realización de estas pruebas, ha habido dos problemas que han provocado que no se pudiera conseguir el comportamiento deseado para Deluge. Al principio únicamente se podía diseminar y reprogramar una sola vez y si se intentaba volver a reprogramar o diseminar otro programa, ya fuese en un volumen utilizado o no, no había respuesta de la mota. Esto ocurre por un flag definido en el Makefile de la aplicación y al quitar dicho flag, se consigue poder reprogramar tantas veces como se quiera y utilizando cualquier volumen.

Es aquí donde aparece el segundo problema y es que aunque se consigue lo deseado, las motas no responden a los pings lanzados. Esto era culpa de otro flag en el mismo archivo, solo que esta vez era que no estaba definido y por eso no devolvía la información pedida.
Como se ha podido comprobar, hay que prestar mucha atención a los flags definidos en el archivo Makefile de la aplicación. Estos son los dos flags mencionados:

- #CFLAGS += -DDELUGE_BASESTATION: Este es el comentado en primer lugar, y es necesario si se desea que esa mota sea la base para todas las demás, de forma que solo tiene que estar definida en la mota que haga de base.

- #CFLAGS += -DDELUGE_LIGHT_BASESTATION. En el caso de este, es necesario si se quiere hacer ping a la mota y en este proyecto estará presente en las dos aplicaciones, cliente y base.

3.4.2 Prueba II (Reprogramar varias veces con programas distintos pero sólo un volumen)

Comandos/pasos:

1.- Diseminar y reprogramar con prueba1 (comando -dr 1)
2.- Inyectar prueba2 en el volumen 1 (comando -i 1)
3.- Diseminar y reprogramar con prueba2 (comando -dr 1)
4.- Inyectar Blink en el volumen 1 (comando -i 1)
5.- Diseminar y reprogramar con Blink (comando -dr 1)

Desarrollo de las pruebas:

Esta prueba es la misma que la realizada en el apartado de arriba, solo que esta vez, únicamente se ha utilizado uno de los volúmenes definidos por Deluge, definiendo los otros dos con solo unos pocos bytes. Como era de esperar el resultado es satisfactorio.

3.4.3 Prueba III (Apagar mota mientras se disemina)

Comandos/pasos:

1.- Diseminar y reprogramar con prueba1 (comando -dr 1)
2.- Diseminar y reprogramar con prueba2 (comando -dr 1)
3.- Cuando se está diseminando apagar la mota y volver a encender

Desarrollo de las pruebas:

Al apagar las motas mientras se disemina una aplicación y volver a encenderla, se ha visto como se vuelve a intentar diseminar la aplicación, de forma que el resultado es favorable. Se puede observar que al apagar la mota no borra los datos ya diseminados y sigue desde el punto en el que estaba, ya que una vez que vuelve a diseminar, lo hace más rápido que si tuviese que empezar de nuevo.

Por otra parte, también se ha querido probar el comportamiento del comando stop, que anula los comandos de diseminar y/o reprogramar. En este caso, al apagar la mota, se ha ejecutado el comando -s y se ha encendido la mota cliente de nuevo. Esta vez, al no haber terminado de diseminar y por lo tanto no tener una versión estable, se ha reseteado el programa que tenía anteriormente.

3.4.4 Prueba IV (Apagar mota mientras se reprograma)

Comandos/pasos:

1.- Diseminar y reprogramar con prueba1 (comando -dr 1)
2.- Diseminar y reprogramar con prueba2 (comando -dr 1)
3.- Cuando se está reprogramando apagar la mota y volver a encender

Desarrollo de las pruebas:

En este caso, los resultados han sido los mismos que en la prueba anterior, es decir, al encenderlas de nuevo, siguen desde el punto en el que estaban. Eso se debe a que Deluge gestiona por sí mismo los fallos en caso de que la mota se apague o no disponga de energía.

Las motas IRIS tienen un mecanismo para poder forzar a la mota a recuperarse con la Golden Image. Esto se puede hacer únicamente de forma manual y se hace reseteando la mota en varias ocasiones y en un lapso muy corto de tiempo. En estos casos, la mota se quedaba bloqueada y no era capaz de recuperarse. Cómo este caso solo puede darse teniendo acceso a la mota físicamente (usando el botón de encendido de la mota), no es significativo para este proyecto.

4 Otros componentes software

Este capítulo se centra en realizar un análisis exhaustivo de los componentes software que TinyOS trae implementados y son interesantes para las aplicaciones a realizar. Estos componentes, nos proveen de varias interfaces, las cuales hay que ver cuáles son interesantes para el proyecto.

Los componentes elegidos para este análisis son los siguientes: la memoria flash, la radio, el GPS y la medida de la tensión de la batería. Esta elección se hizo en base a los requerimientos de este proyecto, como son la necesidad de tomar, almacenar y transmitir datos, para los que se necesitan los tres primeros componentes y la comprobación de uno de los elementos más condicionantes, la batería.

Para realizar este análisis, lo primero que se hace es una introducción de los componentes, mencionando algunas de las características más importantes de éstos y explicando cómo funcionan. Después, se hace un estudio de las interfaces que nos proveen dichos componentes. Con todo esto, se crea un programa de prueba simulando la tarea que va a realizar dentro de las aplicaciones. Por último, se realizan pruebas de esos programas y se examina si hay que hacer algún cambio o está todo correcto. Este último paso se repetirá hasta que funcione correctamente o se descarte el uso del componente.

4.1 Lectura/escritura de datos en la memoria Flash

En esta sección se realiza un análisis de la manera en la que están organizadas la lectura y la escritura en memoria Flash. Para ello se detalla cómo se particiona la memoria, se analizan los componentes que aporta TinyOS y se definen los programas y pruebas realizados para conocer la forma de implementar este aspecto en la implementación final.

4.1.1 Volúmenes

Como ya se ha visto en el apartado 2.1.1, la memoria Flash tiene capacidad para 640 KB y está dividida en dos memorias separadas, por un lado la memoria de programa de 128 KB, en la cual se guarda el programa que se ejecutará cuando el dispositivo esté en uso; y por el otro, la memoria de datos de 512 KB, la cual se puede utilizar para guardar los datos que se quieran. Esta última, se puede dividir en varias fracciones o volúmenes, que se podrán destinar a distintos tipos de datos.

Para este proyecto se necesitará dividir la memoria en varios volúmenes. En uno se guardaran los datos recibidos por el GPS. Para la programación con nuevas versiones de software de la mota de forma remota, como ya se ha comentado, se necesitan otros cuatro volúmenes.

Este fraccionamiento se especifica en un archivo concreto dentro de la carpeta de la aplicación. Es un archivo XML al que se le pondrá el nombre de “volumes-” seguido del nombre del modelo de la memoria flash que usa la mota. El dispositivo de memoria Flash que viene incorporado en el procesador de la mota IRIS es de la serie AT45DB de la empresa Adesto Technologies, por lo que el archivo se llama “volumes-at45db.xml”.

4.1.2 Componentes e interfaces

Existen tres formas distintas de escribir en la memoria flash, incluyendo objetos pequeños, registros (log, en inglés) y objetos de gran tamaño. En este proyecto se usan registros ya que es lo más indicado para guardar eventos y estructuras de poco tamaño. En este tipo de escritura en memoria Flash se utiliza un único componente, *LogStorageC*.

Para poder utilizar este componente, hay que crearlo con los dos parámetros necesarios, que son el nombre del volumen en el que se guardarán los datos manejados por este y True o False, dependiendo de si se necesita que los datos sean guardados mediante una cola circular o no.

LogStorageC ofrece dos interfaces distintas, una para la operación de lectura, llamada *LogRead* y otra para la escritura, llamada *LogWrite*. En la Tabla 3 y en la Tabla 4 se muestran los comandos que nos proveen ambas, además de las funciones que se tendrán que implementar, o mejor dicho, eventos que este componente enviará.

LogRead	
Comandos	Descripción
error_t read(void* buf, storage_len_t len)	Lee el siguiente registro a la posición actual.
storage_cookie_t currentOffset()	Devuelve la posición actual de lectura.
error_t seek(storage_cookie_t offset)	Marca offset como la posición de lectura.
storage_len_t getSize()	Devuelve el tamaño del volumen.
Eventos	Descripción
void readDone(void* buf, storage_len_t len, error_t error)	Señala la finalización de la operación de lectura.
void seekDone(error_t error)	Reporta el éxito o fracaso de la operación seek.

Tabla 3. Comandos y eventos de la interfaz *LogRead*.

LogWrite	
Comandos	Descripción
error_t append(void* buf, storage_len_t len)	Agrega un regitro al volumen.
storage_cookie_t currentOffset()	Devuelve la posición actual de escritura.
error_t erase()	Inicia la operación de borrado.
error_t sync()	Asegura que todos los datos sigan guardados.
Eventos	Descripción
void appendDone(void* buf, storage_len_t len, bool recordsLost, error_t error)	Señala la finalización de la operación de escritura.
void eraseDone(error_t error)	Reporta el éxito o fracaso de la operación erase.
void syncDone(error_t error)	Reporta el éxito o fracaso de la operación sync.

Tabla 4. Comandos y eventos de la interfaz *LogWrite*.

4.1.3 Desarrollo del programa de pruebas y comprobación del funcionamiento

4.1.3.1 Primera versión

Para la primera versión se ha realizado un programa simple en el que se han guardado 30 registros de un tipo de datos creado para esta prueba, que no es más que una estructura con un contador y el id de la mota. Después, esos datos serán leídos en dos tandas, primero se leen los 15 primeros registros y se comprueba que los datos leídos concuerdan con lo que debería ser. Luego se hace lo mismo con los últimos 15 registros. Se presta especial atención a los índices de posición u offsets que manejan ambas interfaces, de forma que se conozca su funcionamiento para más adelante poder hacer un control más adecuado de la gestión de la memoria flash.

