TESIS DEFENDIDA POR

Issac Noé García Garza

Y aprobada por el siguiente comité:

Dr. José Antonio García Macías
Director del Comité

Dra. Ana Isabel Martínez García
Miembro del Comité

Dr. Jaime Sánchez García
Miembro del Comité

Dr. Pedro Gilberto López Mariscal
Coordinador del Posgrado en Ciencias en Ciencias de la Computación

Dr. David Hilario Covarrubias Rosales
Encargado del Despacho de la Dirección de Estidios de Posgrado

10 de Diciembre del 2007.

CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

PROGRAMA DE POSGRADO EN CIENCIAS
EN CIENCIAS DE LA COMPUTACIÓN

Reprogramación Incremental en Redes Inalámbricas de Sensores

TESIS

que para cubrir parcialmente los requisitos necesarios para obtener el grado de

MAESTRO EN CIENCIAS

Presenta:

Issac Noé García Garza

Ensenada, Baja California, México. Diciembre de 2007.

RESUMEN de la tesis de **Issac Noé García Garza**, presentada como requisito parcial para obtener el grado de MAESTRO EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN. Ensenada, B. C. Diciembre del 2007.

Reprogramación Incremental en Redes Inalámbricas de Sensores

Resumen aprobado por:

Dr. José Antonio García Macías

Director de Tesis

Las redes de sensores son un área de investigación reciente. Surgen gracias a la miniaturización de dispositivos electrónicos y al estudio del cómputo móvil y ubicuo.
Una red de sensores puede estar conformada por nodos de diferente fabricante, siempre y cuando sus componentes de radiocomunicación permitan comunicarse entre sí. Además de que el sistema operativo con el que trabajan permita comunicarse.
En ocasiones es necesario realizar actividades administrativas en una red inalámbrica de sensores, entre ellas se encuentra cambiar el programa en los nodos. Esta tarea administrativa puede ser debido a varias razones: actualizar el programa a una nueva versión, corregir errores o cambiar un programa por otro.
Para reprogramar los nodos de una red inalámbrica de sensores se puede hacer de dos maneras: la primera es conectar físicamente cada nodo a reprogramar a una computadora que le transmitirá el nuevo programa, la segunda consiste en enviar el nuevo programa utilizando el componente de radiocomunicación.
En este trabajo se toma como base un mecanismo de reprogramación a través del aire llamado Deluge. Se modificó para proveer programación incremental, donde se hace una comparación a nivel de bytes del programa anterior con el nuevo programa y sólo se diseminan las diferencias. Además se incorporan conceptos de enrutamiento multicast para reprogramar sólo un conjunto de nodos basado en ciertos atributos.
Con base en las pruebas realizadas se pudo comprobar que reprogramando de forma incremental se disminuye el consumo de energía al reprogramar, en comparación con diseminar el programa entero.

Palabras clave: Redes Inalámbricas de sensores, Reprogramación Incremental, Protocolo de Enrutamiento Multicast.

ABSTRACT of the thesis presented by **Issac Noé García Garza**, as a partial requirement to obtain the MASTER SCIENCE degree in COMPUTER SCIENCES. Ensenada, B. C. December 2007.

Abstract approved by:

Dr. José Antonio García Macías

Thesis director

Wireless sensor networks is a recent research area . They emerge due to electronic devices miniaturization and research in ubiquitous and mobile computing.
A wireless sensor network can be formed by nodes from different manufacturers, as long as their radio component enables communication with each other. In addition the operating system must allow them to work together.
Sometimes it is necessary to perform administrative tasks in a wireless sensor network, these include changing the program in the nodes. This task can be due to several reasons: upgrade to a newer version, fix bugs or change completely the application logic.
Reprogramming nodes in a wireless sensor network can be done in two ways, the first one is having the node physically connected to a computer that has the new program, the second one is to send the new program through the radio component.
This work is based on a mechanism of reprogramming through the air called Deluge. It was modified to provide incremental reprogramming, which is based in a byte-level comparison of the previous program with the new program, disseminating only the differences. In addition, it incorporates concepts of multicast routing to reprogram only a set of nodes based on certain attributes.
Based on the tests, we noted that incremental reprogramming reduces energy consumption, compared to the dissemination of the entire program.

Keywords: Wireless Sensor Networks, Incremental Reprogramming, Multicast Routing Protocol.

Dedicatoria

Este trabajo se lo dedico a mis padres, por su apoyo y comprensión.

Ensenada, México
10 de Diciembre del 2007.

