PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

DISEÑO DE PROTOCOLOS PARA REDES INALÁMBRICAS DE SENSORES SOBRE LATINOS: METODOLOGÍA Y EJEMPLOS

SANTIAGO BARROS ARTEAGA

Tesis para optar al grado de
Magíster en Ciencias de la Ingeniería

Profesor Supervisor:
CHRISTIAN OBERLI

Santiago de Chile, Enero 2013

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

DISEÑO DE PROTOCOLOS PARA REDES INALÁMBRICAS DE SENSORES SOBRE LATINOS: METODOLOGÍA Y EJEMPLOS

SANTIAGO BARROS ARTEAGA

Tesis presentada a la Comisión integrada por los profesores:

CHRISTIAN OBERLI
MARCELO GUARINI
DIEGO DUJOVNE
JAIME NAVÓN

Para completar las exigencias del grado de
Magíster en Ciencias de la Ingeniería

Santiago de Chile, Enero 2013

A mi familia y amigos

AGRADECIMIENTOS

Quisiera agradecer en primer lugar a mi familia por haberme formado como persona y por su apoyo incondicional durante toda mi vida.

Quisiera agradecer también a mi profesor supervisor Christian Oberli por su confianza durante todo este proceso de formación académica y por darme la oportunidad de realizar mis estudios de magíster en el Laboratorio de Tecnologías Inalámbricas (LATINA). En especial quisiera agradecerle por sus "tres mandamientos", que orientaron la forma de trabajo durante este tiempo y que sin duda perdurarán.

Por último quisiera agradecer a mis compañeros: Camila Muñoz, Fernando Rosas, Francisco Reyes, Carlos Feres, Joaquín Aldunate y Jean Paul de Villers Grandchamps por su constante ayuda y apoyo tanto en lo emocional como en lo científico.

INDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

RESUMEN

Una red inalámbrica de sensores (RIS) es un conjunto de dispositivos electrónicos, llamados nodos inalámbricos, que tienen como objetivo observar algún fenómeno distribuido en el espacio y variante en el tiempo. Para lograr entregar al usuario la información observada, es necesario que los nodos que forman parte de la red cuenten con un protocolo de comunicaciones que coordine la transmisión de datos entre nodos y la canalización de estos hacia un nodo acumulador.

Los protocolos de comunicaciones sufren de una constante evolución y diversificación debido a permanentes avances tecnológicos y diversidad de usos de las RIS. La necesidad de modificar o crear nuevos protocolos, ha sido abordada mediante la utilización los sistemas operativos para RIS, que facilitan al usuario la integración de dichos protocolos al resto del sistema.

Un nuevo sistema operativo llamado “LatinOS”fue recientemente desarrollado en el laboratorio en que se realizó este trabajo de tesis, pero a pesar de las prestaciones que lo hacen atractivo frente a otros sistemas operativos para RIS, carece por sí solo de protocolos de comunicaciones integrados a él.

En este trabajo se describe una metodología para el desarrollo de protocolos bajo LatinOS que permite un proceso ágil y simple. Utilizando esta metodología se implementa un protocolo para monitoreo ambiental y un protocolo de *benchmarking* para la medición de alcance de comunicaciones entre nodos de una RIS. Ambos constituyen aportes en diseño e implementación de protocolos y habilitan a este sistema operativo para ser utilizado en aplicaciones reales.

Palabras Claves: redes inalámbricas de sensores, metodología de desarrollo de protocolos, implementación de protocolos

ABSTRACT

A Wireless sensor network (WSN) is a set of electronic devices, known as wireless nodes, that aim to observe some phenomena distributed in space and variable in time. To deliver observed data to the user, all nodes belonging to the network must have and use a communications protocol that coordinates the data transmission to gather all data into a sink node.

The ever changing wireless communication technologies and the infinite number of possible uses of WSNs require a constant evolution and diversification of communication protocols. For this reason operating systems come to the aid of WSNs by providing the user with services that simplify the creation of new protocols that easily meet these constantly changing requirements.

A new operating system called LatinOS has been recently developed. It offers a series of new features that make it attractive compared to other operating systems. However, there are currently no protocols implemented in LatinOS and neither is there a methodology to develop them in this new enviroment as this new operating system is at an early stage.

This work describes a methodology to simplify and streamline the communication protocols development using LatinOS. Using this methodology, two protocolos are implemented and deployed: an environmental monitoring protocol and a benchmarking protocol for measuring wireless communications maximum range between nodes within a network. Both provide new contributions to the design and implementation of protocols under LatinOS, enabling this operating system to be used in real applications.

Keywords: wireless sensor networks, protocols design methodology, WSN protocol design, WSN protocol implementation

1. INTRODUCCIÓN

1.1. Contexto

En las últimas décadas, la proliferación de dispositivos con capacidad de comunicación inalámbrica ha sido explosiva. Ejemplos son los teléfonos celulares, PDAs y *smartphones*, los que mediante la integración de elementos electrónicos permiten comunicaciones a altas tasas de datos con una autonomía energética de varios días.

Recientemente, estos avances de la tecnología han permitido el desarrollo de nodos inalámbricos. Se trata de dispositivos con una capacidad de procesamiento y tasa de datos reducida en relación a los *smartphones* que cuentan con un microcontrolador, un transceptor (que permite la comunicación inalámbrica con otros dispositivos), una batería y una memoria (ver Figura 1.1), a los que se le agregan sensores específicos. El bajo costo y la reducción dramática del consumo energético en relación a otros dispositivos móviles permiten que estos nodos sean desplegados en cualquier tipo de terreno o ambiente para formar una Red Inalámbrica de Sensores (RIS).

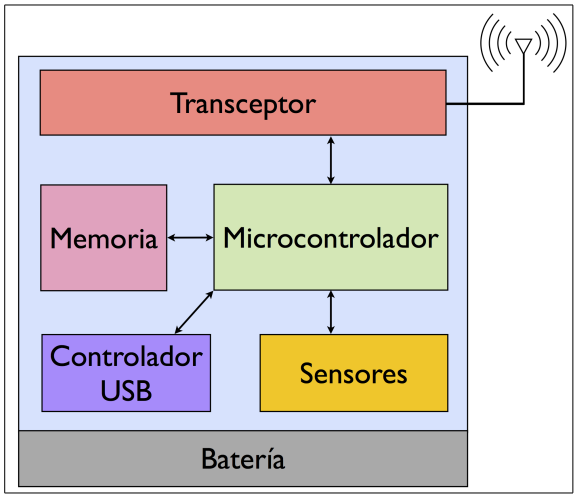

FIGURA 1.1. Arquitectura típica de un nodo inalámbrico. El microcontrolador controla a los periféricos (transceptor, memoria, sensores, USB) e implementa los protocolos de comunicaciones que le permiten comunicarse con el resto de la red.

Una RIS es un conjunto de nodos inalámbricos que se comunican entre sí en forma coordinada para cumplir una función específica, como el monitoreo de temperatura y plagas en un bosque, la humedad del suelo en un predio agrícola o el caudal de agua a lo largo de un río entre otros. Los nodos inalámbricos miden variables de interés del entorno y los envían inalámbricamente a través de la red en saltos de un nodo a otro hacia un nodo acumulador que reporta la información obtenida al usuario.

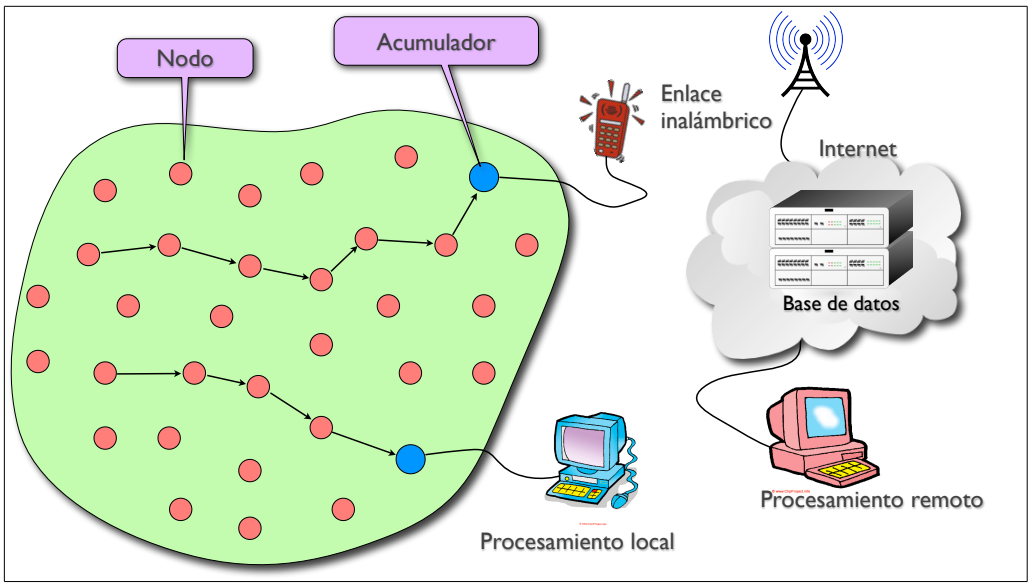

FIGURA 1.2. Red inalámbrica de sensores. El transporte de datos desde los nodos sensores hacia el acumulador ocurre mediante una ruta de múltiples saltos de un nodo a otro. Los datos pueden ser procesados localmente en un acumulador o ser desaguados hacia una base de datos para ser procesados remotamente

Actualmente, varios modelos de nodos inalámbricos se encuentran disponibles en el mercado, y cada uno de ellos ofrece variadas prestaciones acorde a los distintos microprocesadores y periféricos que utilizan. En la Figura 1.3 es posible ver algunos modelos disponibles comercialmente.

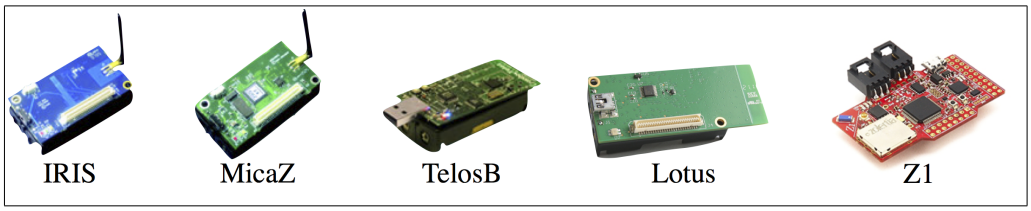

FIGURA 1.3. Modelos de nodos inalámbricos disponibles en el mercado. Estos modelos son comercializadas por MEMSIC (*MEMSIC Wireless Modules*, 2010) (IRIS, MicaZ, TelosB y Lotus) y por Zolertia (*Zolertia Wireless Node*, 2011)

En esta tesis todos los experimentos y pruebas se llevan a cabo en el nodo Z1 (*Zolertia Wireless Node*, 2011). Tiene integrados un sensor de temperatura y un acelerómetro, un transceptor, una antena interna, un microcontrolador y una memoria *flash* (ver Figura 1.4).

El nodo Z1 utiliza un microcontrolador de la familia MSP430 diseñado para redes de sensores por su bajo consumo energético. Además el nodo Z1 utiliza el transceptor CC2420 (*Texas Instruments CC2420 Tranceiver*, 2012) que permite realizar transmisiones inalámbricas con un alcance de hasta 200 m utilizando una antena externa.

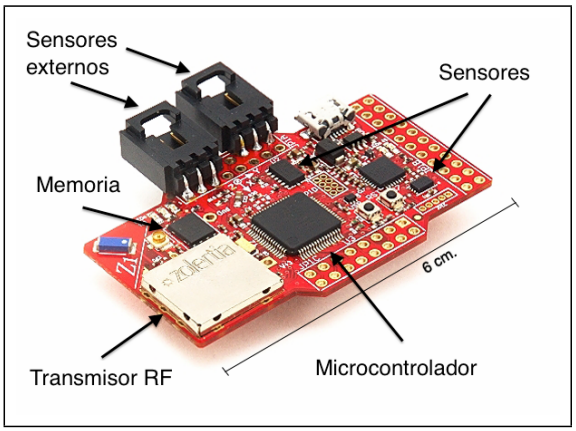

FIGURA 1.4. Nodo inalámbrico Z1. Plataforma utilizada para experimentos y pruebas fabricada por Zolertia

Para que los nodos puedan operar, el microprocesador debe ejecutar un pequeño programa llamado *firmware*, que consiste en un set de instrucciones y reglas que controlan el funcionamiento del nodo.

Como parte del *firmware*, se encuentra el protocolo de comunicaciones o *protocol stack*, que consiste en un sub-conjunto de instrucciones del *firmware* relacionadas a la comunicación con otros nodos y que definen el comportamiento y función de cada uno de ellos dentro de la RIS. Estas reglas e instrucciones son comunes a todos los nodos de la red.

El *firmware*, además de contener el *protocol stack*, contiene también los *drivers*, que corresponden a instrucciones que permiten configurar el microcontrolador y controlar sus periféricos (transceptor, memoria, sensores, etc).

Para facilitar el desarrollo del *firmware* y optimizar los recursos disponibles de cada nodo, es muy común el uso de un sistema operativo, que modulariza el firmware en bloques de código reutilizables evitando grandes cambios de código para para pequeños cambios en el comportamiento del nodo. Por otro lado, los sistemas operativos entregan librerías con los *drivers* para configurar y operar los periféricos específicos de cada nodo disponible y que se utilizan independientemente de la aplicación de la RIS requerida.

Entre los sistemas operativos existentes para RIS está LatinOS, un sistema operativo relativamente nuevo, que tiene la ventaja de permitir simular en un computador de escritorio el mismo código que se ejecuta en el nodo como *firmware* permitiendo la detección temprana de errores de código y una simulación del comportamiento de la red completa frente a distintos escenarios de despliegue. Otras prestaciones y ventajas son analizadas con profundidad en el trabajo de (Campamá, 2012).

1.2. Motivación y problema

A pesar de que la arquitectura de LatinOS fue diseñada para alojar múltiples *protocol stacks*, LatinOS no consta de ningún *protocol stack* implementado todavía, lo que impide por ahora desplegar una RIS. Además, debido a que se trata de un sistema operativo nuevo, no existe una metodología formal ni un manual práctico de cómo implementar un *protocol stack* en LatinOS, de tal forma de asegurar su correcto funcionamiento bajo el entorno de programación y arquitectura que este sistema operativo utiliza.

El foco de esta tesis está en encontrar una forma reproducible de desarrollar e implementar una gran variedad de *protocol stacks* en el sistema operativo LatinOS procurando aprovechando la arquitectura modular con que este sistema fue diseñado.

1.3. Objetivos

El objetivo general de este trabajo es habilitar a LatinOS para ser utilizado en aplicaciones reales. Para ello, deben cumplirse los siguientes objetivos específicos:

1. Elaborar una metodología para la implementación de protocolos sobre LatinOS.
2. Acondicionar LatinOS para facilitar la integración de nuevos protocolos.
3. Validar esta metodología implementando protocolos funcionales.
4. Desplegar una RIS en terreno utilizando LatinOS como sistema operativo.

1.4. Organización del documento

Esta tesis está estructurada como se indica a continuación. El capítulo 2 corresponde a una descripción del estado del arte, relativo al desarrollo de protocolos, la estructura interna de un *protocol stack* y una descripción de la arquitectura y funcionamiento de LatinOS. En el capítulo 3 se propone una metodología para el desarrollo de protocolos sobre LatinOS, la que es validada en los capítulos 4 y 5 mediante dos ejemplos reales. Finalmente, en el capítulo 6 se presentan las conclusiones obtenidas a partir del trabajo realizado.

2. MARCO DE REFERENCIA

En este capítulo se introducen los conceptos fundamentales que permiten explicar con claridad el trabajo de esta tesis y sus aportes.

Las RIS son conjuntos de nodos capaces de monitorear un fenómeno distribuido en el espacio y tiempo. El elemento importante que convierte a un conjunto de nodos sensores en una red, es la capacidad de mover información entre nodos y desaguar datos a través de un nodo acumulador. A menudo es necesario usar nodos intermedios para transferir la información en multiples saltos al acumulador.

Las comunicaciones que ocurren en forma directa entre nodos se denominan comunicaciones de 1-*hop*. "Nodos vecinos"son aquellos capaces de realizar este tipo de enlaces. Las comunicaciones que utilizan a uno o más nodos como "puente"para comunicar información a otro nodo fuera de alcance, son llamadas comunicaciones *multi-hop*.