Una vez ejecutado el programa, los resultados no son los esperados. Las escrituras se realizan de manera correcta, pero como se puede apreciar en la Figura 3, a la hora de realizar las lecturas, aparece una primera lectura incorrecta, en la que los datos (id, contador y offset) que aparecen son todos iguales a 0. Lo que significa que en la posición 0 de memoria no ha encontrado ningún dato válido. Por otra parte, en las dos figuras que aparecen abajo, podemos comprobar que los índices de ambas interfaces no son iguales, siendo el offset de escritura 7620 mientras el de lectura es 7626. De esta forma, cuando en el programa se ha implementado un control con la condición de que paren las lecturas cuando los dos índices sean iguales no se ejecuta nunca y está todo el rato leyendo la última posición.

```
[3] Contador: 24, posicion: 6350.
[3] Contador: 25, posicion: 6604.
[3] Contador: 26, posicion: 6858.
[3] Contador: 27, posicion: 7112.
[3] Contador: 28, posicion: 7366.
[3] Contador: 29, posicion: 7620.


                LECTURAS

[0] Contador: 0, Posicion: 0.
[3] Contador: 0, Posicion: 260.
[3] Contador: 1, Posicion: 514.
[3] Contador: 2, Posicion: 768.
[3] Contador: 3, Posicion: 1022.
[3] Contador: 4, Posicion: 1276.
[3] Contador: 5, Posicion: 1530.
[3] Contador: 6, Posicion: 1784.
[3] Contador: 7, Posicion: 2038.
```

Figura 3. Final de las escrituras e inicio de las lecturas para la primera versión.

```
[3] Contador: 25, Posicion: 6610.
[3] Contador: 26, Posicion: 6864.
[3] Contador: 27, Posicion: 7118.
[3] Contador: 28, Posicion: 7372.
[3] Contador: 29, Posicion: 7626.
[3] Contador: 29, Posicion: 7874.
[3] Contador: 29, Posicion: 7874.
[3] Contador: 29, Posicion: 7874.
[3] Contador: 29, Posicion: 7874.
```

Figura 4. Final de las lecturas para la primera versión.

Como conclusión de esta prueba, se puede decir que los offset de ambas interfaces no son fiables por lo que se necesita otro mecanismo de control para conocer cuándo es el final de las lecturas. Además, se necesita controlar que las lecturas realizadas son datos válidos.

4.1.3.2 Segunda versión

En esta versión se han añadido dos índices nuevos para sustituir los offsets de logRead y logWrite. En este caso son dos contadores, que cuentan cuantas lecturas y escrituras se han realizado. Asimismo, se ha tratado de solucionar el problema de la primera lectura errónea mediante una condición que obligue a comprobar que los datos son válidos.

En la figura 5 y 6, se aprecia como los resultados son los esperados, siendo los mismos datos los almacenados que los recuperados al leer.

```
[3] Contador: 27, Escritura 28.
[3] Contador: 28, Escritura 29.
[3] Contador: 29, Escritura 30.


                LECTURAS

[3] Contador: 0, Lectura 1.
[3] Contador: 1, Lectura 2.
[3] Contador: 2, Lectura 3.
[3] Contador: 3, Lectura 4.
```

Figura 5. Final de las escrituras e inicio de las lecturas para la segunda versión.

```
[3] Contador: 23, Lectura 24.
[3] Contador: 24, Lectura 25.
[3] Contador: 25, Lectura 26.
[3] Contador: 26, Lectura 27.
[3] Contador: 27, Lectura 28.
[3] Contador: 28, Lectura 29.
[3] Contador: 29, Lectura 30.
FIN
```

Figura 6. Final de las lecturas para la segunda versión.

4.2 Radio

En los siguientes apartados se analiza el comportamiento de la radio, que es la encargada de mandar y recibir los datos recogidos por las motas que llevan las gaviotas a la estación base. Esta tarea se realiza mediante el envío de mensajes, los cuales pueden ser de distintos tipos, de los que se hablará. Además, se analizarán los componentes e interfaces que se utilizan y se comentarán las pruebas realizadas para ratificar el correcto funcionamiento de estos.

4.2.1 Componentes e interfaces

TinyOS proporciona una serie de interfaces para abstraer los servicios de comunicaciones subyacentes y un número de componentes que proporcionan (implementan) estas interfaces. Todas estas interfaces y componentes utilizan una estructura de mensajes común, llamado *message_t*. En este proyecto se utiliza la capa Active Message (AM) para multiplexar el acceso a la radio, ya que habrá varios servicios usando la radio al mismo tiempo.

Los componentes principales son tres, *ActiveMessageC, AMSenderC* y *AMReceiverC*. El primero de ellos, es usado para proveer de la interfaz *SplitControl*, encargada de encender y apagar la radio. En la Tabla 5 se muestran los comandos y eventos de dicha interfaz.

SplitControl	
Comandos	Descripción
error_t start()	Inicia la operación de inicio de la radio.
error_t stop()	Inicia la operación de parada de la radio.
Eventos	Descripción
void startDone(error_t error)	Reporta el éxito o fracaso de la operación start.
void stopDone(error_t error)	Reporta el éxito o fracaso de la operación stop.

Tabla 5. Comandos y eventos de la interfaz *SplitControl*.

En cuanto a *AMSenderC* y *AMReciverC*, se necesita crear una instancia de ellos, a la que se le pasa como parámetro el tipo de mensajes a intercambiar. A pesar de que proporcionen varias interfaces, sólo se usarán tres de ellas, *AMSend* y *Packet*, para el envío de paquetes y *Receive* para la recepción. En las siguientes figuras se pueden observar las funcionalidades que brindan.

AMSend	
Comandos	Descripción
error_t send(am_addr_t addr, message_t* msg, uint8_t len)	Envia un mensaje a la dirección especificada.
error_t cancel(message_t* msg)	Cancela el envio requerido.
uint8_t maxPayloadLength()	Identico a la misma función especificada en Packet.
void* getPayload(message_t* msg, uint8_t len)	Identico a la misma función especificada en Packet.
Eventos	Descripción
void syncDone(error_t error)	Reporta el éxito o fracaso de la operación send.

Tabla 6. Comandos y eventos de la interfaz *AMSend*.

Packet	
Comandos	Descripción
void clear(message_t* msg)	Limpia el contendo del paquete.
uint8_t payloadLength(message_t* msg)	Devuelve la longitud del paquete.
void setPayloadLength(message_t* msg, uint8_t len)	Asigna la longitud del paquete pasado.
uint8_t maxPayloadLength()	Devuelve la longitud máxima puede tener un paquete.
void* getPayload(message_t* msg, uint8_t len)	Devuelve un puntero al paquete deseado.

Tabla 7. Comandos y eventos de la interfaz *Packet*.

Receive	
Eventos	Descripción
message_t* receive(message_t* msg, void* payload, uint8_t len)	Recibe un paquete mediante un buffer y devuelve el paquete que usara en la siguiente recepción.

Tabla 8. Comandos y eventos de la interfaz *Receive*.

4.2.2 Tipos de mensajes

Como ya se ha comentado, al crear los componentes *AMSenderC* y *AMReceiverC*, hay que especificar el tipo de mensaje que se enviará o recibirá. TinyOS define para ello un tipo de datos llamado *am_id_t* y será el usuario el que tenga que especificarlo, que no es más que un número entero.

En este proyecto se necesitarán definir dos tipos de mensajes: uno para los mensajes que enviarán los dispositivos colocados en las aves a la estación base y otro para los mensajes periódicos que envía la estación base para que los otros dispositivos sepan si están en rango de comunicación. A estos últimos se les denominará a partir de ahora con el nombre de *Heartbeat* y son necesarios para saber que la mota está en rango de enviar datos.

4.2.3 Desarrollo del programa de pruebas y comprobación del funcionamiento

El objetivo principal de esta prueba es asegurar la multiplexión del acceso a la radio, de forma que las motas cliente (las puestas en las aves), sean capaces de recibir Heartbeats mientras están descargando los datos almacenados en la flash. No obstante, también se presta atención a las direcciones que maneja la radio, es decir, si se pueden enviar mensajes sólo entre motas concretas y no sólo mediante broadcast.

Con este fin se han creado dos aplicaciones, una para la base y la otra para las motas cliente. La primera envía un Heartbeat cada medio segundo e imprime por pantalla los datos recibidos por los demás nodos. El software para los dos nodos cliente utilizados para esta prueba, se basa en enviar el valor de un contador a la base, siempre y cuando reciba Heartbeats. Para ello, se han creado dos tipos de mensaje, AM_HEARTBEAT y AM_LOG.

En la Figura 7 se pueden observar varias capturas de pantalla que corresponden a dos pruebas distintas realizadas. Primero se ha realizado un test con una base y un cliente y después se ha incorporado otro cliente a la prueba con el fin de ver si hay colisiones entre las transmisiones que realizan sobre una misma base. En la primera imagen se puede ver el correcto funcionamiento de los clientes en los que se envía el valor del contador cuando hay heartbeats. Si no los recibe, como es el caso de que la base está apagada, no envía ni realiza ninguna acción.

En la segunda imagen, correspondiente a la primera prueba, se puede advertir cómo está constantemente mandando heartbeats y que sólo recibe de la mota con id 1 hasta el valor 46, momento en el que se ha apagado dicha mota. Por su parte, en la tercera imagen, toman parte dos motas cuyo id son 3 y 12. En este caso, se ve como entre cada heartbeat enviado recibe mensajes de ambos clientes.

Clientes

```
Heartbeat recibido!!
Enviando(Id: 1, Count= 23)...
Envio correcto!
Enviando(Id: 1, Count= 24)...
Envio correcto!

Heartbeat recibido!!
Enviando(Id: 1, Count= 25)...
Envio correcto!
Enviando(Id: 1, Count= 26)...
Envio correcto!
Enviando(Id: 1, Count= 27)...
Envio correcto!

Heartbeat recibido!!
Enviando(Id: 1, Count= 28)...
Envio correcto!
Enviando(Id: 1, Count= 29)...
Envio correcto!
Enviando(Id: 1, Count= 30)...
Envio correcto!
```

Base con 1 mota

```
[1] Contador: 41.

Enviando Heartbeat...
Heartbeat enviado!!!

[1] Contador: 42.
[1] Contador: 44.

Enviando Heartbeat...
Heartbeat enviado!!!

[1] Contador: 45.
[1] Contador: 46.

Enviando Heartbeat...
Heartbeat enviado!!!


Enviando Heartbeat...
Heartbeat enviado!!!


Enviando Heartbeat...
Heartbeat enviado!!!
```

Base con 2 motas

```
[3] Contador: 76.
[12] Contador: 51.
[3] Contador: 77.
[12] Contador: 52.
[3] Contador: 78.
[12] Contador: 53.

Enviando Heartbeat...
Heartbeat enviado!!!

[3] Contador: 79.
[12] Contador: 54.
[3] Contador: 80.
[12] Contador: 55.

Enviando Heartbeat...
Heartbeat enviado!!!

[3] Contador: 81.
[12] Contador: 56.
[3] Contador: 82.
[12] Contador: 57.
```

Figura 7. Capturas de pantalla de diversas pruebas.

Como conclusión de esta prueba se puede comprobar cómo se puede recibir y enviar al mismo tiempo sin que se produzca ningún error. Además, se ha contemplado cómo ha recibido los mensajes de dos motas mientras envían a la vez a una misma mota. No obstante, esta segunda prueba se ha realizado únicamente con dos dispositivos y enviando mensajes de 2 bytes de longitud, por lo que no se puede asegurar que en el caso real de este proyecto (20 nodos y 9 bytes de longitud) vaya a ser exitoso.

4.3 Medición de la posición

En la siguiente sección se realiza el estudio del instrumento encargado de hacer las mediciones de las posiciones de las gaviotas, que son las que se vienen buscando en este proyecto. Este dispositivo es un GPS de la compañía GlobalTop Tech y el modelo que se maneja es el FGPMMOPA6H. Este modelo utiliza la nueva generación de chip GPS MT3339 que tiene un gran nivel de sensibilidad (-165dB) y uno de los consumos de energía más bajos del mercado.

Para el análisis de este componente, se siguen los mismos pasos que en los anteriores. Primero se analizan las características más importantes, para después detallar los componentes e interfaces que TinyOS proporciona para el manejo del GPS. Por último, se describe la aplicación creada para asegurar el correcto funcionamiento de este componente.

4.3.1 Funcionamiento del GPS

La configuración y la petición de datos al GPS se hace mediante unos comandos definidos en el pack PMTK command packet. Estos comandos se pasan, a su vez, mediante comandos en la aplicación que se quiera crear, pero de eso se hablará en otro apartado de esta misma sección.

Existen tres tipos de comandos para pedir los datos. El primero de ellos es NMEA y este nos devuelve los datos del GPS en varios paquetes distintos, por lo que este no será elegido. Para ello, están los otros dos comandos, que envían la información en un solo paquete. Uno de ellos es Global Top Binary Sentence, que devuelve los datos en binario y que no ha sido el elegido ya que hay que realizar varias conversiones para poder manejar los datos. Por último, esta Global Top One Sentence, que es el que más se ajusta a las necesidades del proyecto. Este comando devuelve una cadena en ASCII con varios campos como son latitud, longitud, día…

Este comando hace que esté constantemente midiendo la posición, haciendo una medición tras otra sin ninguna pausa. Este proceso consume mucha batería de forma que no es conveniente y se necesita de un mecanismo para hacer que el dispositivo entre en un estado de descanso hasta hacer la siguiente medición. Con motivo de este problema, se hace uso de otro comando: Periodic mode. Este comando tiene varios parámetros y uno de ellos es el modo de operación en el que poner el GPS. Alternando entre el modo normal y el modo back-up, se puede conseguir un consumo de energía óptimo.

4.3.2 Datos recibidos del GPS

Los datos devueltos por el GPS son de casi 90 bytes, como se puede observar en la Figura 8, una cantidad muy grande dada la limitación de la memoria flash. Por lo tanto, solo se cogerán los datos más importantes y se guardarán en un tipo de datos que se creará para almacenarlos. Los datos más significativos son el día, el mes, la hora y los minutos de la toma de la medición, la latitud y la longitud.

$PGTOP , 064951.000 , 2307.1256 , N , 12016.4438 , E , A , 0.03 , 165.48 , 0.03 , N , 0.06 , K*Checksum

Cabecera Tiempo UTC Latitud Norte/sur Longitud Este/oeste

Figura 8. Ejemplo de paquete One Sentence devuelto por el GPS

No obstante, como la información está codificada en una cadena de números ASCII es necesaria una conversión antes de almacenarla. Para ello, se ha creado una función que separa los datos en los campos definidos por One Sentence, los convierte a decimal y coge los datos que interesa guardar, descartando el resto para no ocupar memoria.

4.3.3 Componentes e interfaces

Para la interacción con el módulo del GPS, únicamente es necesario un componente, *GPSC*. *GPSC* no está implementado por TinyOS, pero ha sido añadido a la biblioteca de TinyOS para poder utilizarlo en este proyecto. Este componente, a su vez, nos provee de 3 interfaces, *Init*, *StdControl* y *GPSData*. De éstas, sólo se utilizan la primera, que nos proporciona un comando para iniciar el GPS y la última, que es mediante la que se envía los comandos y se reciben los datos.

Init	
Comandos	Descripción
error_t init()	Inicia el componente.

Tabla 9. Comandos y eventos de la interfaz *Init*.

GPSData	
Comandos	Descripción
error_t sendCommand (uint8_t *buf)	Envio de un comando al GPS.
Eventos	Descripción
void sendDone ()	Reporta el éxito o fracaso de la operación sendComand.
void dataReady(uint8_t *buf, uint16_t length)	Devuelve un buffer con el valor medido por el GPS.

Tabla 10. Comandos y eventos de la interfaz *GPSData*.

4.3.4 Desarrollo del programa de pruebas y comprobación del funcionamiento

La prueba de este componente se ha realizado mediante una aplicación de ejemplo, cuyo funcionamiento es muy simple. Primero se realiza la medida de la posición. Esta se realiza cada diez segundos y nos proveerá de una cadena de números en código ASCII, separados en varios campos. Con este dato, se ejecuta el algoritmo para separar esos campos y guardarlos en una estructura creada para este tipo de datos. Como ya se ha comentado, solo se guardan los datos más significativos. Por último, se imprimen por pantalla los datos, en un formato que nos indica la hora, la latitud y la longitud, como se puede observar en la Figura 8. La latitud y la longitud únicamente se han imprimido con los grados y minutos, pero si se desea se puede utilizar más precisión.

```
[8:41] 42.84 2.68
[8:41] 42.84 2.68
[8:41] 42.84 2.68
```

Figura 9. Datos recibidos del GPS

Para comprobar que es correcta la medida se ha utilizado Google Maps, cogiendo las coordenadas desde las que se han hecho las medidas. En la siguiente figura se puede ver como ambas medidas coinciden.

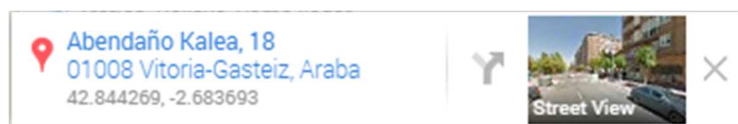


Figura 10. Datos recogidos mediante Google Maps.

4.4 Medición de la tensión de la batería

Un elemento muy importante en este proyecto es la batería, ya que aunque lleve una placa fotovoltaica, puede que no siempre se disponga de batería suficiente para llevar a cabo ciertas acciones. Una de ellas es realizar mediciones con el GPS, que necesitará un mínimo de 2,7 V para confirmar que el dato devuelto es válido. Por este motivo, se necesitará medir la tensión periódicamente y de ello trata este apartado.

4.4.1 Componentes e interfaces

Para la medición de la tensión se necesita instanciar un componente llamado *VoltageC*. Este componente utiliza, a su vez, la interfaz *Read*. Leer un dato consta de dos fases, y está dividida en el comando *Read.read()* y el evento *Read.readDone().* Ambos se detallan en la Tabla 6.

Read	
Comandos	Descripción
error_t read()	Inicia la operación de lectura de un sensor.
Eventos	Descripción
void readDone(error_t result, val_t val)	Devuelve el valor medido del sensor.

Tabla 11. Comandos y eventos de la interfaz *Read*.

4.4.2 Medidas y prueba de funcionamiento

El CAD (conversor analógico/digital) incorporado en el dispositivo IRIS es de 10 bits, por lo que los valores devueltos no son los voltios de la batería, sino un valor equivalente. Para conseguir saber el valor en voltios nos valdremos de la función representada en la Figura 11. Las medidas se realizan en milivoltios, ya que NesC no es capaz de almacenar datos de tipo float. Para IRIS, el valor de referencia para el CAD es de 1,1V.

$$V = \frac{(Vref * 1024)}{data}$$

Donde:
Vref = valor de referencia de la mota
data = valor devuelto por el sensor

Figura 11. Función para conocer en voltios el valor devuelto por el ADC.

Con el fin de probar este componente, se ha realizado un sencillo programa que muestre el funcionamiento del mismo y compruebe si los valores devueltos son razonables. Los resultados son aceptables, de forma que se puede concluir que su funcionamiento es correcto.

```
Valor del sensor 342
Valor en voltios 3293mV

Valor del sensor 342
Valor en voltios 3293mV

Valor del sensor 342
Valor en voltios 3293mV
```

Figura 12. Resultado de las pruebas

5 Diseño de las aplicaciones

En este capítulo se realiza el diseño de las dos aplicaciones que se precisan en este proyecto: una para los dispositivos que estarán incorporados en las aves, que recogerán cada cierto tiempo la posición en la que están y otra para la mota que actúa como estación base, que es la encargada de la descarga de los datos recogidos por las demás y de mantener el software de estas actualizado.