2.1. Modelo de capas de un protocolo de comunicaciones para RIS

Las reglas que regulan el tráfico de datos entre nodos son denominadas e implementadas como un "*protocol stack*".

Para poder entender el funcionamiento del protocolo de comunicaciones, es necesario entender cómo esta estructurado. El *protocol stack* puede subdividirse, en cuatro capas (Charfi, Masmoudi, y Derbel, 2009), que están dispuestas según lo indica la Figura 2.1. Éstas se comunican entre ellas utilizando funciones de intercambio de datos llamadas "interfaces".

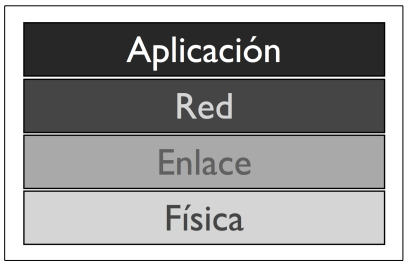

FIGURA 2.1. Modelo de capas para RIS. Las capas del *stack* de protocolos se subdividen asignando funcionalidades específicas a cada capa. El usuario accede al sistema a través de la capa de Aplicación (capa superior). Los datos enviados desde la aplicación son procesados por las capas de Red y Enlace hasta llegar a la capa Física para salir al aire y viceversa.

la capa Física (PHY) Las comunicaciones entre nodos vecinos (*1-hop*) están controladas por y la capa de Enlace. Esta última se subdivide en dos subcapas: MAC (*Medium Access Control*) y LC (*Link Control*). La subcapa MAC controla el acceso al medio inalámbrico, que es el canal de comunicación que comparten todos los nodos. La subcapa LC se encarga de administrar los enlaces de comunicacion entre nodos vecinos. Esto incluye labores como el control de sanidad de paquetes recibidos y la implementación de paquetes de confirmación de transmisiones, llamados *ACK*.

Una vez provisto el enlaces entre los nodos vecinos (mediante la operación articulada de las capas PHY y Enlace) la capa de Red posibilita la operación coordinada de la red entera (estableciendo enlaces *multi-hop*). Ello se logra mediante algoritmos de enrutamiento que aseguran la transferencia de información entre cualquier par de nodos y cualquier cantidad de saltos.

Finalmente, la capa de Aplicación administra los datos relevantes para el usuario. Ello incluye labores como el sensado de datos, la compresión y almacenamiento de éstos y de reportar dichos datos al usuario final.

2.1.1. Sobre la implementación de protocolos para RIS en sistemas operativos

No existen trabajos relacionados al desarrollo de protocolos para LatinOS, debido a ser un sistema operativo desarrollado recientemente.

En el trabajo de (Thouvenin, 2007) se detalla la implementación de un protocolo particular sobre el sistema operativo TinyOS, sin proponer una metodología formal para el desarrollo de protocolos. Por otra parte, en el trabajo de (He, 2007) se realiza una tarea similar, dado que se implementa otro protocolo particular sobre el sistema operativo Contiki, nuevamente sin una propuesta formal de metodología de desarrollo. Esta falta de metodología dificulta la implementación y la experimentación de nuevos *protocol stacks* en un sistema operativo determinado, por lo que los trabajos citados aportan solamente al despliegue de RIS y no al desarrollo e integración de nuevos protocolos a los sistemas operativos existentes.

Debe notarse que la forma en que se desarrollan protocolos para uno u otro sistema operativo no son necesariamente la mismas, dado que cada uno tiene una arquitectura de sistema distinta. De ahí surge la necesidad de contar con una metodología formal para el desarrollo de protocolos sobre LatinOS, de forma de guiar paso a paso al desarrollador en la implementación de un nuevo *protocol stack* para este sistema operativo.

En la literatura asociada a facilitar la implementación de protocolos, existe una herramienta que permite implementar algoritmos de capa MAC (*Medium Access Control*) bajo el sistema operativo TinyOS utilizando una interfaz gráfica (Ansari, Zhang, Salikeen, y Mahonen, 2011). Entre los protocolos de capa MAC que pueden ser implementados se encuentran B-MAC, X-MAC, CSMA, Aloha y S-MAC. El desempeño de éstos es probado comparativamente por (Huang, Xiao, Soltani, Mutka, y Xi, 2012). Si bien esta herramienta presenta una forma muy atractiva de desarrollar protocolos, ésta abarca solamente una sub-capa del *protocol stack*, lo que la hace inviable para soluciones que necesiten de protocolos de capa cruzada.

2.2. LatinOS

LatinOS es un sistema operativo desarrollado con el fin de integrar la simulación de una RIS y su despliegue en terreno, en un mismo programa que implementa *protocol stack*. LatinOS proporciona un entorno de programación intuitivo y modular, facilitando

al usuario la interacción con los periféricos de cada nodo y entregando herramientas genéricas para la implementación del *protocol stack* sin importar el *hardware* subyacente de cada nodo.

La arquitectura que utiliza LatinOS para crear el *firmware* está conformada principalmente por cuatro módulos: Plataforma, Protocolo, Aplicación y Sistema Operativo. El diagrama de interacción entre los bloques se muestra en la Figura 2.2.

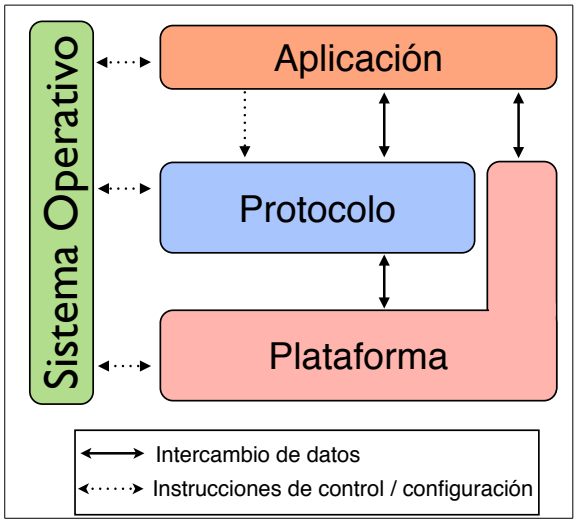

FIGURA 2.2. Módulos de LatinOS que conforman el *firmware*. El módulo Plataforma contiene los *drivers* que configuran y controlan el microcontrolador y sus periféricos sobre los que opera el firmware. El módulo Protocolo, implementa la mayor parte del *protocol stack* necesario para las comunicaciones y el módulo Aplicación se encarga de tomar y recibir datos útiles para el usuario final. El módulo Sistema Operativo provee funciones que permiten integrar los otros tres bloques.

El módulo Plataforma es el que contiene los *drivers* que configuran y controlan el *hardware* del nodo.

El módulo Protocolo implementa las funcionalidades que permiten el funcionamiento de las comunicaciones en red.

El módulo Aplicación es aquel que controla la toma de datos de los nodos sensores y su transmisión. Además se ocupa de recibir e interpretar los datos en el nodo acumulador.

Para realizar dicha tarea, utiliza a los módulos inferiores (Protocolo, Sistema Operativo y Plataforma).

El módulo Sistema Operativo, provee el control del módulo Plataforma y coordina la interacción con los módulos Protocolo y Aplicación mediante interfaces estandarizadas. Sin embargo, no cumple ninguna función en la implementación del *protocol stack*.

Las distintas capas del *protocol stack* son implementadas por los módulos de LatinOS como se muestra en la Figura 2.3 y el intercambio de datos entre módulos es realizado utilizando interfaces predefinidas por LatinOS (Campamá, 2012).

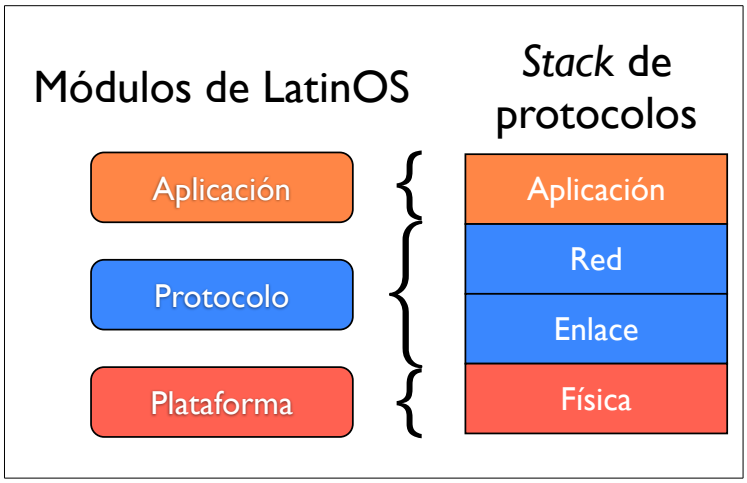

FIGURA 2.3. Asignación de funcionalidades del *stack* de protocolos en LatinOS. El módulo Protocolo implementa las capas de Red (comunicaciones *multi-hop*, direccionamiento, enrutamiento y topología), y Enlace (acceso al medio, confirmación de paquetes, detección de errores). Las capas de Aplicación y Física son implementadas por los módulos Aplicación y Plataforma, respectivamente.

2.2.1. Flujo de ejecución del Sistema Operativo

El diagrama de flujo principal del *firmware* a bordo de los nodos está basado en tres etapas (ver Figura 2.4):

1. **Inicialización de la plataforma:** Configuración del microcontrolador, de sus periféricos y de los sensores del nodo.
2. **Inicialización de módulos Sistema Operativo, Protocolo y Aplicación:** Inicialización del reloj principal y herramientas principales del sistema operativo,

definición de las direcciones físicas de cada nodo (ID) y de parámetros generales de funcionamiento del *protocol stack*. En caso de que el nodo sea el nodo acumulador, deberá ejecutar algunas instrucciones específicas que lo configuran como tal.

3. **Ejecución de tareas y eventos:** El procesador entra en modo de bajo consumo y despierta solamente para procesar eventos y tareas. Las tareas son las labores que un nodo debe ejecutar en forma secuencial según prioridad y orden de llegada a la lista de tareas. Éstas son agregadas por los bloques Aplicación, Protocolo y Sistema Operativo. Los eventos son interrupciones al flujo normal de ejecución y que requieren que el microprocesador ejecute rutinas específicas de forma inmediata. Algunos ejemplos de eventos que generan interrupciones pueden ser la llegada de un paquete desde la radio, un sensor que activa una alarma o simplemente que el reloj interno debe ser actualizado cada segundo para mantener la hora.

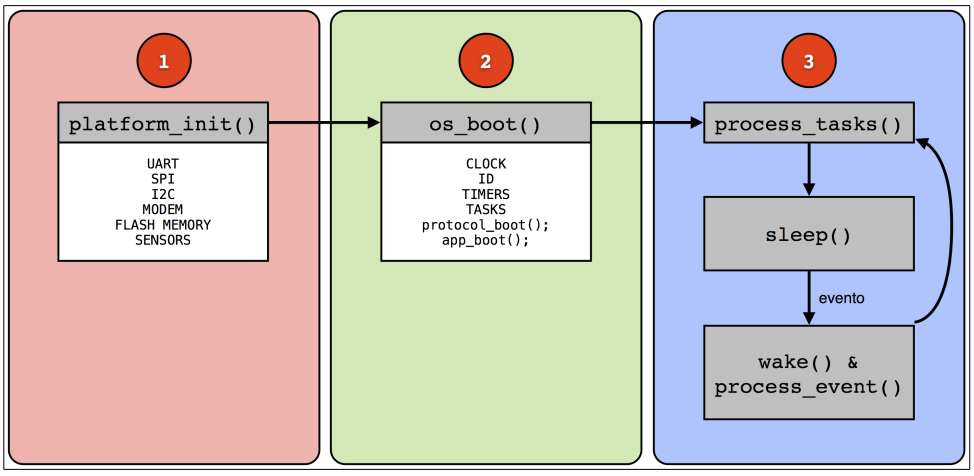

FIGURA 2.4. Diagrama de flujo principal de LatinOS. 1.- Inicialización de la plataforma (periféricos e interfaces de comunicación genéricas). 2.- Inicialización de módulos del sistema operativo (reloj, *timers*, tareas e identificación) y del *stack* de protocolos. 3.- Flujo normal del programa.

2.3. Transmisión y recepción de datos en LatinOS

El intercambio de información entre nodos inalámbricos se realiza a través de paquetes de información. Los paquetes pueden pertenecer al módulo Aplicación o al módulo Protocolo. El flujo de armado y procesamiento de paquetes se ilustra en la Figura 2.5.

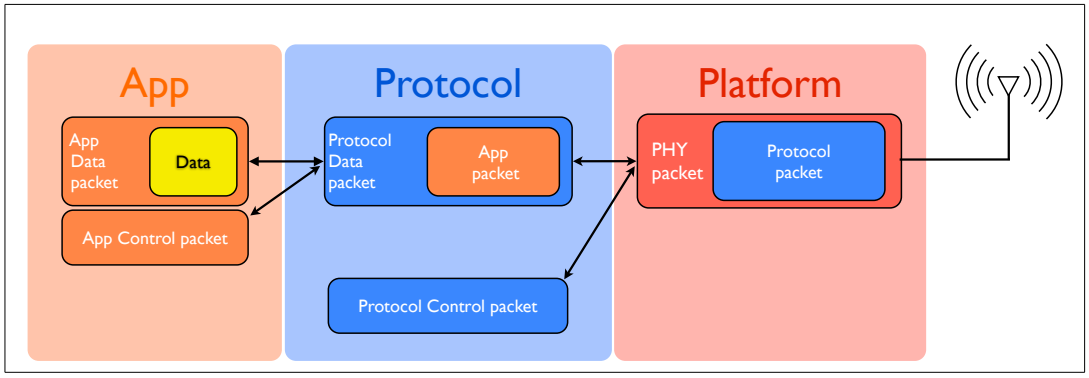

FIGURA 2.5. Esquema de armado y procesamiento de paquetes en los módulos de LatinOS. Cada módulo arma paquetes con los datos propios y los entrega a módulo inferior inferior para su encapsulado y transmisión. En la recepción se procesa el paquete internamente y si existe un paquete encapsulado dentro, éste se envía a la capa superior.

Cada paquete puede contener datos propios y datos de un módulo superior en forma de un paquete encapsulado. Todos los paquetes que se envían y reciben deben pasar por el módulo Plataforma para ser acondicionados o procesados respectivamente. Así, la labor de armado de paquetes en el caso de transmisión, es entregar los datos del nivel propio a los módulos inferiores hasta llegar al módulo Plataforma para ser acondicionados para salir al aire. En el caso de recepción, el módulo debe procesar los datos del nivel propio y entregar el resto al módulo superior.

Cuando un paquete no encapsula a otro perteneciente a un módulo superior, es considerado como paquete de control del módulo correspondiente.

2.3.1. Funcionamiento del módulo Protocolo

Durante el flujo normal del programa existen tres tipos de rutinas que gatillan el funcionamiento del protocolo:

1. **Recepción de paquetes del módulo Plataforma:** Cada vez que un paquete es recibido desde el aire, es guardado en un *buffer* del módulo Plataforma y procesado por el módulo Protocolo. Si el paquete contiene información que le corresponde al módulo Aplicación el procesamiento de esa sección de información se realiza en dicho módulo. Terminada la operación, el paquete se elimina del *buffer*.
2. **Paquetes provenientes desde el módulo Aplicación:** Los paquetes son procesados por el módulo Protocolo y entregados al módulo plataforma para su transmisión por el medio inalámbrico.
3. **Operaciones internas gatilladas por el módulo Sistema Operativo:** En algunas ocasiones, como por ejemplo esperar un paquete de confirmación o encender la radio durante una ventana de tiempo para transmisión/recepción de datos, el módulo Protocolo requiere realizar acciones en forma asíncrona a la recepción o transmisión de paquetes. En estos casos, el módulo Protocolo debe generar *timers* que serán gatillados por el módulo Sistema Operativo y ejecutados por el módulo Protocolo cuando corresponda.

El diseño del encapsulado de los datos provenientes del módulo Aplicación de LatinOS, así como la definición de los paquetes de control y su procesamiento, son algunos de los aspectos que deben ser diseñados en el desarrollo de protocolos de comunicaciones. La metodología desarrollada en esta tesis contempla estos aspectos.

3. METODOLOGÍA DE IMPLEMENTACIÓN DE PROTOCOLOS

La metodología para la implementación de protocolos sobre LatinOS tiene 5 etapas que son detalladas en este capítulo:

1. Especificación funcional del *stack* de protocolos.
2. Diseño de paquetes.
3. Diseño de flujo del módulo Protocolo.
4. Implementación de diagramas de flujo.
5. Simulación y despliegue.