5.1 Diseño de la aplicación de la estación base

El diseño de este software tiene que dar solución a los dos cometidos que tiene, la recepción de información por parte de los clientes y el envío de una señal para que los clientes sepan si están a rango de enviar.

Lo primero que se realiza después de que el componente se haya iniciado, es iniciar la radio. Una vez iniciada la radio, se inicia un timer o reloj de forma periódica, para que cada cierto periodo de tiempo señale un evento. Este evento tiene como objetivo saber cuándo se tienen que enviar los heartbeats, descritos en el apartado 3.2.2, por lo que cuando se señala se envía un paquete de tipo heartbeat a las motas cliente. Además de enviar estos paquetes, la aplicación está continuamente pendiente de recibir datos. En la Figura 13 se muestra un gráfico con todo este comportamiento.

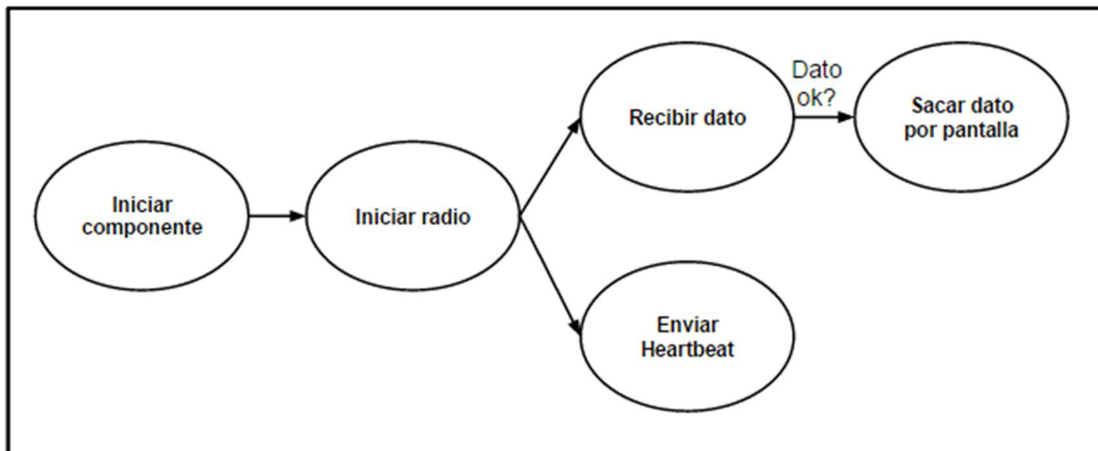


Figura 13. Gráfico del diseño de la aplicación base.

5.2 Diseño de la aplicación cliente

El enfoque que se sigue en el diseño de la aplicación cliente, es el propuesto en el proyecto realizado con anterioridad a este. Este enfoque trata de diseñar un programa basado en la ejecución de diferentes modos de funcionamiento. Los modos de funcionamiento que se proponen tienen una relación directa con el lugar en el que se encuentre el dispositivo en cada momento. Los modos propuestos son: fuera del nido, entrando al nido, dentro del nido y saliendo del nido. Además, se propone un quinto, el modo seguro, al que entran si se dan ciertas situaciones, como son la tensión baja de la batería y la necesidad de actualización software, entre otros, pero que en esta implementación se tratan dentro de cada modo.

Estos estados se han definido dados los hábitos de estas aves. Las gaviotas suelen pasar las horas de luz del día fuera de la zona de anidamiento, mientras que en las de oscuridad vuelven al nido. Es por ello, que la medición de la posición se realiza cuando estén fuera de la zona de acción de la estación base, colocada en la zona de anidamiento. De forma que cuanto vuelvan por la noche a esta zona se recuperen todos los datos recogidos.

5.2.1 Fuera del nido

Esta situación se da cuando el ave esté fuera del alcance de la mota base. En este momento, el dispositivo guarda periódicamente la posición en la que se encuentra el ave a lo largo del día utilizando un GPS.

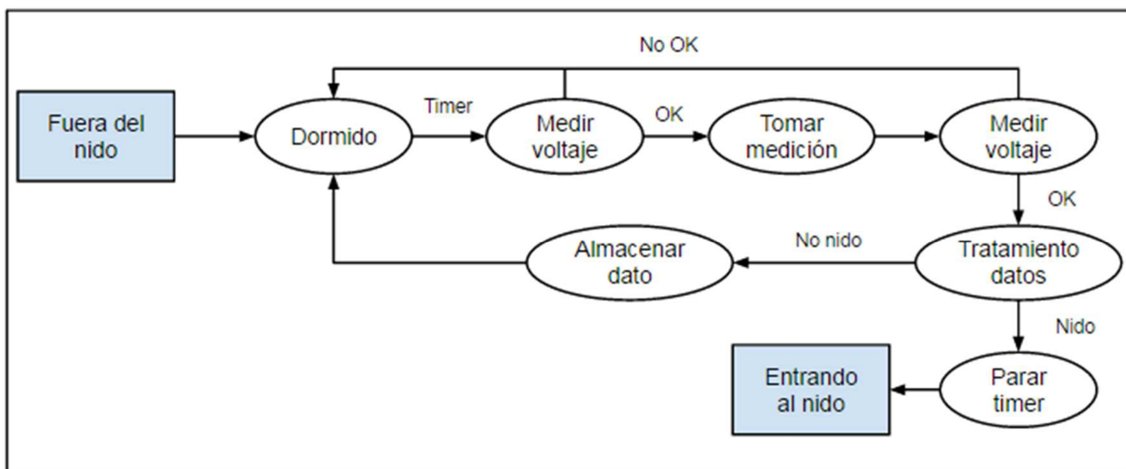


Figura 14. Gráfico del diseño cuando esta fuera del nido.

Este estado es un bucle que se repite hasta que se ha detectado mediante las coordenadas GPS que el ave está cerca del nido y se utiliza un timer para controlar cada iteración de este bucle. Mientras no se señale que el timer ha acabado, la mota y el GPS están dormidos o en modo ahorro de energía. Cuando hay una señal del timer, se mide la tensión de la batería para determinar si hay energía suficiente para que el GPS tome la posición de la gaviota. En caso positivo, coge las coordenadas y vuelve a medir la tensión, esta vez para asegurarse de que no ha habido una diferencia significativa entre las dos medidas. La existencia de una gran diferencia puede deberse a alguna alteración en el GPS o a que alguna de las dos medidas de tensión puede ser errónea, por lo que ese dato sería descartado.

Si todo fuese bien hasta este punto, el siguiente paso es coger los datos que nos interesan de la lectura del GPS. Como ya se ha comentado, los datos enviados por el GPS son demasiado pesados y como la memoria flash es de tamaño reducido, hay que utilizar un algoritmo de acortamiento.

Por otro lado, hay que certificar si el dispositivo se encuentra en el área de acción de la estación base. Si no fuera así, el dato recogido se almacenaría en memoria y se continuaría con este ciclo hasta que estuviese en una posición dentro del radio de la base. En ese momento, se para el timer que controla las mediciones y se pasa al siguiente modo, entrando al nido.

5.2.2 Entrando al nido

Una vez que se ha detectado que el dispositivo se encuentra en el radio de acción de la estación base, se necesita confirmar que está al alcance de realizar transmisiones por radio, esto se sabe si recibe algún Heartbeat. El objetivo principal de este modo es asegurar que el dispositivo está en condiciones de realizar la descarga de datos e iniciar los componentes necesarios para ello. En la Figura 15 se muestra el diseño de este modo:

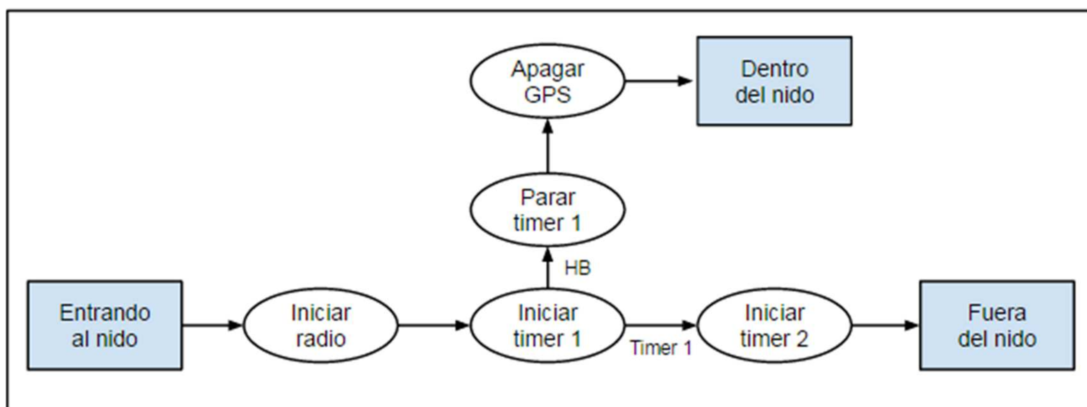


Figura 15. Gráfico del diseño cuando está entrando al nido.

El primer paso es asegurarse de que está al alcance de la base. Para ello, es necesario encender la radio y un timer que regule si recibe un Heartbeat o no. En este caso el timer uno hace esa función y el tiempo con el que será definido tiene que ser mayor al periodo entre Heartbeats. Si el timer llega a su fin, se sabe que no se ha recibido ninguna señal de la base y que por lo tanto no está al alcance. En caso de que esto ocurra, se inicia un segundo timer, el mismo timer que controla el modo fuera del nido y se pasa a dicho modo.

Por el contrario, si se recibe un Heartbeat antes de que el tiempo del timer uno se agote, significa que se está al alcance de la radio. Como no se desea que el timer uno señale el evento de terminado, hay que pararlo. Una vez parado, se apaga el GPS para ahorrar energía y entra en el siguiente modo, dentro del nido.

5.2.3 Dentro del nido

Como la radio no puede recibir simultáneamente de todas las motas, se necesita que las motas hagan la descarga de los datos una detrás de la otra. Para ello cada dispositivo tendrá una franja de tiempo para descargar la información.

Se ha utilizado un algoritmo para gestionar los tiempos en los que las motas descargan los datos sin que haya interferencias entre ellas. Dentro de este algoritmo hay que tener en cuenta el tiempo que se deja para que las motas pueden ser reprogramadas y en el cual no habrá comunicaciones de los clientes a la base.

	Reprogramación					Descarga de datos																			
Tiempo total	5 minutos					10 minutos																			
Minuto	0	1	2	3	4	5		6		7		8		9		10		11		12		13		14	
Motas	Todas					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Tabla 12. Algoritmo de gestión de descarga de datos.

En la Tabla 12 se muestra como la solución dada es la de utilizar franjas de 15 minutos en los que 5 de ellos están reservados para la posible reprogramación del dispositivo y los otros 10 a las transmisión de los datos de los clientes a la base. Como la cantidad de motas que se utilizarán en la práctica son 20, cada mota tendrá 30 segundos en los que poder descargar. Este tiempo es el tiempo máximo que se ha estimado que puede tardar en enviar todos los datos.

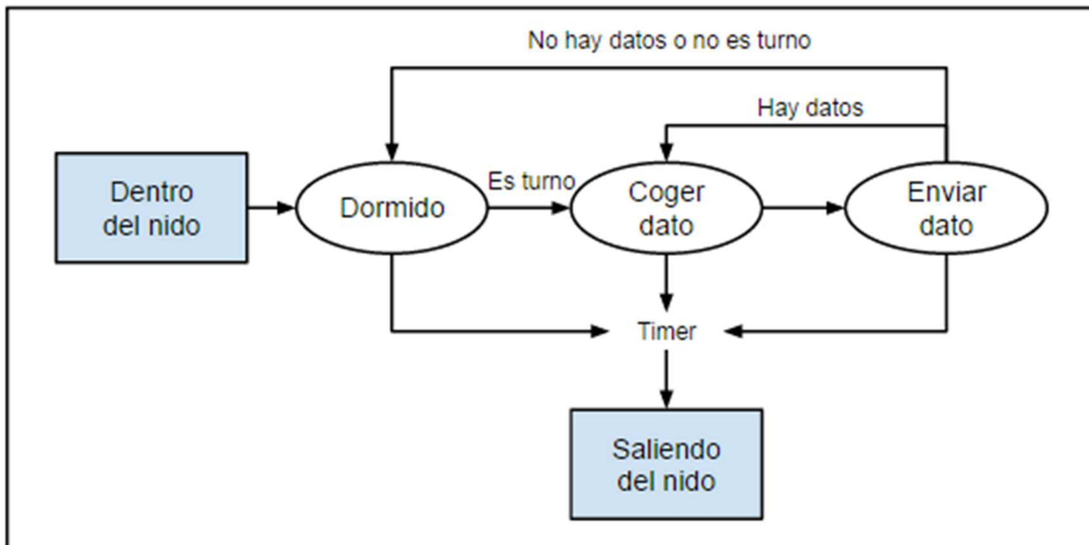


Figura 16. Gráfico del diseño cuando está dentro del nido.

Para empezar, cuando se dice que la mota está dormida, lo que está pasando es que la mota no realiza ninguna acción aparte de estar escuchando para recibir Heartbeats, siendo estos los que gestionen la descarga de datos de las mota cliente. Estos mensajes traen un campo especial, en el que se envía un número correspondiente a una mota, la que tendrá el turno de descargar. Una vez se reciba el Heartbeat con el número de la propia mota, se inicia un bucle en el que primero se sacan de memoria flash los datos y luego se envían a la base. Este bucle acaba bien cuando no hay más datos, o bien cuando ya no es el turno de esa mota.

Por otro lado, se dispone de un timer que, como en el modo anterior el primer timer, es el encargado de gestionar si está dentro del nido o no. Este timer se reinicia cada vez que la mota recibe un Heartbeat, de forma que si en un momento dado se le acaba el tiempo es porque ya no está al alcance de la base y por lo tanto está fuera del nido. En caso de que esto ocurra y no esté en plena descarga de datos, el dispositivo pasa al modo saliendo del nido. Se puede dar el caso de que salga sin haber terminado de intercambiar todos los datos. Si esto ocurre, estos datos no son borrados y siguen en la memoria, con el inconveniente de que la memoria no tiene todo su espacio para la siguiente transición de fuera del nido.

5.2.4 Saliendo del nido

A este modo se entra cuando el dispositivo colocado en el ave deja de recibir Heartbeats de la estación y no es más que una simple transición hacia el siguiente modo. El objetivo de este modo no es más que iniciar los componentes necesarios para el modo fuera del nido y apagar los que no son necesarios para ahorrar energía.

Lo primero que realiza es el apagado de la radio, que no es necesario para el modo fuera del nido. Después se arranca el módulo del GPS y se inicia el timer que gestiona periódicamente la medición de la posición de la gaviota.

Figura 17. Gráfico del diseño cuando está saliendo del nido.

6 Implementación de las aplicaciones

En este capítulo se analiza cómo se han implementado las dos aplicaciones que son el objetivo de este proyecto. Para empezar se estudian los elementos que intervienen en ambas aplicaciones, como son ciertas variables y estructuras de datos. Después, se explica el procedimiento seguido en la implementación de cada una de las dos aplicaciones. El enfoque seguido es analizar cada uno de los archivos que contiene la carpeta en la que se implementan ambas aplicaciones. Los archivos son en ambos caso los siguientes: Makefile, el que define los volúmenes (volumes-at45db.xml), un archivo cabecera (.h), la configuración y el módulo.

6.1 Elementos comunes

Lo primero que se analiza en este capítulo son los elementos que aparecen en la implementación de ambas aplicaciones, como son las estructuras de datos y algunas variables.

6.1.1 Estructuras de datos

El dato más importante que se maneja en este proyecto es la toma de mediciones de la posición de las gaviotas, por lo que se necesita crear un tipo de datos que recoja estos datos. Para ello se ha creado una estructura de datos llamada log_t, cuyos campos son los que se pueden observar en la Figura 18.

```
typedef nx_struct log_t {
  nx_uint8_t mes;
  nx_uint8_t dia;
  nx_uint8_t hora;
  nx_uint8_t minutos;
  nx_int8_t lat_grados;
  nx_uint8_t lat_min;
  nx_uint16_t lat_mmin;
  nx_int8_t long_grados;
  nx_uint8_t long_min;
  nx_uint16_t long_mmin;
} log_t;
```

Figura 18. Estructura de datos log_t.

No obstante, no es la única estructura que se ha requerido, ya que para el intercambio de mensajes por radio se necesitan otras dos. Una para definir los mensajes que se envían de los clientes a la estación base y la otra para definir los que se envían de la base a los clientes, es decir, los Heartbeats.

```
typedef nx_struct heartbeat_t {
    nx_uint8_t numero;
} heartbeat_t;
```

```
typedef nx_struct msg_t {
  nx_uint8_t id;
  log_t log;
} msg_t;
```

Figura 19. Estructuras de datos heartbeat_t y msg_t.

6.1.2 Variables de entorno

Las variables que usan las dos aplicaciones son las necesarias para radio, más concretamente, el tipo de mensaje que hay que definir para los mensajes a enviar, que como se ha visto en el apartado 4.2.2 son dos, uno para los Heartbeats y otro para los datos enviados de los clientes a la base.

```
AM_HEARTBEAT = 6,
AM_LOG = 7
```

Figura 20. Variables de entorno.

6.2 Implementación de la aplicación base

En este apartado se analizan los archivos que toman parte en la implementación de la aplicación que se ejecutará en la mota que realiza la función de base.

6.2.1 Makefile de la aplicación base

En el archivo Makefile se han incluido las líneas de código necesarias para que la mota sea capaz de realizar la función de base de Deluge (véase el apartado 3.4.1). Además, se han incluido otros flags necesarios para el correcto funcionamiento de Printf. El archivo Makefile queda de la manera que se muestra en la Figura 21.

```
COMPONENT=BaseC

CFLAGS += -DDELUGE_BASESTATION
CFLAGS += -DDELUGE_LIGHT_BASESTATION      DELUGE

CFLAGS += -I$(TOSDIR)/lib/printf
CFLAGS += -DNEW_PRINTF_SEMANTICS          PRINTF

include $(MAKERULES)
```

Figura 21. Contenido del archivo Makefile de la base.

6.2.2 volumes-at45db.xml de la aplicación base

Como ya se ha estudiado anteriormente, este archivo es necesario para poder separar la memoria flash en varios volúmenes. En la implementación de la base se usan cuatro volúmenes para Deluge, de forma que el archivo quedaría como en la Figura 22.

```
<volume_table>
  <volume name="GOLDENIMAGE" size="65536" base="0" />
  <volume name="DELUGE1" size="65536"/>
  <volume name="DELUGE2" size="0"/>
  <volume name="DELUGE3" size="0"/>
</volume_table>
```

Figura 22. Contenido del archivo volumes-at45db.xml de la base.

6.2.3 Archivo de cabecera de la aplicación base

En este archivo, que tiene el nombre de base.h, se definen varias constantes y los tipos de datos que se utilizan a lo largo de la implementación de la aplicación base, como son los mencionados en el apartado 6.1. A parte de estos datos, es necesaria otra variable, el tiempo que pasa entre Heartbeats.