La metodología propuesta tiene por objetivo evitar errores en cada etapa de modo de no arrastrar los errores de una etapa a la siguiente. Ello acelera el proceso de diseño e implementación de un *protocol stack*.

3.1. Especificación funcional del *stack* de protocolos

Como fue explicado en la figura 2.3, el módulo Protocolo de LatinOS implementa solo un subconjunto del *stack* de protocolos. Por esta razón, es necesario separar las funcionalidades que corresponden al módulo Protocolo de aquellas que no. Ello luego permite listar aquellas funcionalidades y servicios que el módulo Protocolo debe prestar a la red. Por ejemplo, algunos de los servicios del *protocol stack* que puede requerir una RIS son:

- Sensado de canal para control de acceso al medio.
- Regulación de ciclos de trabajo de la radio.
- Confirmación de paquetes recibidos (ACK).
- Detección y/o corrección de errores en los datos.
- Direccionamiento de paquetes.
- Descubrimiento automático de vecinos.
- Manejo de la topología de la red.
- Sincronización entre nodos.

- Segmentación o agregación de datos.
- Procesamiento o compresión de datos *on-net*.
- Enrutamiento variable o fijo.
- Capacidad de *streaming* de datos.
- Control de congestión de paquetes.

Varios de los servicios listados arriba podrían tener requerimientos específicos de *hardware* y/o *software* para funcionar correctamente. Por ejemplo, si es necesario implementar sensado de canal antes de transmitir un paquete, el módem y sus *drivers* deben tener habilitada alguna función que ejecute esta operación y entregue el resultado. Por lo tanto, si hay protocolos que requieran servicios que exceden las capacidades de la plataforma a utilizar, estos servicios deben ser modificados en esta etapa, ya que de lo contrario estos errores se podrían detectar solamente una vez desplegada la red en terreno.

3.2. Diseño de paquetes

Para gatillar distintos comportamientos en un nodo, como por ejemplo, retransmitir datos al acumulador o propagar un paquete de sincronización de tiempo, el módulo Protocolo utiliza los "tipos de paquete". Esto responde a la necesidad de la red de tener un lenguaje común entre los nodos. Para ello, LatinOS utiliza el mismo principio de estructura y procesamiento de paquetes presentado por (Buonadonna, Hill, y Culler, 2001). Éste plantea que cada tipo de paquete desencadena la ejecución de un set de instrucciones específico en el nodo, de acuerdo a la información que cada paquete contiene.

La definición de los tipos de paquetes que son utilizados, la información que contiene cada uno de ellos y cómo deben procesarse al ser recibidos por el módulo Protocolo de LatinOS es labor de quien diseña e implementa el protocolo.

Cada paquete del módulo Protocolo debe tener una cabecera, en adelante *header*, que contiene al menos dos campos: `packet_type` y `destination`, correspondientes al tipo de paquete y destinatario del mismo. Con estos dos parámetros es posible procesar por

separado los tipos de paquetes que son recibidos por el nodo y saber el destinatario de cada mensaje y procesarlo acorde a esta información. Cada parámetro ocupa 1 byte de memoria permitiendo como máximo 256 nodos y 256 tipos de paquete, aunque es posible utilizar 2 o más bytes para aumentar la cantidad máxima de nodos y/o tipos de paquete, según lo requiera la red. No obstante, el *header* de paquete puede tener más campos que los dos enunciados arriba como por ejemplo, identificadores únicos de paquete, el origen del paquete, etc.

El *header* contiene los campos de datos que son comunes a todos los tipos de paquete de un mismo *protocol stack*. Cualquier set de datos que el módulo Protocolo necesite para procesar un tipo de paquete específico debe ir dentro del campo de "carga útil", también conocido como *payload*. El tamaño de esta carga útil puede ser distinto para cada tipo de paquete, acorde a las información requerida para el procesamiento. Por ejemplo, si el paquete fuera de datos, el *payload* correspondería a un paquete del módulo Aplicación, y si fuera un paquete de sincronización, el *payload* debiera contener la hora a la cual debe sincronizarse el nodo.

En general, también se agrega a cada paquete una cola, conocida como *footer*, que sirve para incluir un chequeo de sanidad del paquete utilizando algún código de detección y/o corrección de errores.

En la Figura 3.1 puede apreciarse la estructura genérica de aquellos paquetes que recibe el módulo Protocolo desde el módulo plataforma vistos en la Sección 2.3.

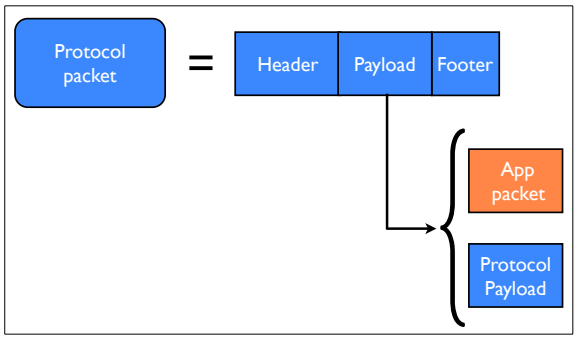

FIGURA 3.1. Estrucura de paquetes del módulo Protocolo. Los paquetes contienen un *header* y *footer* comunes y un *payload* que puede tener un paquete del Módulo Aplicación dentro y/o información de control para el módulo Protocolo, dependiendo del tipo de paquete.

3.3. Diagrama de flujo del módulo Protocolo

La elaboración de un diagrama de flujo del funcionamiento del módulo Protocolo de LatinOS facilita la escritura de código y filtra errores asociados a diferencias entre los servicios requeridos por la especificación técnica y las funcionalidades que entrega el diagrama de flujo.

Para simplificar la elaboración de este diagrama de flujo, se puede subdividir la acción del protocolo en tres procesos principales:

1. Recepción de paquetes (gatillada por el módulo Plataforma).
2. Transmisión de paquetes (gatillada por el módulo Aplicación).
3. Ejecución de *timers* y tareas (gatillados(as) por el módulo Sistema Operativo).

3.3.1. Recepción de paquetes

En este proceso se determina el tipo de paquete y el destinatario y se ejecuta la rutina correspondiente a ese tipo de paquete.

Como ejemplo, se presenta en la Figura 3.2 el comportamiento de un nodo desplegado para el monitoreo en un bosque ante la llegada de un paquete de alarma de incendio.

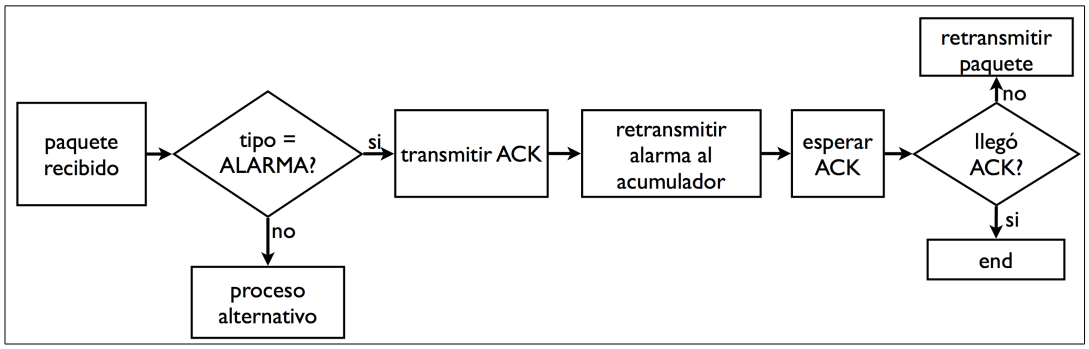

FIGURA 3.2. Ejemplo de diagrama de flujo. Al recibir un paquete el módulo Protocolo detecta el tipo de paquete para ejecutar la rutina que le corresponda. Luego contesta con una confirmación en forma inmediata, reenvía el paquete al acumulador y espera a que vuelva la confirmación de esa última transmisión. Si el paquete de confirmación no llega luego de un cierto tiempo, el paquete es retransmitido.

3.3.2. Llegada de un paquete o instrucción desde el módulo Aplicación

En este paso, es necesario especificar la forma en que se llenan los campos agregados al paquete de aplicación y funciones que deben implementarse.

Una vez que los paquetes están armados, deben ser entregados a la función de transmisión de datos de la plataforma para que salgan al aire. Antes de enviar el paquete, es posible implementar un *buffer* u otro tipo de proceso intermedio que la especificación funcional requiera.

A modo de ejemplo se ilustra, en la Figura 3.3, el comportamiento del protocolo ante la llegada de un paquete desde la aplicación que debe ser enviado al nodo acumulador.

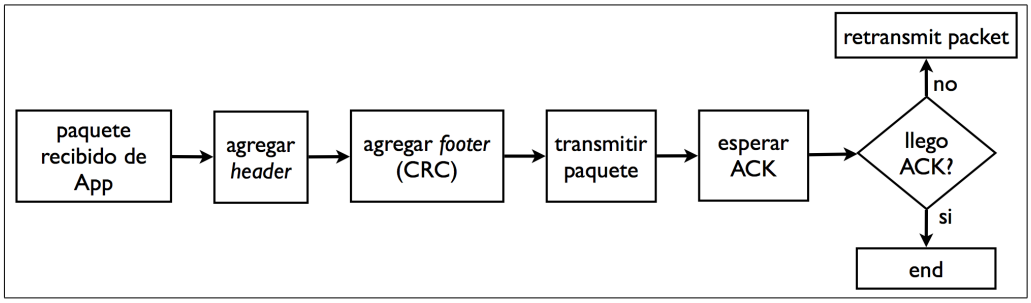

FIGURA 3.3. Procesamiento de paquetes que ingresan desde el módulo Aplicación. El módulo Protocolo debe armar un paquete con el *header*, el paquete del módulo Aplicación y el *footer* y transmitirlo a través del módulo Plataforma

3.3.3. Eventos del protocolo gatillados por el módulo Sistema Operativo

Cada vez que alguno de los módulos de LatinOS agrega un *timer* o una tarea que hace referencia a una función del módulo Protocolo (u otro módulo), el módulo Sistema Operativo gatilla su ejecución a través de lista de tareas en el tiempo correspondiente. Por ejemplo, cuando el módulo Protocolo debe esperar un lapso de tiempo para recibir la confirmación (ACK) de algún paquete enviado, agrega un *timer* que le permite realizar otras tareas mientras ese tiempo se cumple. De la misma forma, el módulo Aplicación puede agregar *timers* para sensar datos en intervalos de tiempo determinados. Para ambos casos el nodo puede entrar en un modo de bajo consumo y gatillar los timers cuando corresponda.

3.4. Implementación de diagramas de flujo

La implementación en código del diagrama de flujo del paso anterior implica que el programa debe ejecutar ciertas rutinas de procesamiento. Estas rutinas pueden ejecutarse mediante tres clases de instrucciones:

- **Rutinas *inline***: Son ejecutadas dentro de la función, y son utilizadas para instrucciones que deben realizarse con alta prioridad, como por ejemplo la sincronización del reloj.
- **Tareas diferidas**: Son tareas que se ejecutan después de un espacio de tiempo mediante el uso de *timers* del módulo Sistema Operativo. Estas tareas pueden ser canceladas si no se ha comenzado la ejecución.
- **Tareas inmediatas**: Son rutinas idénticas a las tareas diferidas con la particularidad de que el tiempo de espera es nulo. Son agregadas a la lista de tareas y se ejecutan inmediatamente una vez terminada la función en ejecución, según la prioridad definida y orden de llegada.

El Código 3.1 (escrito en C) muestra un ejemplo de implementación en LatinOS para el diagrama de flujo presentado en la Figura 3.2.

```
void protocol_receive(*packet){
        if(packet_type == ALARM) {
        //send acknowledgement back to sender
        ack_transmit(packet.sender);
        //forward alarm to basestation in 0 seconds
        timer_add_with_priority(&forward_alarm,0,PRIORITY_HIGH);
        //if no ack received in 1 sec
        //retransmit packet
        timer_add(&retransmit_alarm, 1_SEC);
        }
}
```

CÓDIGO 3.1. Ejemplo: El paquete recibido debe confirmar inmediatamente la llegada del paquete al nodo emisor (rutina *inline*), enviar el paquete (tarea inmediata) y retransmitirlo en caso de que no llegue confirmación de entrega del nodo receptor (tarea diferida).

3.5. Simulación y despliegue

El sistema operativo LatinOS desarrollado por (Campamá, 2012) tiene una capacidad integrada de simulación, lo que permite encontrar fallas en la implementación del protocolo antes de la etapa de despliegue. Esto es recomendado cuando es costoso (ya sea en términos de tiempo y/o dinero) desplegar la red en terreno o ensayarla en un laboratorio y permite distinguir fallas relativas al protocolo de aquellas relativas a la plataforma.

Para aprovechar esta capacidad de simulación de LatinOS, es necesario compilar el código para una plataforma virtual llamada `testnode_sim` (nodo de simulación). Así el producto de la compilación (un archivo binario) es ejecutable por el simulador de LatinOS. La configuración del simulador está descrita en el trabajo de (Campamá, 2012). Los detalles técnicos de cómo compilar para plataformas reales y virtuales, así como también la configuración del simulador se detallan en el Anexo A.3.

Luego de realizar las simulaciones del código y corregir los eventuales errores asociados al *protocol stack* implementado en el módulo protocolo, es posible compilar el código para ser utilizado en el nodo inalámbrico.

El hecho de compilar el código para un nodo real, implica que existen nuevas fuentes de errores o diferencias de ejecución en relación a la simulación. Éstas diferencias son inherentes a una plataforma que tiene capacidades de procesamiento limitadas (la velocidad de procesador es tres órdenes de magnitud más lento que la un computador de escritorio, y la memoria es del orden de kilobytes en comparación con los gigabytes de un PC). Estas limitaciones revelan una necesidad de probar el protocolo implementado en los nodos en un laboratorio, antes de desplegarlo en terreno para detectar y corregir comportamientos difieren de la simulación realizada y de la especificación funcional.

Esta metodología exige que las pruebas de laboratorio sean realizadas antes de cualquier despliegue porque es la forma más rapida y confiable de chequear el funcionamiento del protocolo ejecutado dentro de la misma plataforma en que será desplegado. El hecho de probar el correcto funcionamiento de la red en el laboratorio permite detectar fallas asociadas a la plataforma que no pueden detectarse en la etapa de simulación debido a la limitación del simulador de emular todas las diferencias entre un nodo inalámbrico y un computador de escritorio. De esta forma los nodos se encuentran accesibles en todo momento para ser monitoreados, diagnosticados y reprogramados para dar solución a las posiblas fallas que estos pudieran sufrir.

Una vez que los nodos han sido probados en el laboratorio y arreglados los eventuales errores de implementación o ajustes necesarios del protocolo que surgen debido las diferencias entre el nodo inalámbrico y un computador de escritorio, es posible desplegar la red en terreno.

4. PROTOCOLO DE MONITOREO AMBIENTAL

En este capítulo se presenta la implementación de un *protocol stack* para el monitoreo ambiental. La implementación de este *protocol stack* fue diseñada para LatinOS utilizando la metodología propuesta en el capítulo anterior.

La implementación del *protocol stack* de monitoreo ambiental está basado en el protocolo *Sensorscope* (Barrenetxea et al., 2008). Este protocolo consiste en la concentración de datos hacia un nodo acumulador, mediante la creación de una red cuya topología es auto-configurada y actualizada a cada instante. La descripción del funcionamiento se encuentra completamente detallada en el documento citado, sin embargo cabe destacar que algunas de sus características han sido modificadas con el fin de mejorar el desempeño general y/o con el propósito de simplificar el funcionamiento del protocolo bajo LatinOS.

4.1. Especificación funcional del *protocol stack* de *Sensorscope*

La especificación funcional y técnica del *protocol stack* de *Sensorscope* es utilizada como base, pero algunas modificaciones son realizadas en relación a:

- La inicialización y sincronización de la red.
- Manejo de enlaces con vecinos.
- Ahorro de energía mediante comunicaciones intermitentes utilizando ciclos de trabajo temporales en el transceptor.
- Modificación del *header* de paquetes.