6.2.4 Implementación de la configuración de la aplicación base

NesC tiene dos tipos de componentes, módulos y configuraciones. En este caso se analiza la configuración, es decir, las conexiones entre todos los componentes que toman parte en esta aplicación. Por este motivo, es interesante analizar estos componentes y qué funcionalidades se aprovechan.

COMPONENTES

MainC. Es el encargado de iniciar todos los componentes cuando se encienda la mota. La interfaz de este componente se llama Boot y devuelve un evento cuando se haya iniciado correctamente.

DelugeC. Este componente no provee ninguna interfaz, únicamente hay que añadirlo.

ActiveMessageC, AMSenderC y AMReceiverC. Son los necesarios para la radio. Cuando creemos las instancias de AMSenderC y AMReceiverC, tendremos que pasarle como parámetro AM_HB a la primera y AM_LOG a la segunda.

TimerMilliC. Es necesario crear un timer para la aplicación, al que se renombra con el nombre de Timer.

PrintfC y SerialStartC. Estos dos componentes solo se añadirán, no tienen ninguna interfaz que proveer y por lo tanto ningún comando o evento. No obstante tienen algunas funciones de las cuales se puede hacer uso.

CONEXIONES

Estos componentes proveen de varias interfaces, de la cuales no se utilizan todas. Es necesario realizar enlaces o “cablear” (‘wire’, en inglés) las interfaces a las que llaman los módulos incluidos en la aplicación y el componente en el que éstas interfaces vienen implementados. Por ejemplo, como se puede observar en la Figura 23, cuando en el módulo ‘Base’ se hace uso de la interfaz *Boot*, esta interfaz será provista e implementada por el componente *MainC*. Estos son los enlaces o “cables” (‘wires’ en inglés) que se realizan:

```
Base.Boot -> MainC;
Base.Timer -> Timer;
Base.AMControl -> ActiveMessageC;
Base.Receive -> AMReceiverC;
Base.Packet -> AMSenderC;
Base.AMSend -> AMSenderC;
```

Figura 23. Conexiones entre componentes.

6.2.5 Implementación del módulo de la aplicación base

La implementación se ha seguido de acuerdo al diseño realizado. Una vez encendida la mota, los componentes que toman parte en la aplicación se inician mediante *Boot*, que señala el evento *Boot.booted*. En la respuesta a dicho evento se llama al comando *start* de la interfaz *SplitControl*, para que inicie la radio.

Con la radio encendida, se pueden realizar dos acciones, enviar y recibir. Para recibir se dispone de la interfaz *Receive*, que está continuamente a la escucha y que señala el evento *receive* si recibe algún dato del tipo con el que fue creado (AM_LOG). El envío, como ya se ha comentado, se envía mediante un timer, el cual se inicia una vez señalizado el evento *startDone* de *SplitControl*. Este timer se define con el periodo de un Heartbeat: 30 segundos. Cuando el evento *fired* del timer sea señalado, se realiza el envío mediante *send* de *AMSend*.

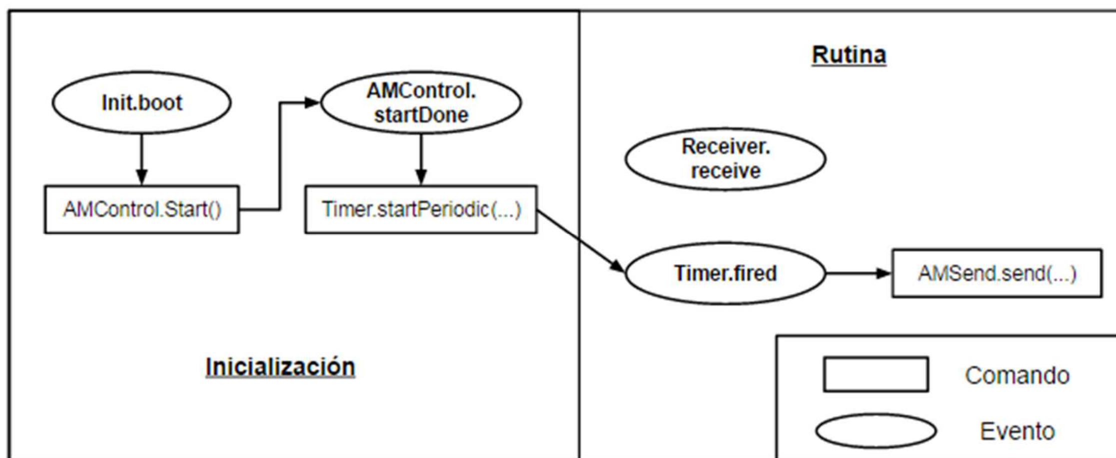


Figura 24. Implementación de la aplicación de la base

6.3 Implementación de la aplicación cliente

En este apartado se analizan los archivos que toman parte en la implementación de la aplicación que se ejecutará en las motas que irán implantadas en las aves.

6.3.1 Makefile de la aplicación cliente

En esta aplicación hace falta incluir una línea de código para que la toma de mediciones se haga mediante una placa concreta, en este caso el GPS que se utiliza en este proyecto.

```
COMPONENT=proyecto

SENSORBOARD?=TarGps

CFLAGS += -I$(TOSDIR)/lib/printf
CFLAGS += -DNEW_PRINTF_SEMANTICS

include $(MAKERULES)
```

Figura 25. Contenido del archivo Makefile de la base.

6.3.2 volumes-at45db.xml de la aplicación cliente

En este caso, además de los volúmenes definidos en el mismo archivo de la aplicación base, es necesario un volumen más, en el que se guardan los registros tomados por el GPS. Como se puede observar en la Figura 20, a este volumen se le ha asignado todo el espacio restante de la flash, es decir, 384 KB, el resultado de quitar 128 KB de los volúmenes de Deluge al espacio total de la memoria flash, 512 KB.

```
<volume_table>
  <volume name="GOLDENIMAGE" size="65536" base="0" />
  <volume name="DELUGE1" size="65536"/>
  <volume name="DELUGE2" size="0"/>
  <volume name="DELUGE3" size="0"/>
  <volume name="DATA" size="393216"/>
</volume_table>
```

Figura 26. Contenido del archivo volumes-at45db.xml de la base.

6.3.3 Archivo de cabecera de la aplicación cliente

Como en el archivo de cabecera de la estación base, se requiere alguna variable extra específica para esta aplicación. Estas variables son por un lado el tiempo de finalización de dos timers, el que controla el modo de fuera del nido y el que controla el final de la recepción de Heartbeats, y por el otro, la definición de una variable para definir el máximo de caracteres que puede recibir del componente del GPS.

6.3.4 Implementación de la configuración de la aplicación cliente

La aplicación que funcionará en la mota cliente, utiliza más funcionalidades que la aplicación base, como son el GPS, la memoria flash y la medida de la tensión. Por ello, son necesarios más componentes y por ende, habrá más conexiones. A continuación se explican los componentes y las conexiones que se añaden en esta implementación.

COMPONENTES

GPSC. Como su propio nombre indica, es el componente encargado de proveer las interfaces necesarias para poder utilizar el GPS.

LogStorageC. Este componente es el encargado de realizar la escritura y la lectura de memoria flash. A este componente hay que especificarle que volumen utilizará para guardar los registros, además de si los datos se guardarán en una cola circular o no.

VoltageC. Gracias a este componente se podrá medir la tensión que tiene la batería.

CONEXIONES

Las conexiones que se realizan en esta aplicación en comparación con la anterior, son las que se muestran en la siguiente figura.

```
proyectoC.LogRead -> Log;
proyectoC.LogWrite -> Log;
proyectoC.GPSData -> GPSC;
proyectoC.GPSInit -> GPSC.GPSInit;
proyectoC.Volt -> Volt;
```

Figura 27. Conexiones entre componentes

6.3.5 Implementación del módulo de la aplicación cliente

En la implementación del módulo de esta aplicación se aplica el mismo enfoque utilizado en su diseño: separar la aplicación en varios modos de funcionamiento de forma que sea más fácil explicar la forma en la que funciona. Los modos son los mismos que se han definido en el diseño, que son fuera del nido, dentro del nido y las transiciones de dentro a fuera y de fuera a dentro de la zona de anidamiento.

Con la finalidad de comprender mejor la marcha de la aplicación, se hace uso de varios gráficos. En estos gráficos se muestran la lógica que se sigue para conseguir un software que cumpla con los objetivos buscados. Se hace uso en varios momentos de algunos colores para remarcar varias situaciones especiales. Por un lado, hay un evento señalado con el color azul, que será el que inicie la aplicación. Los otros colores son el verde, que marca por qué evento empezará un modo de funcionamiento y el naranja, que es donde acabará un modo de funcionamiento.

FUERA DEL NIDO

Este es el modo por el que la aplicación empieza. Como en todas las aplicaciones de TinyOS, el inicio está marcado por el evento *booted* de la interfaz *Boot* del componente *MainC*. Desde el evento señalado por esta interfaz se inicia *TimerGPS* de forma periódica. Este timer es el que inicia cada una de las iteraciones de las operaciones que se ejecutan mientras se está fuera del nido.