Siguiendo la especificación de *Sensorscope* (Barrenetxea et al., 2008) sumado a las modificaciones listadas anteriormente y renunciando a los servicios de capa MAC que prestaba TinyOS, que es el sistema operativo sobre el cual fue desarrollado *Sensorscope*, el protocolo a implementar debe prestar las siguientes funcionalidades:

- **Topología de red dinámica y autoconfigurada:** La red debe autoconfigurar su topología con el reconocimiento y mantención automática de enlaces con nodos vecinos.
- **Sincronización de tiempo:** Es una funcionalidad fundamental para RIS cuyos nodos que deben marcar los datos sensados con un identificador de tiempo común a la red y también para permitir la ejecución de procesos coordinados en toda la RIS como por ejemplo, ciclos de actividad/inactividad de los transceptores. Esta sincronización debe tener una precisión dependiente de la aplicación, que en este caso (monitoreo ambiental) corresponde a una precisión de segundos para el estampado de tiempo y de milisegundos para las acciones coordinadas de la RIS. Además, debe permitir que nodos que pierden el sincronismo, vuelvan a adquirirlo.
- **Ciclos de trabajo del transceptor:** El trasceptor es el periférico que más energía consume cuando está encendido, por lo tanto, para prolongar la autonomía de cada nodo (y de la red), es imprescindible que se disponga de un mecanismo que permita mantenerlo apagado sin sacrificar conectividad. Para esto, el *protocol stack* debe operar con ventanas de tiempo para la transmisión de datos y mantener el transceptor apagado el resto del tiempo.
- **Confirmación de paquetes:** Cada vez que un paquete de datos es recibido exitosamente, el nodo receptor debe enviar una confirmación al nodo emisor para evitar que éste lo retransmita.
- **Detección de errores:** Todos los paquetes transmitidos por la red (ya sean de datos o control) deben ser creados y procesados utilizando un sistema de revisión de sanidad para asegurar que el procesamiento de paquetes se realice sobre un paquete sin errores.
- **Enrutamiento:** Debe estar basado en criterios de calidad de enlace, cercanía y conectividad con el nodo acumulador.

- ***Buffering* de paquetes a transmitir:** Es posible que no todos los paquetes sean transmitidos inmediatamente después de ser creados. Por ello, es necesario tener la capacidad de guardar esos paquetes para enviarlos cuando sea posible.

4.2. Diseño de paquetes

Las distintas funciones de comunicación que podrían requerir de tipos de paquete propios son:

- Inicialización.
- Sincronización.
- Confirmación de paquetes.
- Datos de aplicación.

No obstante, por requerimientos de topología y direccionamiento, todos los paquetes deben tener el encabezado de protocolo descrito en la Figura 4.1.

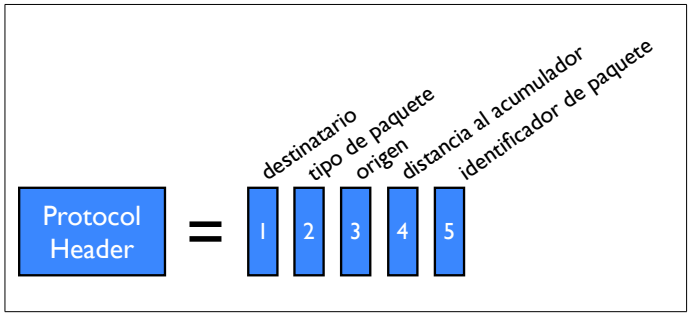

FIGURA 4.1. *Header* de paquetes del protocolo de monitoreo ambiental. Consta de 5 bytes de información. El *destinatario* es la dirección de red a la que el paquete se envía, y permite al nodo receptor descartar paquetes que no le corresponden. El *tipo de paquete* indica qué rutina debe ejecutarse para procesar el paquete entrante. Los campos de *origen*, *distancia al acumulador* e *identificador de paquete* son necesarios para formar y mantener una topología y una lista de vecinos que permita un enrutamiento de paquetes eficiente.

Además del *header*, algunos paquetes contienen información extra. Específicamente:

- Los paquetes de inicialización, llamados `HELLO`, no llevan información extra además de la cabecera.

- Los paquetes de sincronización (llamados `TIME_SYNC`) llevan el tiempo propio que tenía el nodo transmisor al momento de enviar el paquete (previa sincronización de este último).
- Los paquetes de confirmación (llamados `ACK`) llevan un identificador que permite al receptor conocer a qué paquete enviado corresponde esa confirmación.
- Datos de aplicación.

La figura 4.2 muestra la estructura y contenido de los 4 tipos de paquete implementados. Tal como se indica en la especificación funcional de la sección 4.1, todos los paquetes llevan también un chequeo de sanidad, que se implementa agregando un *footer* con un CRC (chequeo de redundancia cíclica). Este CRC permite identificar si hay errores en los datos de algún paquete.

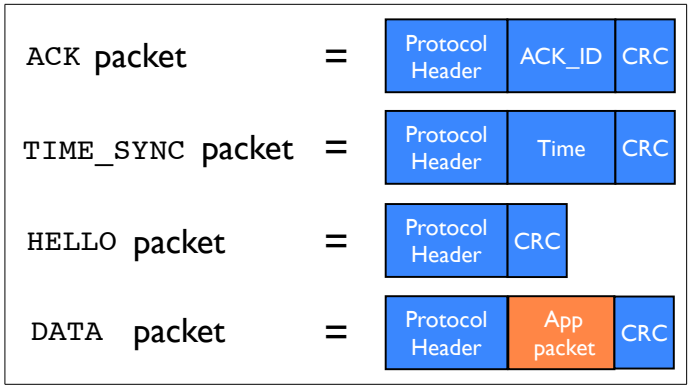

FIGURA 4.2. Diseño de paquetes para protocolo de monitoreo ambiental. Todos los paquetes comparten un *header* y un *footer* con los mismos campos de datos.

4.3. Diagrama de flujo del módulo Protocolo

En esta sección se describe el diagrama de flujo general del módulo Protocolo y se describen los procesos de recepción de paquetes desde el módulo Plataforma, transmisión de paquetes del módulo Aplicación y eventos gatillados por el módulo Sistema Operativo (ver Sección 3.3).

En la Figura 4.3 se muestra el flujo que siguen todos los paquetes desde que son armados hasta que son procesados por el nodo receptor.

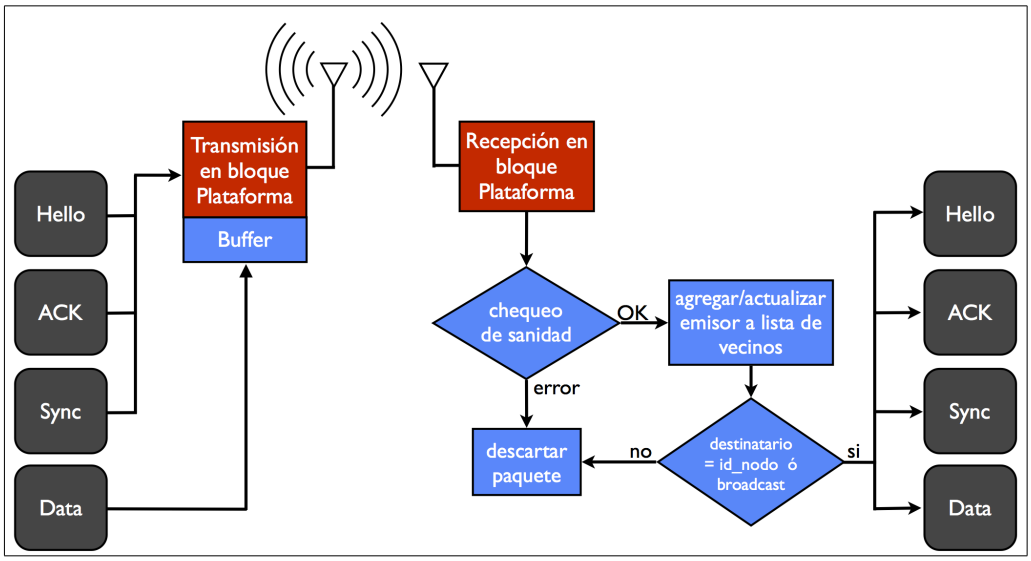

FIGURA 4.3. Diagrama general de flujo de paquetes. Arquitectura de transmisión y recepción de datos en el protocolo de monitoreo ambiental.

4.3.1. Inicialización

Para poder inicializar la red, es necesario que los nodos desplegados se encuentren a la escucha de paquetes válidos en el aire. Si un nodo detecta un paquete en el aire (independiente del destinatario) registra la información del *header* para establecer un enlace válido.

Para poder armar una topología que permita el desague de los datos hacia el nodo acumulador, inicialmente este último transmite un mensaje de tipo `HELLO` que es recibido por todos los nodos que se encuentan a *1-hop* de distancia. Éstos a su vez, retransmiten este mensaje a los nodos que se encuentran a *2-hop*, y así sucesivamente, generando una "avalancha" conocida como *flooding* en que todos los nodos que tengan conectividad con el acumulador (aunque sean enlaces *multi-hop*) reciben y procesan paquetes de tipo `HELLO` de todos sus nodos vecinos. Estos paquetes contienen la información estrictamente necesaria para generar una topología de red válida. Para evitar la retransmisión innecesaria de paquetes de tipo `HELLO`, una vez que un nodo ha recibido y retransmitido un paquete de este tipo, no vuelve a retransmitirlo nuevamente.

4.3.2. Paquetes de confirmación

Cada vez que se envía un paquete que requiere de confirmación del destinatario, se configura un *timer* de tiempo `WAKE_FOR_ACK_TIME` (en milisegundos) predefinido en el archivo de encabezado del módulo Protocolo. Si el paquete de confirmación no llega en ese lapso de tiempo, se ejecuta la función `no_ack()` (ver figura 4.4), la que retransmite el paquete. Esto se repite hasta 3 veces, luego de lo cual `no_ack()` procede a penalizar la calidad del enlace con el vecino correspondiente (ver Lista de vecinos en la sección a continuación). Si el paquete de confirmación es recibido a tiempo, entonces se cancela el *timer* y la función `no_ack()` no se ejecuta.

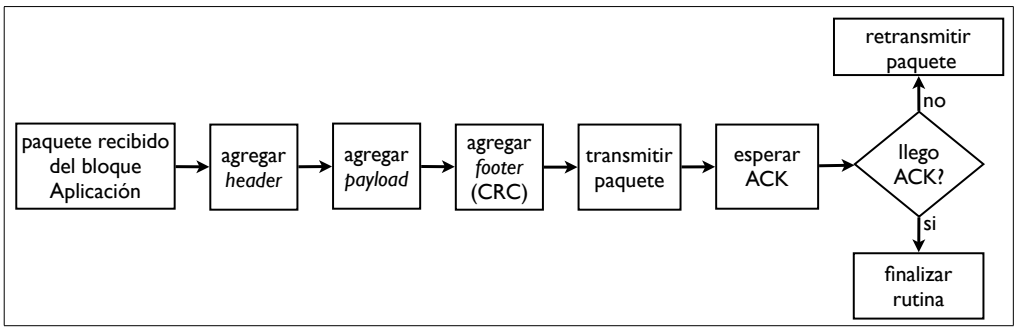

FIGURA 4.4. Diagrama de confirmación de paquetes. Flujo que sigue el transmisor después de enviar un paquete que requiere de confirmación o `ACK`.

4.3.3. Lista de vecinos

Cada nodo maneja una lista de vecinos que se utiliza para mantener indicadores de calidad y confiabilidad de los enlaces que cada nodo tiene con sus vecinos (*1-hop*). Cada entrada de la lista (1 entrada por vecino) tiene asociada la siguiente información:

1. Dirección de red (igual a la dirección física).
2. Distancia al acumulador medida en *hops*.
3. Número de paquetes de confirmación no recibidos de este nodo.
4. Número de paquetes recibidos correctamente de este nodo.
5. *ID* del último paquete recibido de este nodo.
6. Calidad de enlace calculado a partir de los 3 ítemes anteriores.

El manejo de esta lista se realiza en forma idéntica a *Sensorscope*, con la excepción de que los enlaces caducan (dejan de ser enlaces válidos reinicializando los indicadores de ese vecino en particular) por fallas consecutivas en la confirmación de paquetes enviados a ellos, a diferencia de una caducidad gatillada por tiempo (Barrenetxea et al., 2008).

4.3.4. Ciclos de trabajo del transceptor

Como no es factible mantener durante demasiado tiempo el transceptor de un nodo a la escucha de paquetes (por limitaciones de consumo energético), se hace necesario que para lograr el objetivo impuesto por la especificación funcional, las rutinas de *sleep/wake* del transceptor sean coordinadas para asegurar que las ventanas de tiempo coincidan. Las ventanas de tiempo creadas para transmitir datos deben ser lo suficientemente grandes como para que todo el tráfico requerido se lleve a cabo en forma satisfactoria. Dado que las tasas de datos de esta aplicación de RIS son relativamente bajas, el tiempo en que el transceptor debe encontrarse encendido es también bajo, alcanzándose un alto ahorro de consumo energético.

Durante las ventanas de tiempo en que la red no transmite paquetes, es necesario que los datos obtenidos de los sensores sean almacenados hasta el proximo ciclo de desague de datos. Esto con lleva la implementación de un buffer de paquetes.

En la sección a continuación se explica la forma coordinada de gatillar estos ciclos de trabajo del transceptor, enunciando antes el algoritmo de sincronización utilizado en esta implementación.

4.3.5. Sincronización de tiempo

Los algoritmos de sincronización para redes de sensores tienen dos objetivos principales. El primero busca posibilitar que distintos nodos de la red puedan ejecutar acciones en forma coordinada, como por ejemplo, asignar ventanas de tiempo en los que cada nodo puede realizar transmisiones para evitar colisión de paquetes tarea que requiere de una precisión del orden de milisegundos. El segundo objetivo de la sincronización es

marcar los datos sensados con la fecha y hora en la que fueron tomados. Esto requiere una precisión que depende de la aplicación, que puede relajarse al orden de segundos.

Para lograr sincronizar múltiples nodos dentro de una red, el principal problema que buscan resolver los distintos algoritmos presentes en la literatura, es determinar el tiempo que transcurre entre que un nodo informa su observación de tiempo a otro nodo y éste último lo procesa (ver Figura 4.5).

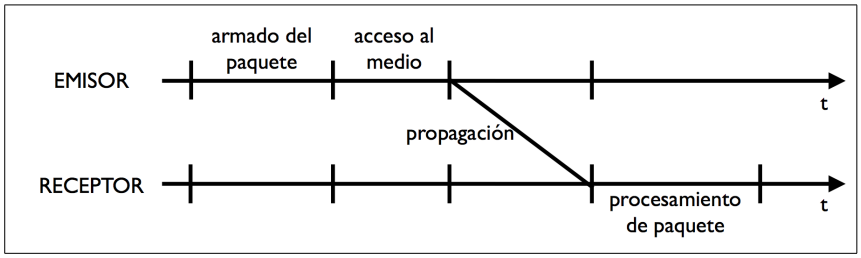

FIGURA 4.5. Tiempos relevantes para la sincronización entre nodos. Entre que un nodo informa su tiempo al otro, transcurren los lapsos de armado de paquete, acceso al medio, propagación y procesamiento. El nodo receptor solamente puede conocer el instante del último de ellos

Los distintos algoritmos de sincronización para RIS transan entre la complejidad del algoritmo y la precisión de la sincronización. Entre estos algoritmos se encuentran ETA (Kusy et al., 2006) y protocolos como RBS, FTSP, TPSN y LTS, detallados y comparados en el trabajo de (Sivrikaya y Yener, 2004).

En esta sección se presenta un algoritmo de sincronización con un enfoque similar a LTS (Gruenen y Rabaey, 2003), en el que se busca la simplicidad del algoritmo por sobre una exhaustiva búsqueda de precisión. En vez de utilizar tres paquetes para lograr una sincronización de tiempo entre dos nodos, este nuevo algoritmo utiliza solamente uno, lo que conlleva una pérdida en la precisión pero permite disminuir la cantidad de mensajes y el tiempo de propagación y por ende el tiempo necesario para sincronizar la red completa.

Este algoritmo propuesto sincroniza los relojes de todos los nodos para permitir incluir información de fecha y hora a los datos tomados por los sensores.

En este *protocol stack* es posible aprovechar la rápida propagación de los mensajes `TIME_SYNC` (mediante la ténica de *flooding* o "avalancha"), para gatillar el inicio de los ciclos de trabajo de transceptor, permitiendo que las rutinas para iniciar los ciclos de trabajo del transceptor se ejecuten con un desfase similar al tiempo que demora un nodo en armar un paquete y enviarlo.