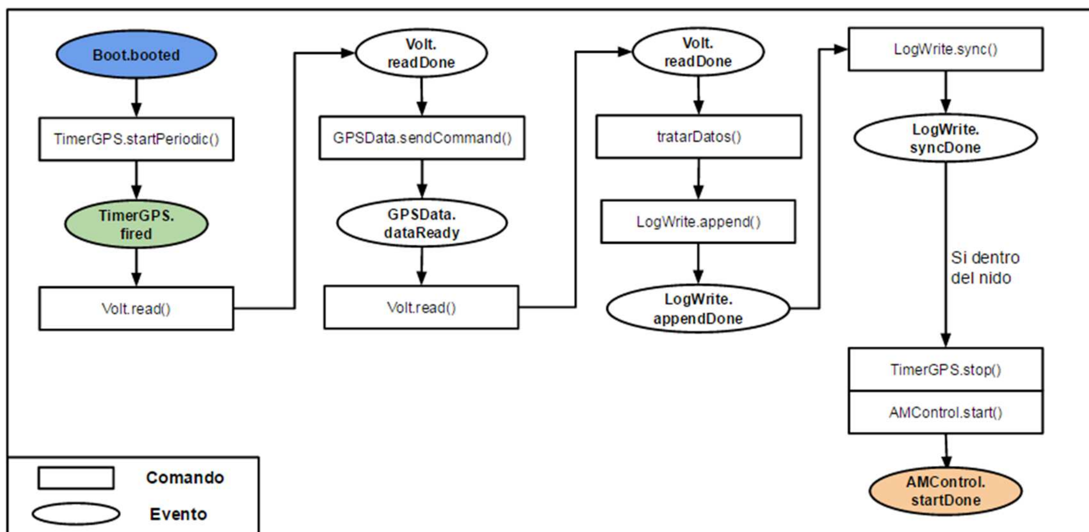


Figura 28. Implementación del modo fuera del nido.

Este proceso empieza cuando se señala el evento *fired* del timer, midiendo la tensión mediante la llamada a *read* de la instancia creada de *VoltageC*. Cuando se conozca la tensión de la batería, si es suficiente para hacer una medición de GPS, se hace dicha medida, enviando el comando necesario (*GPSData.sendCommand*). Cuando los datos sean recogidos y el evento *dataReady* señalado, se vuelve a medir la tensión de la misma forma que anteriormente, de forma que se pueda descartar una bajada de tensión que pudiese haber causado una mala lectura de los datos.

Si todo fuese correcto, se continua llamando a la función *tratarDatos()* creada para dar solución al algoritmo que coge la cadena de caracteres devuelta por el GPS, la acorta y la guarda en la estructura *log_t* que se utiliza en el proyecto. Desde esta función, se llama al comando *append* de la interfaz *LogWrite*, para que almacene el dato en la memoria flash. Si el guardado es correcto, se ejecuta el comando *sync* de la misma interfaz para asegurar que los datos han sido guardados.

Es en este punto donde se controla si el ave está dentro de la zona de anidamiento. En caso de detectar que está en la zona, se pasa a parar *TimerGPS*, para que no siga realizando mediciones. Sino, seguirá con otra iteración del bucle. Además de apagar el timer, también se inicia la radio de forma que se puedan recibir Heartbeats y asegurarnos de que realmente está al alcance de la estación base.

ENTRANDO AL NIDO

El objetivo de este modo es controlar si ha entrado dentro de la zona de acción de la base. El control se realiza no solo en este modo, sino que también en el modo dentro del nido. Este se realiza mediante el timer *TimerNoHB* y el evento *receive* de la interfaz con su mismo nombre, *Receive*. *TimerNoHB* está definido con un valor mayor al tiempo entre Heartbeats (30 segundos), de forma que si se señala el evento *fired* de este timer, significa que no se ha recibido ningún Heartbeat y que por lo tanto ha salido del radio de acción. Para que este método de control sea efectivo, cada vez que se reciba un Heartbeat, se tiene que reiniciar *TimerNoHB*, para que sólo llegue a su fin cuando no reciba nada de la estación base.

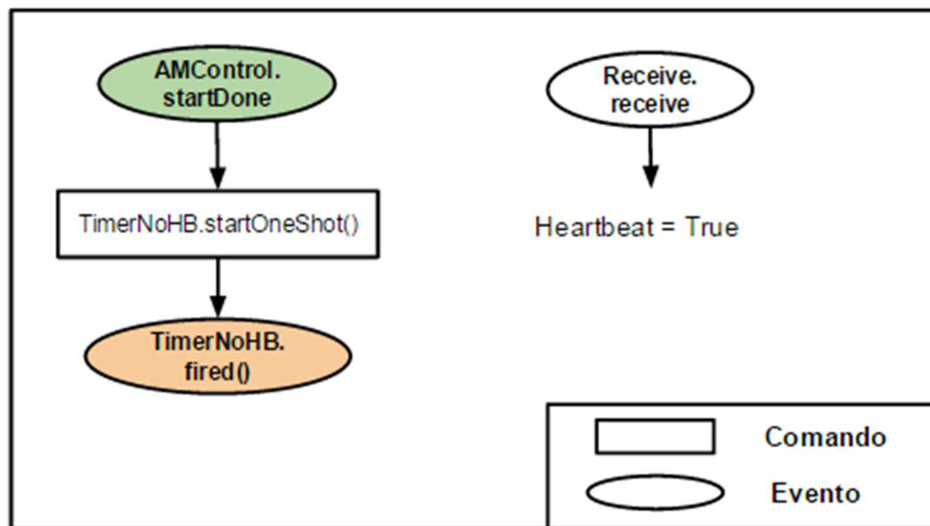


Figura 29. Implementación del modo entrando al nido.

DENTRO DEL NIDO

Como se ha explicado en el diseño de este modo, el dispositivo no ejecuta nada hasta que no reciba un Heartbeat con el número que concuerda con el ID de la mota. Aunque no sea el turno de descarga de la mota, siempre tendrá que reiniciar TimerNoHB, para que el sistema no lo interprete como que ha salido del nido.

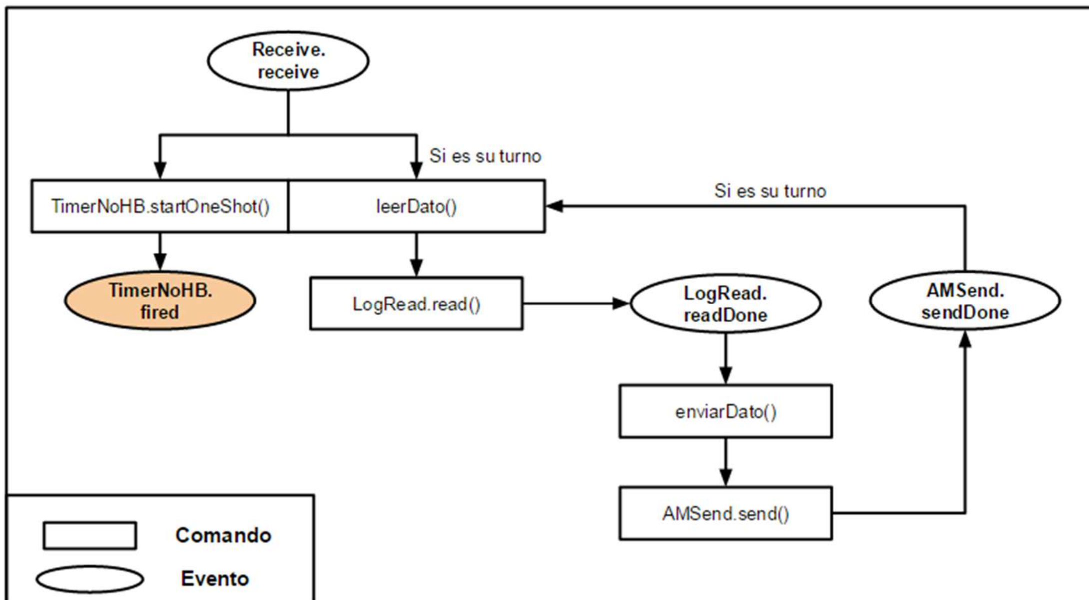


Figura 30. Implementación del modo dentro del nido.

Cuando se detecte que es el turno de descarga, se inicia un bucle, que iterará siempre y cuando sea el turno de descarga de la mota. Este bucle consta de dos funciones, la primera es la llamada *leerDato(),* que tiene como cometido controlar si hay más datos en la memoria flash y en caso positivo, sacar un dato. Para ésto, utiliza la llamada *read* de la interfaz *LogRead*. En la respuesta a este comando tenemos el evento *readDone*, en el que se hace uso de la segunda función, *enviarDato().* Como su nombre indica, esta función realiza las acciones necesarias para preparar los paquetes que se envían y los envía con el comando *send*. Cuando este comando se haya ejecutado y su evento correspondiente señalado, se vuelve a llamar a la función *leerDato()* empezando otra iteración del bucle.

SALIENDO DEL NIDO

A este modo se entra cuando se ha detectado que el ave ha abandonado la zona de anidamiento, es decir, cuando el evento *fired* de *TimerNoHB* es señalado. Antes de iniciar de nuevo el bucle definido en el modo fuera del nido, se apaga la radio para ahorrar batería. Esta acción se realiza llamando al comando *stop* de la interfaz *AMContro*l, que devuelve el evento *stopDone*. En la implementación de este evento se inicia de nuevo *TimerGPS*, pasando al estado fuera del nido.

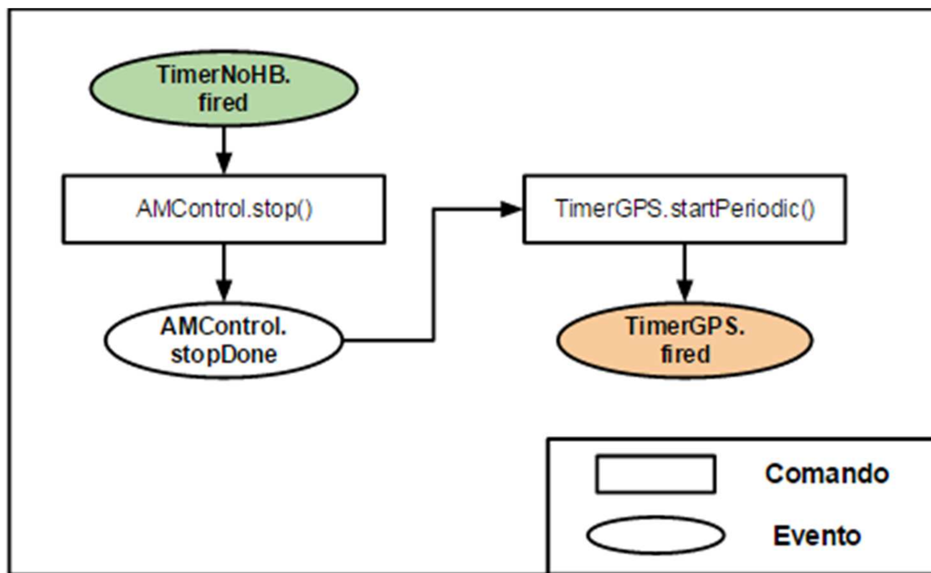


Figura 31. Implementación del modo saliendo del nido.