A continuación se detalla el funcionamiento de la sincronización de relojes para el estampado de tiempo de los datos y luego de la implementación de los ciclos de trabajo del transceptor.

Al recibir el paquete de sincronización desde el acumulador, cada nodo debe actualizar su reloj, y gatilla el incio de los ciclos de trabajo del transceptor. Hecho esto, retransmite el paquete a sus nodos vecinos vía *broadcast* (ver Figura 4.6). Para evitar que los nodos que ya fueron sincronizados vuelvan a hacerlo (por la posible recepción de retransmisiones de nodos cercanos), antes de procesar un paquete recibido, cada nodo debe revisar si alguna sincronización fue hecha recientemente. De ser así, el nuevo paquete de `TIME_SYNC` se ignora, a menos que venga del nodo acumulador.

Toda sincronización tiene un tiempo de vigencia llamado `sync_timeout`, definido como parámetro del *protocol stack* y reiniciado luego de cada sincronización. Al cumplirse este tiempo sin recibirse un nuevo paquete de sincronización, el protocolo evita que cualquier proceso apague el transceptor hasta encontrar algún paquete que le permita re-sincronizarse.

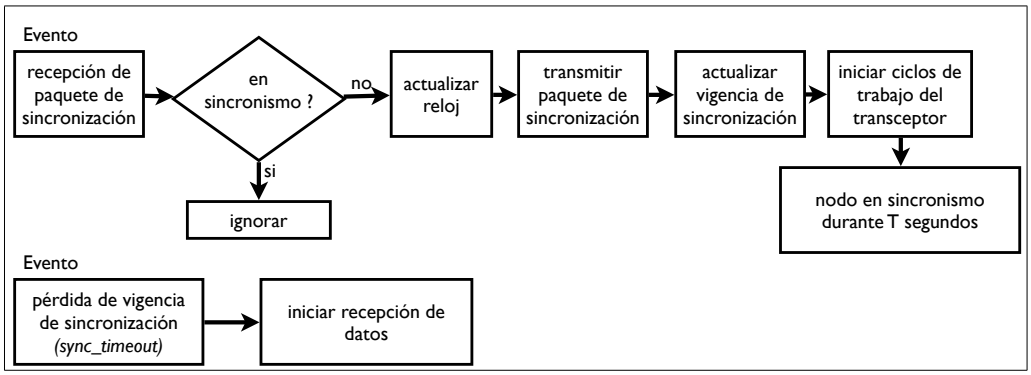

FIGURA 4.6. Diagrama de recepción de paquetes de sincronización. Al llegar un paquete de sincronización, se actualiza el reloj, se transmite un paquete de sincronización al resto de los nodos y luego se actualiza el tiempo de vida de esa sincronización. Para finalizar, el nodo ejecuta un *timer* que bloquea la entrada de paquetes de sincronización durante `T` segundos. Al perder sincronismo el protocolo mantiene encendido el transceptor esperando un nuevo paquete para volver a sincronizarse.

La figura 4.7 muestra el diagrama de flujo que describe las acciones que debe tomar un nodo cuando se gatillan los ciclos de trabajo del transceptor.

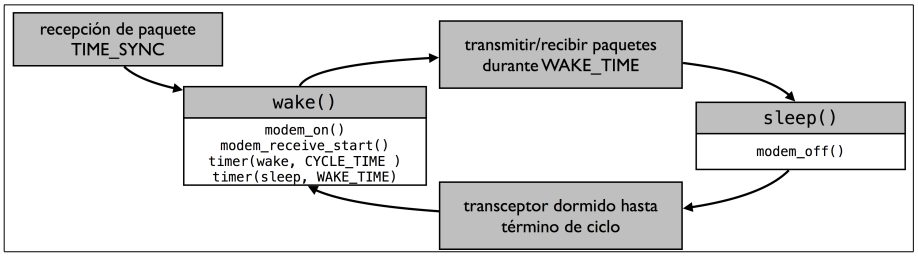

FIGURA 4.7. Ciclo de trabajo de la radio. El algoritmo funciona utilizando *timers* para gatillar el encendido y apagado del transceptor. Inicialmente el módulo Protocolo inicia el ciclo de trabajo del transceptor junto con la propagación del paquete de sincronización (`TIME_SYNC`).

4.4. Implementación de diagramas de flujo

La implementación en código C de los diagramas de flujo descritos en la sección anterior son insertados en el módulo Protocolo de LatinOS acorde a ciertas reglas de implementación (específicas para LatinOS) enunciadas en los Anexos A.1 y A.2. Luego todos los módulos de LatinOS se compilan utilizando la herramienta *open-source*

MSPGCC (*GCC toolchain for MSP430*, 2012) para la plataforma Z1 y utilizando *GCC* (*General C Compiler*, 2012) para el nodo virtual de simulación `testnode_sim`.

4.5. Simulación y despliegue en terreno

Tanto en simulación como en despliegue, se utilizan los siguientes parámetros de funcionamiento de la red:

Bloque	**Parámetro**	**Valor**
Protocolo	`WAIT_FOR_ACK_TIME`	300 ms
	Tamaño *buffer* de transmisión	20 paquetes
	Límite de vecinos por nodo	20
	Máxima distancia al acumulador	15 *hops*
	`WAKE_TIME`	10 s
	`SLEEP_TIME`	6 min
	`sync_timeout`	25 min
Sistema Operativo	*Buffer* de recepción	20 paquetes
	Tamaño máximo de paquetes	120 bytes
	Cola máxima de tareas	30
	Cola máxima de *timers*	12

La simulación fue realizada con el objetivo de depurar el código y el funcionamiento del protocolo (asegurar que los datos llegaran al nodo acumulador). Sin embargo, para la depuración de aspectos relativos a diferencias de *hardware* fue necesario realizar pruebas en el laboratorio utilizando nodos reales.

Dado que la simulación es ejecutada en una arquitectura de *hardware* distinta a la arquitecura de un nodo inalámbrico, no es posible detectar errores asociados a estas diferencias en la etapa de simulación. Ejemplos de esto son:

- **Imprecisiones relaciondas a tareas coordinadas:** Esto se debe a que la sincronización de tiempo nunca es perfecta porque está afecta a una imprecisión

entre los relojes de cada nodo desplegado. Estos desfases en la ejecución de tareas coordinadas no están implementados en el simulador de LatinOS.

- **Diferencias de arquitectura:** Dado que la arquitectura de memoria de un nodo inalámbrico es distinta a la arquitectura donde se ejecuta la simulación (computador de escritorio), tampoco es posible detectar vía simulación fallas relacionadas con problemas de lectura o escritura en memoria.

El despliegue en terreno fue realizado encapsulando los nodos inalámbricos en tubos de PVC sellados con tapas recubiertas de goma, como lo muestra la Figura 4.8. Esto permite proteger a los nodos de polvo, lluvia, y otros agentes nocivos.

FIGURA 4.8. Encapsulado de los nodos Z1 para despligue en terreno. La utilización de tubos de PVC como caja protectora para el nodo inalámbrico permite resistencia del nodo a agentes externos que pudieran dañarlo físicamente, sin interferir significativamente la calidad de las comunicaciones inalámbricas.

La información recibida desde la RIS por el nodo acumulador escrita en una base de datos *mysql* mediante una rutina de un PC que lee la información entregada por el acumular vía un puerto USB. Esta información es desplegada en una página web como lo muestra la Figura 4.9. Además via una segunda página web, es posible desplegar gráficos como lo muestra la figura 4.10.

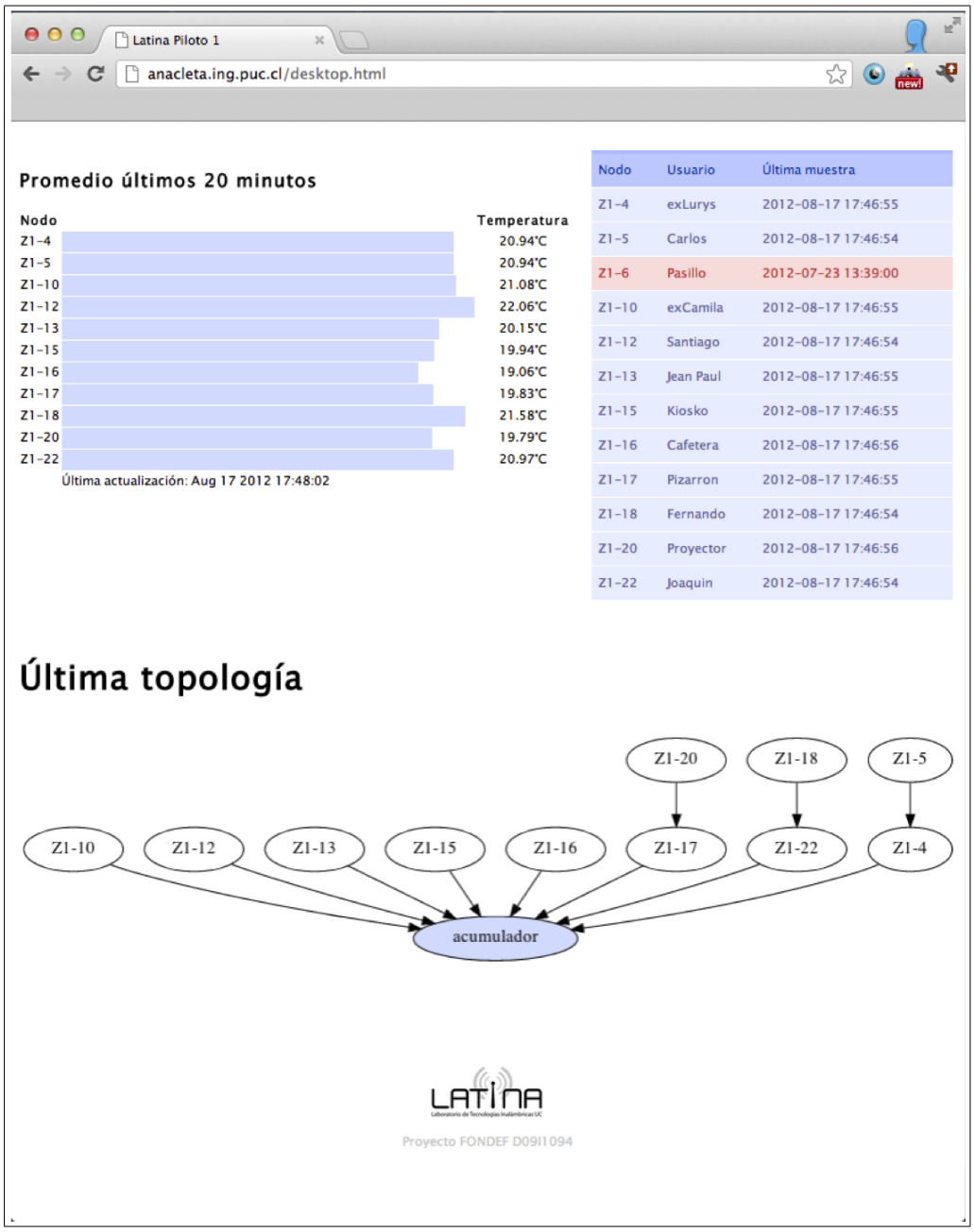

FIGURA 4.9. Red de monitoreo ambiental en funcionamiento. Despliegue de datos en una página web alojada en un servidor local que muestra la información entregada por la RIS.

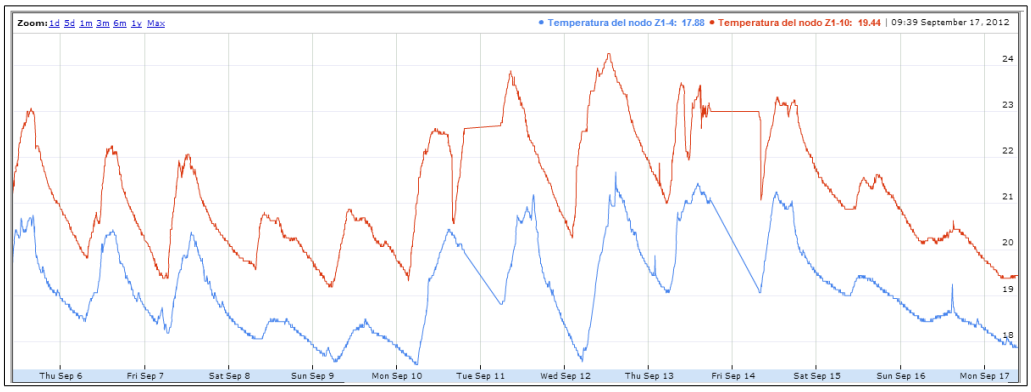

FIGURA 4.10. Datos de temperatura reportados por nodos de la RIS. Éstos pueden ser desplegados en nuestra página web en tiempo real mientras la red se encuentra activa.

El despliegue de esta red, su funcionamiento desde el mes de Enero de 2012, y la acumulación de más de 1 millón de datos, demuestra el correcto funcionamiento del protocolo desarrollado y en consecuencia valida también la metodología desarrollada para su implementación.

El set de herramientas desarrollado para la acumulación y despliegue de datos, hace posible la experimentación y monitoreo en terreno para aplicaciones comerciales o de investigación. En un futuro cercano se espera entregar conectividad celular y/o satelital al nodo acumulador con el fin de desplegar RIS que puedan reportar datos en áreas que se encuentran fuera de alcance de redes celulares o de internet.

5. PROTOCOLO DE BENCHMARKING

Este protocolo tiene como objetivo determinar la calidad de un enlace inalámbrico para distintas distancias de separación entre nodos. Un buen índice de corbertura está dado por la razón entre la cantidad de paquetes recibidos por el nodo receptor y el total de paquetes transmitidos por el nodo transmisor.

Para lograr medir la conectividad entre varios pares de nodos en forma simultánea, los nodos son ubicados a lo largo de una línea recta, de modo que mientras uno transmite paquetes, el resto los recibe a distintas distancias. Para asegurar la obtención de datos insesgados, este experimento cumple una rutina de ejecución que consta de las etapas mostradas en la Figura 5.1 y descritas a continuación:

1. Chequeo de sanidad de la red.
2. Elección de un nodo transmisor y difusión de paquetes: El nodo transmisor de turno realiza una difusión (*broadcast*) de “paquetes experimentales”. Los nodos restantes registran los “paquetes experimentales” recibidos en memoria *flash*.
3. Repetición del paso 2 en forma secuencial para todos los nodos de la cadena.
4. Recolección de todos los registros de “paquetes experimentales” recibidos a través de la misma RIS hacia el nodo acumulador.

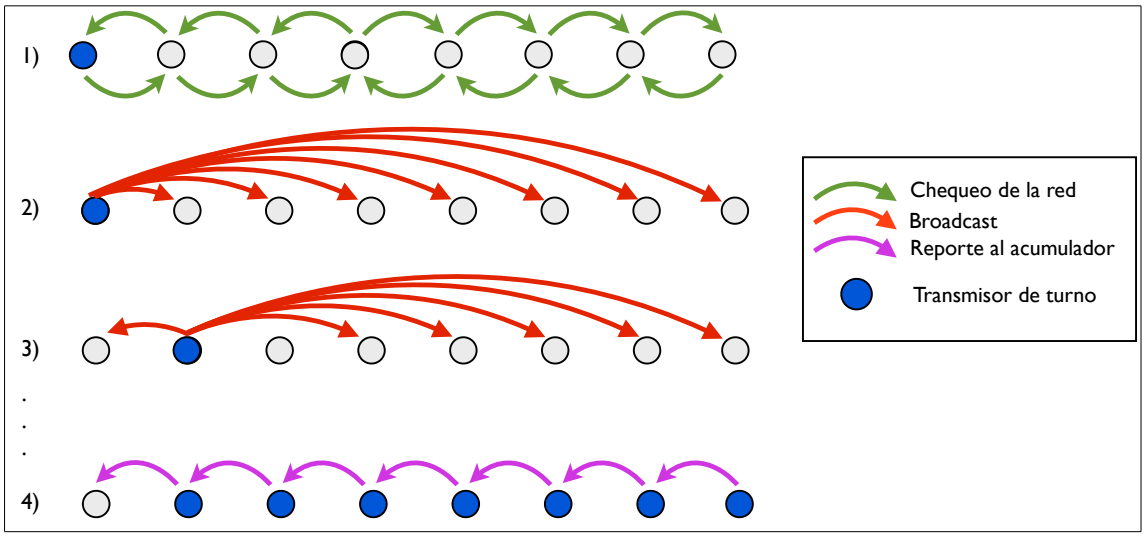

FIGURA 5.1. Etapas del protocolo de *Benchmarking*. 1) Chequeo de la Red. 2) Elección del nodo como transmisor (inicialmente el primero) y *broadcast* de paquetes. 3) Repetición del paso 2 para todos los los nodos de la cadena. 4) Reporte de datos al acumulador.