7 Prueba de las aplicaciones

Este capítulo está dedicado a las pruebas de funcionamiento de las aplicaciones que se han implementado en el capítulo anterior. Para ello, primeramente se expone la preparación de las prueba y se explica cuál es el procedimiento a realizar en las pruebas. Después, se analizan los errores que se han obtenido a la hora de probar las aplicaciones. Finalmente, se detalla la solución que se le ha dado a los errores aparecidos.

7.1 Preparación de las pruebas

Para las pruebas se dispone únicamente de tres motas, por lo que una se reserva como estación base y las otras dos se utilizan como nodos de la red de sensores inalámbrica. A pesar de que en el estudio de las gaviotas patiamarillas se vayan a utilizar veinte motas y en este caso sólo dispongamos de dos, no supone ninguna pérdida de fiabilidad en las pruebas. Esto ocurre gracias a que como está implementado, la ausencia de las otras dieciocho motas no influye en la red porque el control de las descargas esta implementado en la estación base que simplemente no recibe los datos de éstas motas aunque si las cuente para el algoritmo de repartición de franjas de tiempo para las descarga de datos.

Las pruebas se basan en la observación de los dos tipos de motas que hay, la estación base y las montadas en las aves. Para ello se han programado las tres motas disponibles, con un id diferente para cada una. Las identidades se reparten de la siguiente forma: del número 1 al número 20 se reservan para las motas cliente y el número 21 para la estación base. En estas pruebas, a las motas que hacen de nodos de la red se les asigna los id 1 y 2 y a la estación base, su correspondiente, el 21.

El análisis se realiza conectando las motas a un ordenador y visualizando los mensajes que estas transmiten utilizando la aplicación *Printf* que se le ha añadido a las aplicaciones realizadas.

7.2 Prueba de la estación base

En las pruebas de la estación base, se imprime por pantalla un mensaje periódicamente, cada vez que se envía un Heartbeat y que contiene el id de la motas a la que le toca descargar los datos. Estos mensajes tienen como objetivo conocer el turno de descarga y cerciorarse de que se envían correctamente los Heartbeats. En la Figura 32 se pueden observar los resultados.

```
Enviando Heartbeat(1)...OK!
Enviando Heartbeat(2)...OK!
Enviando Heartbeat(3)...OK!
Enviando Heartbeat(4)...OK!
Enviando Heartbeat(5)...OK!
Enviando Heartbeat(6)...OK!
```

Figura 32. Envío de los Heartbeats de la estación base.

7.3 Prueba de la aplicación cliente

En el caso de la aplicación cliente, se han incorporado varias llamadas a la función *printf* para saber qué acción está realizando la mota y conocer en qué modo se encuentra. Los mensajes que se imprimen en pantalla en cada modo son los siguientes:

- En el modo fuera del nido, tendremos un único mensaje, que muestra la medición de la tensión antes de la toma de la localización, los datos recogidos, la medición de la tensión después y si ha guardado el dato.
- En la transición de fuera a dentro del nido, se muestra cuando se inicia la radio y se pone el GPS en estado de Standby.
- Fuera del nido muestra si se ha realizado la lectura del dato de memoria, los datos leídos y si se ha enviado el dato.
- En la transición de dentro a fuera del nido, se muestra cuando se apaga la radio y se vuelve a iniciar el GPS.

Cuando se ha intentado visualizar estos mensajes, ha surgido un problema en la implementación, del cual se habla a continuación.

PROBLEMA ENCONTRADO

A la hora de realizar las pruebas ha surgido un problema con el GPS y la memoria flash. Este problema ha surgido cuando se han intentado almacenar los datos cogidos por el GPS en la memoria flash. El almacenado no ha sido posible.

Con motivo de conocer de dónde procede este problema se han realizado varias pruebas. Primero se ha estudiado la llamada *append* de la interfaz *LogWrite*, la encargada de almacenar en memoria los datos. Lo que se ha querido conocer es el error que devuelve la función y se ha observado que no devuelve ningún error, que simplemente no guardaba nada. Este hecho ha llevado a pensar que el problema es del GPS. Para estudiar el problema se ha intentado cambiar de lugar algunas de las llamadas a comandos de las interfaces utilizadas del GPS, como es el caso de *GPSInit.init*. Tras varios intentos, en uno de ellos se ha comentado esta función y se ha conseguido que la escritura en memoria flash funcionase. Gracias a esto, se ha conocido cual es el origen del error.

Conocido el origen del problema, se ha estudiado el funcionamiento de este comando y del componente que lo implementa, GPSC. Como conclusión al estudio, se ha conocido que el error viene de la transmisión de los comandos al GPS. El hecho de habilitar la posibilidad de envía comandos al GPS mediante la UART de la mota, crea una alteración en la escritura en memoria, no siendo posible hacer las dos acciones a la vez. Se sospecha que las líneas de la UART1 y la flash son compartidas. De forma que si se desea recibir datos del GPS y almacenarlos, se elimina la posibilidad de enviar comandos.

SOLUCIÓN DEL PROBLEMA

Ante la incapacidad de enviar comandos al GPS, se ha buscado una solución en la que sólo sea necesario recoger datos del GPS. Por ello no se ha podido configurar el GPS para que utilice el tipo de paquete One Sentence en el envío de los datos y se han utilizado los paquetes NMEA. De los mensajes que se reciben sólo se usan los de tipo GGA, que son los que traen la información de la hora, latitud y longitud.

Esta solución acarrea un problema consigo y es que no se puede enviar el comando necesario para que el GPS entre en esta de Standby, estando en todo momento cogiendo medidas. Esto hace que el consumo de batería se incremente, un gran inconveniente en este proyecto. Como el objetivo de este proyecto es realizar una primera versión del software que funcione y que pueda ser utilizado la primera vez que se programen las motas, este problema queda fuera del alcance del proyecto. Por ello, será necesario en un futuro realizar un estudio del hardware de la mota para ver si se encuentra la forma de solucionar el problema de no poder enviar comandos al GPS y almacenar datos en la memoria flash en una misma aplicación.

8 Conclusiones y trabajo futuro

A modo de reflexión final, se realiza una serie de comentarios que completan las conclusiones parciales obtenidas en los distintos capítulos. Asimismo, se apuntan posibles líneas de investigación y trabajo futuro.

En este proyecto se ha realizado una implementación de un sistema para el seguimiento de aves, en concreto la gaviota patiamarilla del Cantábrico, una necesidad de los ornitólogos de la Sociedad de Ciencias Aranzadi. El objetivo es, conocida la tecnología a utilizar, realizar una versión del software para la puesta en marcha del sistema y que pueda ir completándose a medida que el estudio de las aves vaya dando lugar a posibles modificaciones.

El sistema a realizar se divide en dos aplicaciones (estación base y nodos cliente) y deben cumplir ciertas especificaciones para cumplir con los objetivos. Entre las especificaciones está la recogida periódica de la posición donde las gaviotas han estado a lo largo del día. Además, está la necesidad de reprogramar con nuevas versiones de software las motas de forma remota.

En el transcurso de la implementación se ha encontrado un problema con la memoria flash y el GPS, y es que si se desean enviar comandos al GPS, no se puede almacenar datos en la memoria flash. Por este problema se han tenido que hacer ciertas modificaciones en la implementación realizada. A pesar de estos cambios, se ha podido realizar las aplicaciones necesarias para el sistema de localización de aves.

La conclusión general de este trabajo es que se ha implementado una primera versión del software requerido y que esta versión cumple con las especificaciones pedidas por la Sociedad de Ciencias de Aranzadi. No obstante, quedan cuestiones a resolver.

El trabajo posterior a la implementación del software que se presenta en este trabajo de fin de grado tendría diferentes fases. Primero, hay que realizar un estudio del hardware para determinar a qué se debe la influencia entre el GPS y la memoria flash. Por otro lado, hay que construir la estación base y desarrollar el sistema de comunicación remota entre la estación base y un ordenador, para la recogida automática de datos, gestión de las versiones del software, etc. Además, para el estudio biológico que se pretende, hay que instalar la estación base en la colonia, y encapsular los dispositivos para su posterior colocación en las aves. Finalmente, hay que tomar la localización exacta de la estación base, con motivo de saber cuál es el rango de la zona de acción de la radio.

BIBLIOGRAFÍA

Redes de sensores inalámbricas y motas IRIS:

- WSN (http://en.wikipedia.org/wiki/Wireless_sensor_network)
- IRIS (http://www.memsic.com/userfiles/files/datasheets/wsn/iris_datasheet.pdf)

Sistema operativo TinyOS:

- Documentación, tutoriales (http://www.tinyos.net/)
- Wikipedia (http://en.wikipedia.org/wiki/TinyOS)

Lenguaje de programación NesC:

- Manual (http://nescc.sourceforge.net/)
- Wikipedia (http://en.wikipedia.org/wiki/NesC)

Dispositivo GPS:

- Datasheet (http://www.adafruit.com/datasheets/GlobalTop-FGPMMOPA6H-Datasheet-V0A.pdf)
- GlobalTop One Sentence (http://www.gtop-tech.com/en/software/Software_Services_19.html)
- Comandos PMKT (http://www.adafruit.com/datasheets/PMTK_A08.pdf)