A continuación se describen los detalles de la metodogía de diseño aplicada a la implementación del protocolo de benchmarking.

5.1. Definición funcional del *stack* de protocolos

El módulo Aplicación toma la labor de dar inicio al experimento, establecer los parámetros de funcionamiento como por ejemplo: la cantidad de nodos utilizados, la cantidad de "paquetes experimentales" a difundir, el tiempo entre transmisión de este tipo de paquetes, etc. y reportar los datos al primer nodo de la cadena (designado como nodo acumulador).

Este *protocol stack* es más simple que el de Monitoreo Ambiental reportado en el capítulo 4, dado que no se requieren funcionalidades relativas al descubrimiento de vecinos, ahorro de energía, sincronización, etc.

Los servicios que debe prestar el protocolo son los siguientes:

- **Direccionamiento**: Las direcciones de red son asignadas en forma ascendiente partiendo desde 1 hasta N (N es la cantidad de nodos que intervienen en el experimento). Esta asignación viene definida por el usuario tal que el número 1 es el acumulador y el número N es el último nodo de la línea.
- **Chequeo de sanidad de la red**: El chequeo de sanidad de la red tiene por objetivo revisar que todos los nodos de la red estén energizados y con comunicación bidireccional y estable al menos con sus nodos inmediatamente vecinos. Por esto, el mensaje de chequeo recorre la cadena hasta el último nodo y vuelve al acumulador.
- **Confirmaciones de paquetes transmitidos**: Aquellos paquetes que controlan las distintas etapas del experimento deben llevar confirmación (ACK) para asegurar que cada nodo realice la etapa de difusión de "paquetes experimentales" en forma correcta y que la transferencia de los datos registrados hacia el acumulador sea exitosa.
- **Topología y enrutamiento fijos y prefedinidos**: Dado que los nodos tienen una topología fija (línea recta), es necesario implementar reglas de enrutamiento acordes a esta disposición de los nodos, asegurando que las transmisiones de paquetes se realicen solamente entre nodos inmediatamente vecinos a excepción de los "paquetes experimentales".
- **Almacenamiento local de datos**: Para hacer más robusto el experimento, los registros de "paquetes experimentales" deben ser guardados tanto en la memoria RAM como en la memoria *flash* del nodo, evitando posibles errores de lectura o escritura que afecten el resultado del experimento.
- **Enrutar datos al acumulador**: Los registros obtenidos por cada nodo deben ser transmitidos al acumulador en forma ordenada, evitando congestión y colisión de paquetes.

5.2. Diseño de paquetes

En este caso, se necesitan 5 tipos de paquetes diferentes:

- `NETCHECK`: Son paquetes transmitidos durante la etapa de chequeo de sanidad de los nodos y de la red.
- `BCAST`: Son los "paquetes experimentales" difundidos vía *broadcast* por cada nodo transmisor y registrados en los receptores para medir la conectividad.
- `NEXT`: Son aquellos paquetes que designan como transmisor a quien los reciba.
- `DATA`: Son los paquetes que contienen los resultados de conectividad de cada nodo.
- `END`: Paquetes que definen el término de reporte de datos de un nodo y gatillan el inicio del reporte de datos para el nodo siguiente.
- `ACK`: Paquetes de confirmación utilizados para comunicaciones de paquetes `NEXT`, `END` y `DATA`.

El funcionamiento del protocolo mediante todos estos tipos de paquete se muestra en la Figura 5.2. Es posible ver que las etapas 3 y 4 han sido ilustradas con mayor detalle para reportar de mejor forma la manera de operar del *protocol stack*.

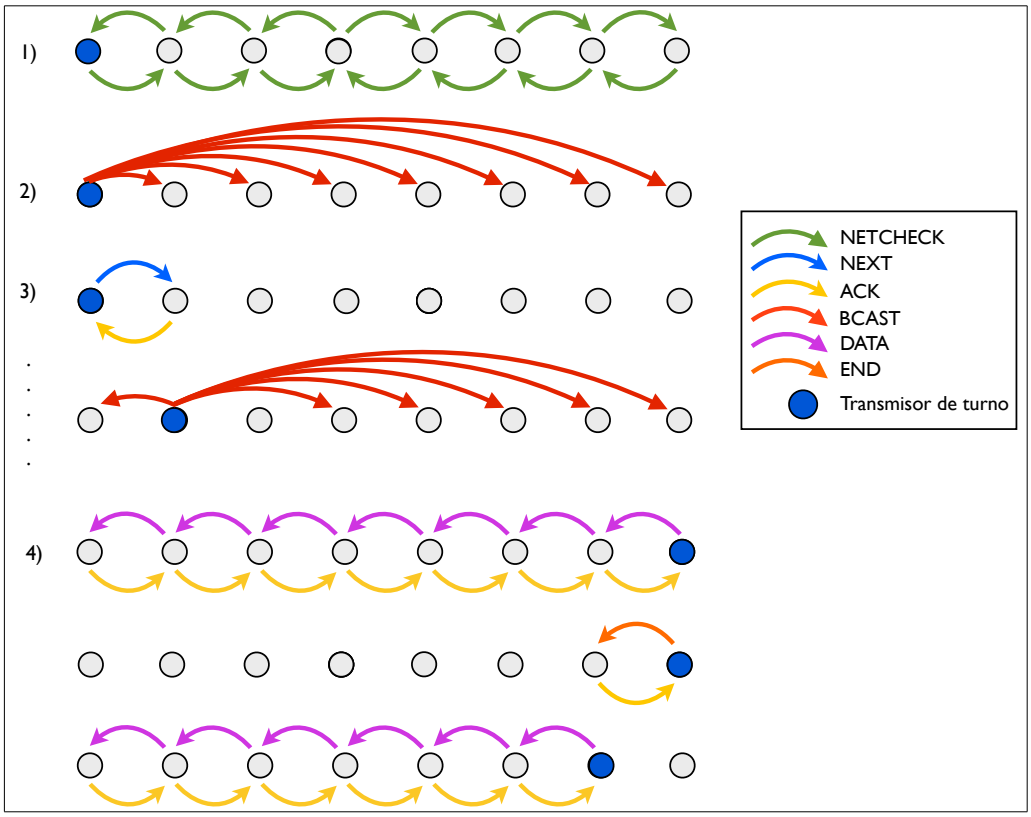

FIGURA 5.2. Etapas detalladas del protocolo de *Benchmarking*. 1) Chequeo de la Red. 2) *Broadcast* de paquetes. 3) Relevo del transmisor y repetición de la etapa 2. 4) Reporte de datos al acumulador.

Luego de la definición de los tipos de paquete, se escoge un *header* que contiene los 2 campos básicos (destinatario y tipo de paquete) y además un campo de "origen" que es útil para hacer el enrutamiento de los paquetes. La Figura 5.3 ilustra la estructura del *header* de paquetes.

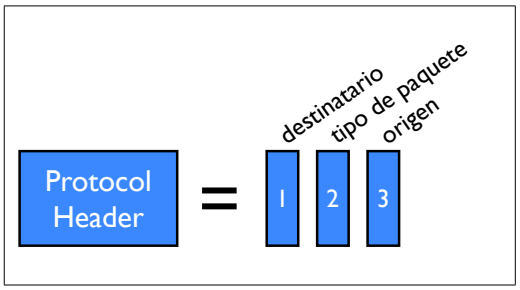

FIGURA 5.3. *Header* de paquetes del protocolo de *Benchmarking*. La cabecera de los paquetes para este protocolo, contienen 3 campos: destinatario, tipos de paquete y origen (qué nodo envió el paquete).

La composición de los paquetes de cada tipo, se detalla en la Figura 5.4. Los paquetes de tipo `BCAST` además del *header* contienen un número de secuencia que los diferencia entre ellos. Este número de secuencia permite hacer un análisis más detallado que la conectividad que será objeto de investigación en el futuro.

Los paquetes de datos (`DATA`), además de encapsular a un paquete de aplicación, llevan información específica del origen de los datos en el campo "*data source*", que se hace necesario para poder enviar un paquete de confirmación al nodo que los generó.

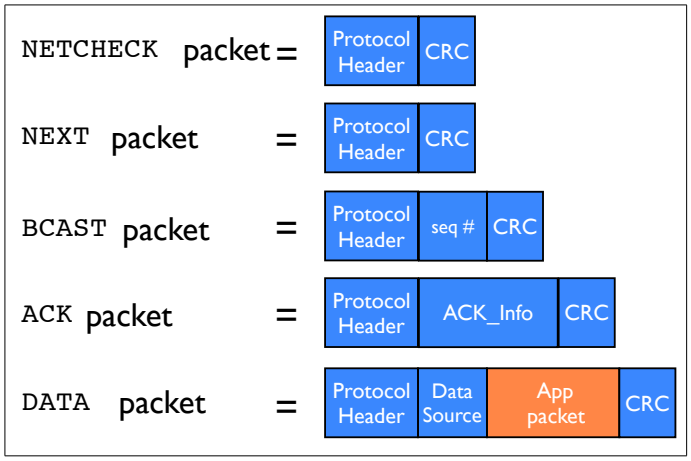

FIGURA 5.4. Diseño de paquetes para protocolo de *Benchmarking*. Los paquetes `NETCHECK`, `NEXT` y `DATA` no contienen información adicional al *header*. Los paquetes de tipo `BCAST` contienen un número de secuencia, y los paquetes de tipo `DATA` contienen un paquete de aplicación y un campo con la dirección del nodo que generó los datos

5.3. Diagrama de flujo del módulo Protocolo

En esta sección se describen el flujo de datos y las rutinas ejecutadas por el módulo Protocolo en la implementación del protocolo de *Benchmarking*.

La arquitectura general de flujo de paquetes en el módulo Protocolo es presentada en la Figura 5.5.

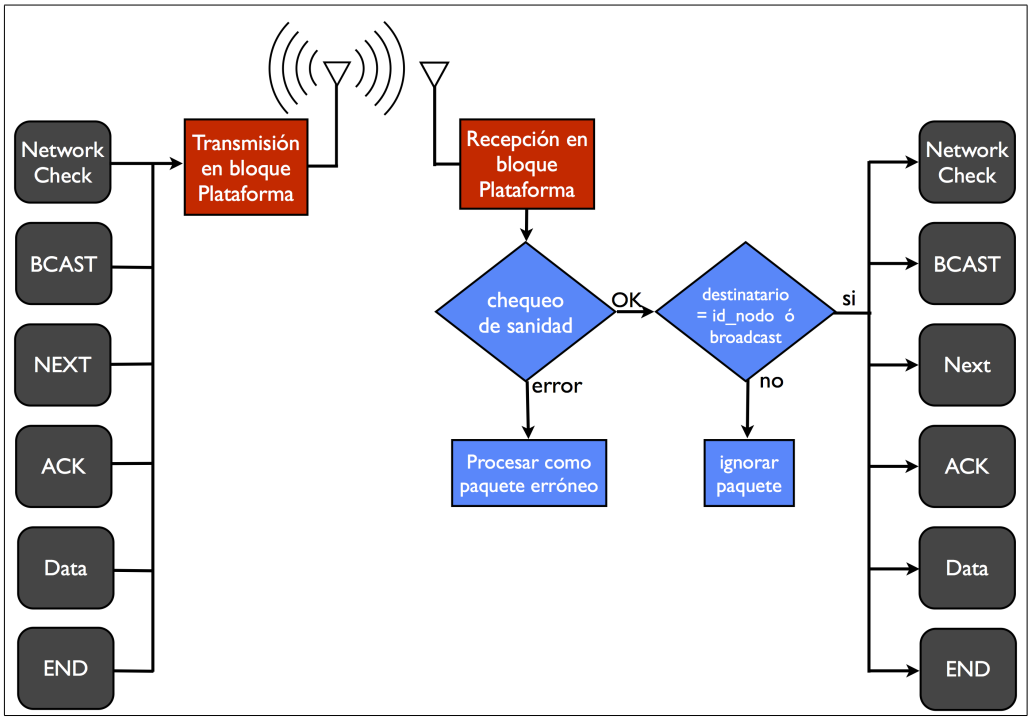

FIGURA 5.5. Diagrama de flujo del módulo Protocolo para *Benchmarking*. Los datos se generan y procesan en forma directa en el módulo Protocolo. En el nodo receptor se revisa la sanidad y el destinatario del paquete. Luego el módulo Protocolo ejecuta rutinas específicas para cada tipo de paquete recibido.

El flujo completo puede ser subdividido en dos de los tipos indicados por la metodología propuesta:

- La reacción del módulo Protocolo ante la recepción de paquetes desde el módulo Plataforma, es decir, de otros nodos, se muestra en el flujo presentado en la Figura 5.6.
- Los eventos internos del protocolo: Gatillan el cambio de etapa del experimento, ya sea para iniciar el chequeo de la red (`NETCHECK`), avisar al nodo siguiente que comience la etapa de *broadcast* (`NEXT`) o iniciar el reporte de datos hacia el acumulador (`DATA`) según lo muestra la Figura 5.7.

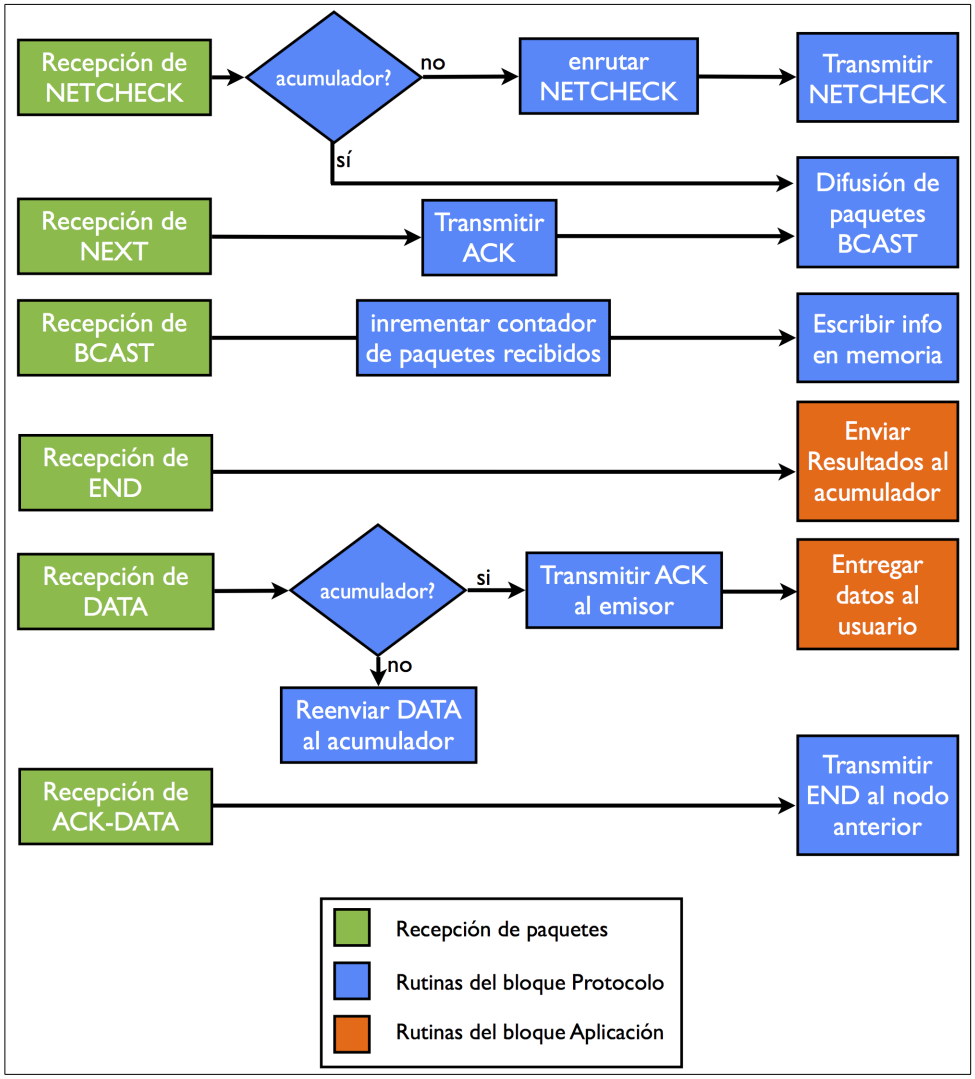

FIGURA 5.6. Recepción de paquetes. Para cada tipo de paquete recibido se describen las rutinas y procesos desencadenados por el módulo Protocolo.

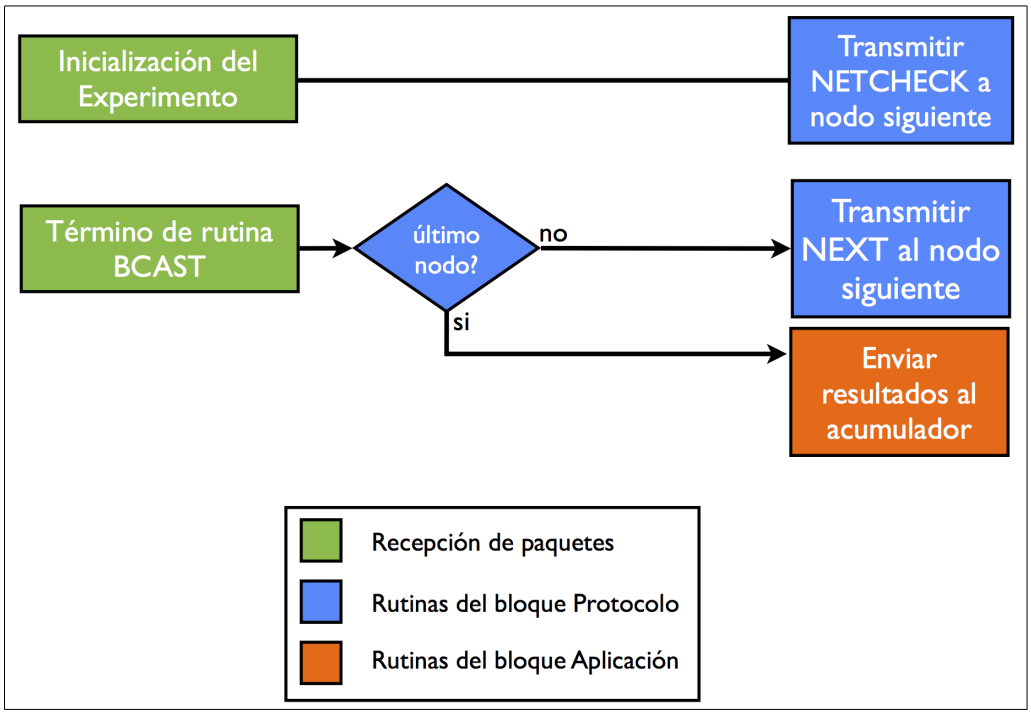

FIGURA 5.7. Eventos internos. Rutinas ejecutadas al iniciarse el experimento o al terminar una etapa de *broadcast*.

5.3.1. Enrutamiento de paquetes

El enrutamiento de paquetes sigue las siguientes reglas:

1. Los paquetes `NEXT` son transmitidos al nodo siguiente hasta llegar al último nodo.
2. Los paquetes `DATA` y `END` son enrutados al nodo anterior.
3. Los paquetes `NETCHECK` son enrutados al nodo siguiente hasta llegar al último nodo y luego enrutados al nodo anterior hasta llegar de vuelta al acumulador.

Para lograr implementar el enrutamiento anterior, se utiliza la dirección de origen del último paquete recibido para enrutar el paquete a transmitir.

A modo de ejemplo, si un paquete `NETCHECK` es transmitido desde el nodo 5 al 4, este paquete será reenviado al nodo 3. Si es enviado del 4 al 5, será reenviado al nodo 6 a menos que el nodo 5 sea el último de la cadena, en cuyo caso se reenviaría el paquete de vuelta al nodo 4.

Paquetes de confirmación (`ACK`)

Los paquetes de confirmación fueron implementados de la misma forma que el experimento de monitoreo ambiental (ver Sección 4.3.2) y usan solamente para confirmar los paquetes `DATA`, `NEXT` y `END`. El paquete `NETCHECK` al retornar al acumulador sirve como su propia confirmación por lo que no necesita de un paquete `ACK` específico. Si el paquete no vuelve dentro de un lapso de tiempo predefinido, la rutina de chequeo vuelve a comenzar.

5.3.2. Difusión de "paquetes experimentales" (`BCAST`)

Cuando a un nodo le toca el turno de comenzar la difusión de "paquetes experimentales" de tipo `BCAST`, debe hacerlo incrementando desde 0 el número de secuencia con cada paquete `BCAST` nuevo que es transmitido.

Además, el transmisor debe esperar un tiempo predefinido entre la transmisión de paquetes sucesivos. Esto es necesario por 2 razones:

1. Asegurar que cada paquete transmitido viaja por un canal inalámbrico independiente del anterior, por lo tanto, la separación entre paquetes debe ser de, al menos, el tiempo de coherencia del canal. El tiempo máximo de coherencia del canal para una banda de 2.4 GHz en baja movilidad (3 Km/h) es de 63 ms aproximadamente.
2. El tiempo entre paquetes debe permitir que los nodos receptores alcancen a procesar cada paquete recibido y registrarlo en la memoria, tarea que puede resultar lenta por limitaciones del *hardware*. El tiempo de procesamiento y escritura en memoria flash en el nodo Z1 con LatinOS bordea los 5 ms.

El tiempo entre paquetes es un parámetro del protocolo y puede ser modificado por el usuario, pero para este experimento en específico se utilizó una separación de paquetes de 100ms para asegurar que los paquetes transmitidos enfrentaran canales independientes.

5.3.3. Recepción y procesamiento de mensajes de *broadcast*

Al recibir un mensaje de tipo `BCAST`, el nodo debe registrar el número de secuencia del paquete en su memoria *flash* para procesamiento posterior. Además, lleva un contador de mensajes `BCAST` recibidos por cada uno de los nodos de la red. Esto quiere decir que en la memoria RAM existe un arreglo de N valores enteros (uno por cada nodo de la red) que incrementan su valor cada vez que llega un mensaje de tipo `BCAST` de alguno de los nodos de la red. Este arreglo es posteriormente solicitado por la aplicación para ser enviado al acumulador.

5.3.4. Reporte de datos

El reporte de datos se realiza cuando todos los nodos de la red han completado la transmisión de paquetes `BCAST`, es decir, cuando el último nodo de la cadena ha terminado de transmitir los "paquetes experimentales" de tipo `BCAST`. En ese momento, los nodos deben reportar al acumulador en orden descendente (del nodo más lejano al nodo más cercano) la cantidad de paquetes recibidos de cada transmisor durante el experimento.

El reporte es iniciado por el último nodo de la cadena. Cuando éste recibe el paquete `ACK` que confirma que los datos llegaron correctamente al acumulador, transmite un paquete `END` al nodo anterior (el penúltimo de la cadena), el que inicia el reporte de datos hacia el acumulador y repite el proceso hasta que el nodo acumulador recibe los datos de cada nodo de la cadena. Cabe mencionar que los paquetes `ACK` para los paquetes de datos son originados solo cuando llegan al acumulador y no a cada salto de la cadena.

Para un análisis más profundo de los datos, es posible descargar manualmente la información presente en la memoria *flash* de cada nodo. Esta información está compuesta por los números de secuencia de cada paquete recibido por el nodo y de separadores que permiten identificar el origen (nodo transmisor) de cada set de paquetes recibidos. Estos números de secuencia, en conjunto con los separadores, constituyen un identificador único de paquete, lo que permite estudiar con mayor detalle el comportamiento de cada enlace durante todo el experimento.

5.4. Implementación de diagramas de flujo

La implementación en código C de los diagramas de flujo descritos en la sección anterior son insertados en el módulo Protocolo de LatinOS acorde con las reglas de implementación enunciadas en los Anexos A.1 y A.2. Luego todos los módulos de LatinOS se compilan utilizando la herramienta *open-source MSPGCC* (*GCC toolchain for MSP430*, 2012) para la plataforma Z1 y utilizando *GCC* (*General C Compiler*, 2012) para el nodo virtual de simulación `testnode_sim`.

5.5. Simulación y despliegue en terreno

La realización de este experimento fue comparada con los resultados del simulador para validar el funcionamiento del simulador de LatinOS utilizando un protocolo que fue diseñado para operar en terreno. Uno de los objetivos fue calibrar los parámetros de simulación para lograr un ajuste entre los resultados del experimento y los resultados de simulación.

Para la etapa de despliegue, los nodos fueron dispuestos en dos ambientes del campus San Joaquín de la Universidad Católica:

1. Ambiente de baja movilidad (ver Figura 5.8). El experimento fue desarrollado en el sector de deportes a las 18:00 hrs, donde el flujo de personas es mínimo y los obstáculos en línea de vista son nulos. Cabe mencionar la presencia de algunos árboles cercanos a los nodos desplegados y la ausencia de edificios en la zona.
2. Ambiente de alta movilidad (ver Figura 5.9). El experimento fue desarrollado en el patio de estudiantes de Pedagogía, en horario de almuerzo. La movilidad estaba dada principalmente por el alto flujo de personas que circulan por el patio a esa hora (13:00 hrs), sumado a la alta densidad de árboles y edificios en el sector.

FIGURA 5.8. Disposición de nodos en ambiente de baja movidad. Posición geográfica de nodos para la realización del experimento de *Benchmarking* en espacio urbano (edificios y alta densidad de peatones). La separación es de 20m abarcando una distancia total de 180m.

FIGURA 5.9. Disposición de nodos en ambiente de alta movilidad. Posición geográfica de nodos para la realización del experimento de *Benchmarking* en espacio abierto sin presencia de peatones. La separación es de 20m abarcando una distancia total de 180m.

Los siguientes parámetros de funcionamiento de la red fueron utilizados para simulación y despliegue del protocolo de *Benchmarking*:

Bloque	**Parámetro**	**Valor**
Protocolo	Cantidad de nodos	10
	paquetes `BCAST` a transmitir por nodo	500
	Intervalo entre paquetes `BCAST`	100 ms
	`WAIT_FOR_ACK_TIME`	3 s
Sistema Operativo	*Buffer* de recepción (paquetes)	20
	Tamaño máximo de paquetes	120 bytes
	Cola máxima de tareas	30
	Cola máxima de timers	12
Capa Física	Separación entre nodos	20 m
	Ambiente (Experimento 1)	Abierto sin movilidad
	Ambiente (Experimento 2)	Urbano con movilidad
	Exponentes de pérdida (simulación)	$\alpha_1 = 2{,}3$ $\alpha_2 = 3{,}8$

Los datos medidos en los experimentos en terreno fueron obtenidos en primera instancia a través del puerto serial del nodo acumulador una vez que los datos fueron reportados a éste por el resto de los nodos de la red. Estos resultados son entregados en forma inmediata para detectar errores o anomalías en la ejecución del protocolo. Posteriormente, los datos presentes en la memoria *flash* de cada nodo son extraídos y procesados utilizando rutinas en MATLAB que entregan los resultados de conectividad. Los datos obtenidos mediante simulación son entregados a MATLAB en el mismo formato en que lo hacen los nodos desplegados en terreno.

Los resultados procesados se ilustran en la Figura 5.10 y contrastan los experimentos realizados en el simulador de LatinOS con los experimentos realizados en terreno. Los parámetros de simulación de la radio fueron ajustados acorde a las especificaciones descritas por el *datasheet* correspondiente al transceptor (*Texas Instruments CC2420 Tranceiver*, 2012).

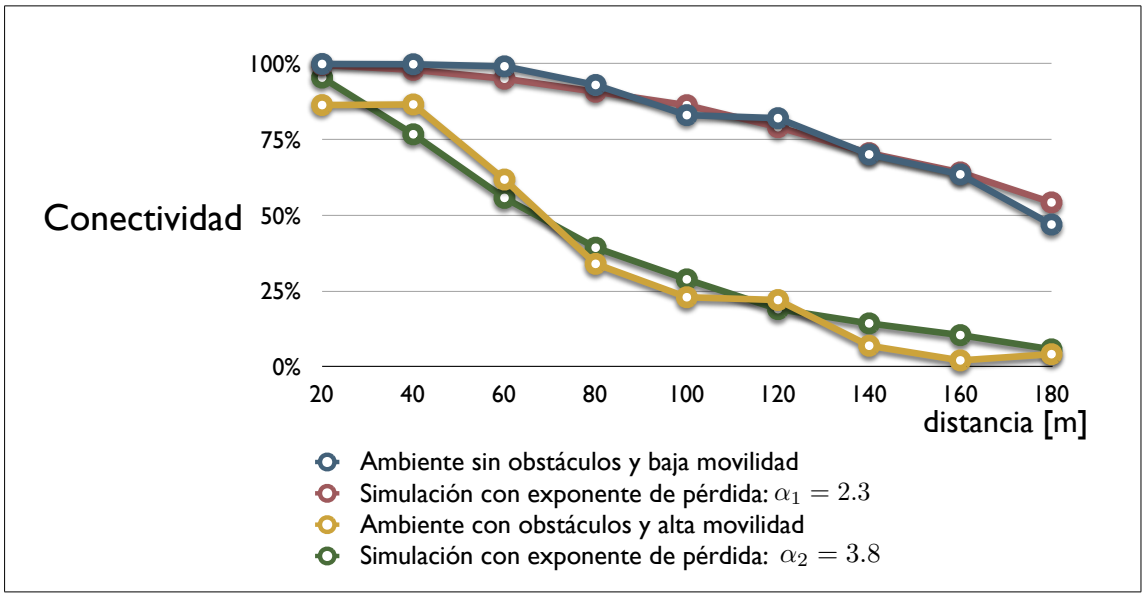

FIGURA 5.10. Resultados del experimento de Benchmarking. Curvas promedio de conectividad vs. distancia de separación entre nodos. Estos datos fueron entregados como resultado de experimentos de benchmarking realizados en ambientes con distintas caracterísiticas de movilidad o obstáculos. Además se muestran los resultado de la simulación para ambientes con exponentes de pérdida por distancia de $\alpha = 2{,}3$ y $\alpha = 3{,}8$

6. CONCLUSIONES Y TRABAJO FUTURO

La contribución principal de esta tesis es una metodología para el desarrollo de protocolos bajo LatinOS validada mediante dos ejemplos de aplicación concretos desplegados en terreno. Estos despliegues constituyen además un aporte al desarrollo de LatinOS como un sistema operativo y lo habilitan para ser utilizado en aplicaciones reales.

La versatilidad de la metodología propuesta se aprecia en que dos protocolos muy distintos son implementados siguiendo la misma secuencia de pasos sobre una misma plataforma de *software* (LatinOS) y de *hardware* (Z1). Esta versatilidad permitirá el desarrollo de una amplia variedad de protocolos y aplicaciones para redes de sensores utilizando LatinOS como sistema operativo, sin la necesidad de modificar los demás módulos del sistema.

Con la implementación del protocolo de monitoreo ambiental fue posible validar la metodología para la implementación en LatinOS de un protocolo escrito para otro sistema operativo (TinyOS) utilizando solamente la especificación funcional de *Sensorsope*.

En relación al experimento de *Benchmarking* es importante recalcar que:

1. Los resultados confirman el correcto funcionamiento del protocolo y por tanto validan la metodología.
2. La información que se obtuvo a partir del protocolo de *Benchmarking* permitió calibrar el simulador de modo que se ajustara a la realidad de cada ambiente de despliegue.
3. Esta calibración hace del simulador una herramienta empíricamente válida para el estudio y análisis de nuevos protocolos y nuevos despliegues en terreno.
4. El protocolo de *Benchmarking* constituye en sí misma una herramienta científica para hacer mediciones de conectividad en un ambiente arbitrario.

Esta tesis constituye un importante paso en el desarrollo del nuevo concepto de sistema operativo, LatinOS, que cuenta hoy con una metodología formal para el desarrollo de protocolos. Ésta permite anticipar y detectar posibles errores tanto en el *protocol stack*

como en el diseño de una RIS, con el fin de agilizar y facilitar su exitoso despliegue en terreno.

La primera línea de investigación futura surge a partir del experimento de *Benchmarking*. El tipo de resultados del experimento permite medir y analizar empíricamente la calidad de los enlaces generados entre nodos en ambientes determinados, lo que tiene uso potencial de modelamiento y simulación de RIS. Este *protocol stack* sin duda puede ser mejorado con nuevas técnicas de transferencia de datos, difusión de paquetes, y otras mejoras que escapan a los objetivos específicos de esta tesis.

Una área más extensa de investigación se vislumbra al concebir una red de sensores con capacidad de comunicaciones con múltiples antenas, es decir, utilizar un transceptor que cuente con la capacidad de direccionar las transmisiones. El implementar esta tecnología en RIS conllevaría un cambio en el paradigma de omnidireccionalidad de las transmisiones en el medio inalámbrico y es objeto del proyecto FONDEF: "Redes Inalámbricas de Sensores con Tecnologías de Múltiples Antenas". Esta tesis provee una herramienta metodológica para desarrollar nuevos protocolos que sean adecuados para ser utilizados con la tecnología de múltiples antenas.

Un tercer foco de investigación puede entreverse en las posibles arquitecturas que pueden tomar los protocolos en relación a su modularidad: La búsqueda de un punto medio entre la opción de implementar un modelo de capas cruzadas versus la implementación del modelo por capas separadas en forma estricta es sin duda un tema de estudio. En este trabajo se utilizó un modelo de capas aún más integrado que aquel propuesto por (Charfi et al., 2009), con desventajas en relación a la modularidad del protocolo, pero con resultados favorables en el tamaño y legibilidad del código además de una simplicidad en la estructura del mismo.

El constante desarrollo de protocolos llevará, sin duda, a nuevas exigencias para el sistema operativo, razón por la cual será necesario que LatinOS sufra también una constante evolución acorde a las nuevas tecnologías y requerimientos de los protocolos.

BIBLIOGRAFIA

Ansari, J., Zhang, X., Salikeen, O., y Mahonen, P. (2011, jan.). Enabling flexible medium access design for wireless sensor networks. En *Wireless on-demand network systems and services (wons), 2011 eighth international conference on* (p. 158 -163).

Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M., Couach, O., y Parlange, M. (2008, april). Sensorscope: Out-of-the-box environmental monitoring. En *Information processing in sensor networks, 2008. ipsn '08. international conference on* (p. 332 -343).

Buonadonna, P., Hill, J., y Culler, D. (2001). *Active message communication for tiny networked sensors.* Disponible en `http://webs.cs.berkeley.edu/tos/papers/ammote.pdf`

Campamá, S. (2012). *Sistema operativo para redes inalámbricas de sensores.* Disponible en `http://web.ing.puc.cl/latina/`

Charfi, W., Masmoudi, M., y Derbel, F. (2009, march). A layered model for wireless sensor networks. En *Systems, signals and devices, 2009. ssd '09. 6th international multi-conference on* (p. 1 -5).

Gcc toolchain for msp430. (2012). Disponible en `http://mspgcc.sourceforge.net/`

General c compiler. (2012). Disponible en `http://gcc.gnu.org`

Gruenen, J. V., y Rabaey, J. (2003). Lightweight time synchronization for sensor networks. En *Wsna.* Disponible en `http://php/pubs/pubs.php/309.html`

He, Z. (2007, dec). *Implementation and evaluation of the sensornet protocol for contiki.* Disponible en http://soda.swedish-ict.se/2715/1/SICS-T--2007-14--SE.pdf

Huang, P., Xiao, L., Soltani, S., Mutka, M., y Xi, N. (2012). The evolution of mac protocols in wireless sensor networks: A survey. *Communications Surveys Tutorials, IEEE*, *PP*(99), 1 -20.

Kusy, B., Dutta, P., Levis, P., Maroti, M., Ledeczi, A., y Culler, D. (2006, julio). Elapsed time on arrival; a simple and versatile primitive for canonical time synchronisation services. *Int. J. Ad Hoc Ubiquitous Comput.*, *1*(4), 239–251. Disponible en http://dx.doi.org/10.1504/IJAHUC.2006.010505

Memsic wireless modules. (2010). One Tech Drive St 325, Andover, MA 01810. Disponible en http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html

Sivrikaya, F., y Yener, B. (2004, july-aug.). Time synchronization in sensor networks: a survey. *Network, IEEE*, *18*(4), 45 - 50.

Texas instruments cc2420 tranceiver. (2012). Disponible en http://www.ti.com/product/cc2420

Thouvenin, R. (2007). *Implementing and evaluating the dymo protocol in wsn.* http://docs.tinyos.net/tinywiki/index.php/Tymo. Disponible en http://tymo.sourceforge.net/PDF/thesis.pdf

Zolertia wireless node. (2011). Disponible en http://zolertia.com/products/z1

ANEXO A. RECURSOS ADICIONALES

A.1. Sobre la implementación práctica de protocolos

Este anexo puede ser considerado como un manual técnico para la implementación de protocolos en LatinOS, considerando aspectos relevantes de estructuras de archivos, funciones a implementar, descripción de interfaces provistas por LatinOS, y algunas reglas para mantener el orden y legibilidad del código.

A.1.1. Estructura de archivos

Los protocolos de LatinOS tienen una estructura de archivos como la presentada en la Figura A.1 en la que se alojan los distintos protocolos. Cada protocolo debe estar ubicado dentro de una subcarpeta que lleva el nombre del protocolo, dentro de la carpeta *protocols* y debe contener al menos tres archivos:

Archivo	**Descripción**
`protocol_name.c`	Archivo principal, que implementa las funciones del protocolo.
`protocol_name.h`	Archivo de encabezado, conocido tambien como *header file*. Contiene las declaraciones de las funciones implementadas
`Rakefile`	Archivo que sirve para la compilación del *firmware* y es genérico para todos los protocolos.

TABLA A.1. Archivos mínimos para implementar un nuevo protocolo sobre LatinOS

Si el usuario desea agregar más archivos para privilegiar la modularidad de la implementación, es posible hacerlo.

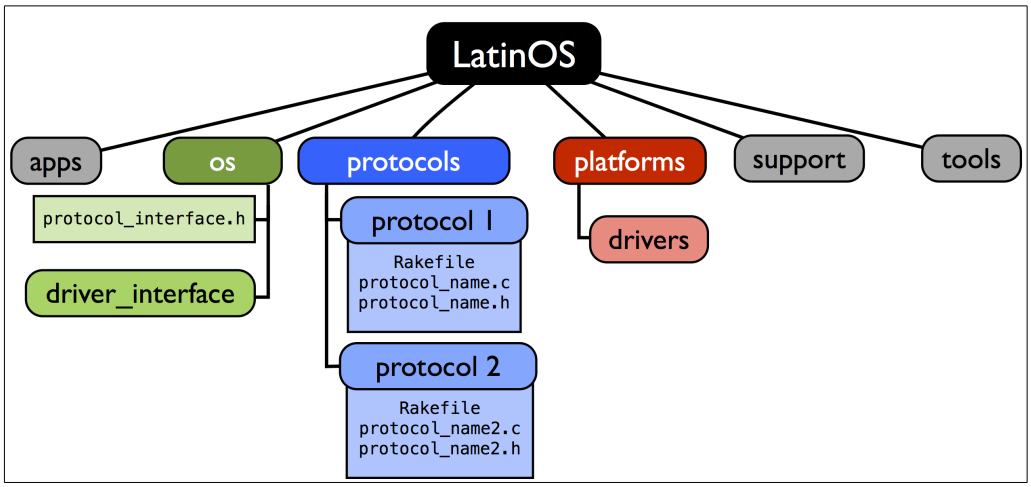

FIGURA A.1. Estructura de archivos. El protocolo debe ser implementado en un subdirectorio que lleve el nombre del protocolo, dentro del directorio *protocols*, y contenga al menos tres archivos: Una cabecera, un archivo principal (ambos en lenguaje C), y un archivo `Rakefile` escrito en lenguaje Ruby.

A.1.2. Funciones mínimas de un bloque Protocolo

Una vez ubicados en la estructura, y creados los archivos del nuevo bloque Protocolo, debemos escribir el *header file*, con las declaraciones de funciones que serán utilizadas en el bloque Protocolo, y las constantes a utilizar. Es recomendable utilizar como ejemplo el *header file* correspondiente al protocolo *default* ubicado en la carpeta *protocols* de LatinOS.

Para que el *protocol stack* integrado al módulo Protocolo de LatinOS sea funcional, debe implementar como mínimo cuatro funciones: Inicialización general, inicialización del(los) nodo(s) acumulador(es), transmisión de datos y recepción de datos. Estas se listan en la tabla A.2:

Función	Descripción
`protocol_boot()`	Ejecuta instrucciones necesarias para inicializar el protocolo
`protocol_sink_boot()`	Ejecuta instrucciones necesarias para inicializar el acumulador
`protocol_receive()`	Recibe y procesa paquetes provenientes de otro nodo y recibidos a través del bloque Plataforma
`protocol_transmit()`	Procesa paquetes a transmitir provenientes desde el bloque Aplicación

TABLA A.2. Funciones mínimas que debe implementar un protocolo en LatinOS.

Para poder transmitir paquetes al aire el protocolo debe utilizar el transceptor invocando la función `modem_transmit`, predefinida por LatinOS. De la misma forma, para que el protocolo pueda entregarle paquetes a la aplicación debe invocar a la función `app_receive`.

Las funciones mínimas `protocol_transmit` y `protocol_receive` forman parte de las interfaces de intercambio de paquetes entre bloques. En la figura A.2 se detalla la interacción de bloques mediante interfaces.

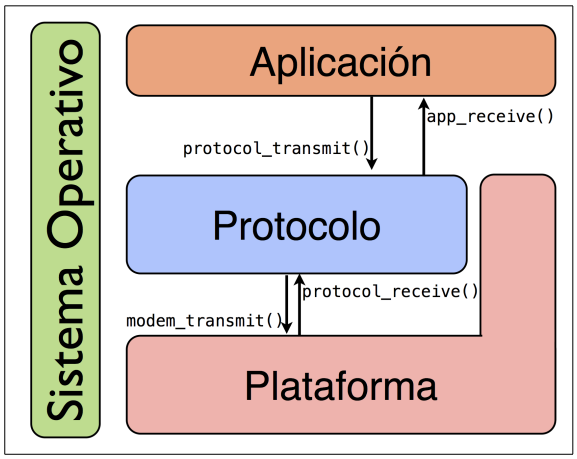

FIGURA A.2. Interfaces de intercambio de datos. Para poder transferir información, cada uno de los bloques debe invocar a las funciones de interfaz presentes en los otros bloques de LatinOS.

A.1.3. Servicios disponibles del bloque Sistema Operativo

Para facilitar la implementación de protocolos, LatinOS provee un set de cinco módulos que implementan funcionalidades primarias de la plataforma. Estos módulos son: Reloj, Tareas, *Timers*, ID, y Módem. Las funciones de estos módulos pueden ser invocadas desde cualquier bloque y se encuentran listadas en la tabla A.3. Información específica del funcionamiento interno de estos módulos se encuentra en el trabajo de (Campamá, 2012).

Módulo	Función	Return	Descripción
Reloj	`clock_time()`	**`uint64_t`**	Devuelve la hora del nodo en resolución de nanosegundos.
	`clock_set_time(uint64_t)`	**`void`**	Entrega un nuevo valor a la hora del nodo en nanosegundos.
Tareas	`task_add(&handler, PRIORITY, arg )`	**`void`**	Agrega una tarea que ejecuta la función `&handler` con `arg` como argumento, con prioridad dada por: `PRIORITY_HIGH`, `PRIORITY_MEDIUM` o `PRIORITY_LOW`.
Timer	`one_shot_timer_add(delay,&handler)`	**`int16_t`**	Agrega una tarea a la cola con prioridad alta, para ser ejecutada en un tiempo `delay` dado en milisegundos y devuelve el ID del *timer*.
	`one_shot_timer_cancel(timer_id)`	**`void`**	Cancela la ejecución del *timer* indicado por `timer_id`.
Modem	`modem_turn_off`	**`void`**	Apaga el módem.
	`modem_turn_on()`	**`void`**	Enciende el módem.
	`modem_receive_start()`	**`void`**	Inicia la recepción de paquetes.
	`modem_set_tx_power(level)`	**`void`**	Controla la potencia de las transmisiones. El nivel va desde 1 (potencia mínima) hasta 4 (potencia máxima).
	`modem_is_on()`	**`uint8_t`**	Entrega el estado del módem: 1 (Encendido) ó 0 (Apagado).
ID	`ident_get()`	**`uint32_t`**	Devuelve el ID único de cada nodo.
	`ident_set(id)`	**`void`**	Escribe un ID único por nodo en la memoria no volátil de la plataforma.

TABLA A.3. Descripción de uso de interfaces de los módulos provistos por LatinOS para el desarrollo de protocolos

A.2. Implementación de código en el entorno de LatinOS

A.2.1. Implementación de tipos de paquete

Los paquetes de control son armados dentro de una función que lleva un nombre definido por el tipo de paquete. Esta función agrega el encabezado básico y el contenido del paquete. La adición de la cola queda a criterio del usuario. Esto se debe a que como todos los paquetes deben agregar un CRC, no tiene sentido escribir este pedazo de código distinto para cada tipo de paquete y hacerlo en forma genérica (para todos los tipos) antes de enviarlo.

En la implementación de código para sistemas embebidos, es muy recomendado reducir al mínimo el uso de memoria dinámica. Para evitar el uso de VLAs (arreglos de largo variable), que pueden producir fallas de *overflow* de memoria, es necesario definir los arreglos de tamaño fijo. Si el largo del tipo de paquete es fijo, debe guardarse un espacio de memoria suficiente para almacenar tanto el paquete mismo como su cola (CRC), y si el largo del paquete es dependiente de la aplicación o de otro, debe asignársele un espacio de memoria del tamaño máximo de un paquete (que debe ser definido en el encabezado del protocolo como `PACKET_MAX_SIZE)`.

Para el armado de paquetes se recomienda la utilización de arreglos de memoria, o bien la utilización de estructuras (*struct*), que deben ser definidos en la cabecera del protocolo.

A.2.2. Implementación de detección de errores

Es muy recomendado que los datos lleven consigo algún tipo de chequeo de redundancia cíclica, conocida como CRC, para asegurar la sanidad de paquetes. Este chequeo implica adjuntar algunos bytes al final de cada paquete enviado en un campo llamado *footer*. Existen protocolos con tres tipos de exigencias respecto de este tema:

1. Protocolos que solo buscan paquetes correctos (y los erróneos deben ser descartados).
2. Protocolos que procesan por igual paquetes correctos y erróneos.

3. Protocolos que necesitan procesar de manera distinta paquetes erróneos y correctos.

A la luz de estos tres enfoques, es más correcto que sea el bloque Protocolo y no el bloque Plataforma el que implemente esta funcionalidad. Si el chequeo se realizara en capas inferiores, el protocolo no sería capaz de satisfacer necesidades para el tercer enfoque en forma limpia y ordenada.

Dado que todos los paquetes son enviados desde el protocolo al sistema operativo con un CRC incluido, se propone la implementación basada solamente en dos funciones. Una para calcular el CRC (`crc_calc()`) y otra para saber si el paquete está correcto o no (`crc_check()`).

La adición del CRC implica que el paquete debe estar armado en forma completa antes de realizar el cálculo y ante cualquier modificación, este debe ser recalculado.

A.2.3. Consideraciones para la escritura de código en LatinOS

Cabe mencionar además que, a lo largo de todo el proceso de implementación, el usuario debe mantener ciertos estándares en cuanto a la escritura de código:

- Todas las funciones escritas en el código deben comenzar con el nombre del archivo donde se encuentran, o con el nombre del bloque de LatinOS al que pertenecen (`protocol`, `app`, `os`). Esto sirve para facilitar la búsqueda de funciones en el sistema de archivos de LatinOS.
- Constantes deben ser definidas en el archivo de encabezado (*header file*) con letras mayúsculas.
- Nuevas estructuras (*structs*) deben ser declaradas con el nombre `structname_t`.
- Las variables utilizadas durante toda la ejecución del programa (y por ende necesitan de un espacio de memoria fijo), deben ser declaradas al principio del archivo principal con el comando: **`static`** `<var_type> <variable_name> = <initial_value>;`
- En lo posible evitar la asignación de memoria forma dinámica.

A.3. Compilación del código para despliegue y simulación

Para compilar el código escrito, es necesario invocar el comando `rake`, seguido de la plataforma para la cual se desea compilar tal como lo muestra la Tabla A.4.

Función	**Comando**
Compilar para despliegue	`rake target=platform_name`
Compilar para simulación	`rake target=testnode_sim`
Instalar *firmware* en nodo Z1	`rake target=z1 install`

TABLA A.4. Comandos para compilar, descargar y simular el código escrito. Para nuestro caso, reemplazamos `platform_name` por `z1`.