Issac Noé García Garza

// Agradecimientos

Indiscutiblemente tengo a muchas personas por agradecer en este trabajo, algunos por sus aportaciones técnicas, otros por lograr un rato ameno en momentos difíciles y de desvele. Otros simplemente por estar a mi lado, apoyarme moral y espiritualmente.
No podría catalogarlos por su importancia pues todos y cada uno de ellos de alguna manera fueron y son importantes en el desarrollo de esto trabajo.
Le agradezco al Dr Antonio por su apoyo, confianza y por su paciencia. De igual manera a los miembros del comité por sus consejos, su ayuda y jaladas de oreja: Dra. Ana Isabel y al Dr. Jaime Sánchez.
A los miembros del club de toby por sus sabios consejos y aportaciones al P. de la B. LeBorregue, tocayo, deivid, lynx y pequeña lulú. También a mis compañeros de cubo, a esos que son firmes en sus convicciones, aquellos por sus consejos en esos momentos de hambre, sin olvidar a los que van a todos los lugares, menos ese. Le agradezco además a la escuela de espartanos por su apoyo moral.
Quiero agradecer a lynx por no haberme acompañado a comer aquel día, por su amistad y sus consejos en la realización de este trabajo.
Agradezco también a los miembros de la ganga y al chino, por todos los años que han estado conmigo. Sin olvidar a mis amigos de ingeniería.
En especial agradezco a Elizabeth por haber aparecido en mi camino, por su apoyo, comprensión, sus risas y su compañía.
Agradezco a mis padres por todo lo que me han dado, mi educación, sus enseñanzas, su cariño y su compañía.
Agradezco de igual manera a mis familiares por confiar siempre en mi, y su apoyo que me han brindado.
Al Centro de Investigación Científica y de Educación Superior de Ensenada.
Al Consejo Nacional de Ciencia y Tecnología, por otorgar la beca que hizo posible la realización de este trabajo.
Agradezco además a todos aquellos que piensan que merecen una mención en esta hoja y no aparecen.

Ensenada, México
10 de Diciembre del 2007.

Issac Noé García Garza

Tabla de Contenido

Capítulo Página

Resumen **ii**
Abstract **iii**
Agradecimientos **v**
Tabla de Contenido **vi**
Lista de Figuras **viii**
Lista de Tablas **x**
I Introducción **1**
I.1 Antecedentes 1
I.2 Planteamiento del Problema 2
I.3 Objetivos 3
I.4 Contenido de la tesis 4
II Reprogramación de Redes Inalámbricas de Sensores **6**
II.1 Redes Inalámbricas de Sensores 6
II.1.1 Aplicaciones 7
II.1.2 Plataforma 8
II.2 Reprogramación de Redes Inalámbricas de Sensores 14
II.2.1 Mecanismos de Reprogramación 16
II.2.2 Programación Incremental 20
II.2.3 Diseminación de nuevos programas 26
II.3 Resumen 27
III Diseminación de Código **28**
III.1 Introducción 28
III.2 Diseminación de actualizaciones 28
III.3 Diseminación de Actualizaciones en Deluge 30
III.3.1 Representación de la información en Deluge 31
III.3.2 El protocolo 33
III.4 Enrutamiento sobre demanda 34
III.4.1 ADMR: Enrutamiento Multicast Adaptativo Sobre Demanda 36
III.5 Diseminación de actualizaciones sobre demanda 41
III.5.1 Protocolo de Formación de Rutas 41
III.5.2 Retransmisión del nuevo programa 43
III.6 Resumen 43
IV Implementación **45**
IV.1 Introducción 45
IV.2 Implementación 45
IV.3 Elección de grupo a reprogramar 46
IV.3.1 Representación de la información 50

Tabla de Contenido (Continuación)

Capítulo Página

IV.4 Formación de Rutas . . . 52
IV.4.1 Elección de ruta . . . 55
IV.4.2 Representación de la información . . . 58
IV.5 Preparación de paquetes a enviar . . . 59
IV.6 Arquitectura . . . 64
IV.7 Detalles técnicos de implementación . . . 66
IV.8 Resumen . . . 67

V Pruebas y resultados 68

V.1 Introducción . . . 68
V.2 Programación Incremental . . . 69
V.2.1 Cambio de constante . . . 69
V.2.2 Cambio en la implementación . . . 71
V.2.3 Cambio de un programa a otro . . . 76
V.2.4 Cambio en la configuración . . . 77
V.3 Resultados de la Programación Incremental . . . 80
V.3.1 Energía al reprogramar . . . 84
V.4 Formación de Rutas . . . 89
V.5 Pruebas de Reprogramación . . . 91
V.5.1 Reprogramando un nodo a varios saltos . . . 92
V.5.2 Reprogramando varios nodos . . . 95
V.6 Resumen . . . 97

VI Conclusiones, aportaciones y trabajo a futuro 98

VI.1 Conclusiones . . . 98
VI.2 Aportaciones . . . 99
VI.3 Trabajo a futuro . . . 100
VI.4 Limitaciones . . . 101

Bibliografía 102

A Algoritmos utilizados 105

A.1 Algoritmo CRC . . . 105
A.2 Algoritmo Rsync . . . 106
A.2.1 Algoritmo . . . 106
A.2.2 Rolling Checksum . . . 107
A.3 Algoritmo MD4 . . . 109

Lista de Figuras

Figura		Página
1	Nodo ESB/2	8
2	nodo Tmote Sky	9
3	Node BTnode	10
4	Estados de Deluge	17
5	Implementación de rsync (Jeong y Culler, 2005)	22
6	Estados de Deluge	30
7	Perfil de una imágen	31
8	Diseminación de páginas	32
9	Representación de la información de un programa	32
10	Ejemplo de árbol multicast formado por ADMR	38
11	Mensajes para formar ruta	42
12	Comportamiento de la retransmisión de un programa	44
13	Estructura del mensaje para formar rutas	47
14	Enlazado de componentes de DelugeMetadata	50
15	Estructura de DelugeImgDesc	51
16	Diseminación de mensajes por flood	52
17	Estructura de la tabla Membresía	53
18	Estructura de la tabla Nodo	53
19	Formación de Ruta	55
20	Métrica MAX_LQI	57
21	(A) Min_Hop (B) Max_Lqi	57
22	Representación de una imagen	60
23	Clase Paquete	62
24	Estructura del mensaje de datos	63
25	Comandos dentro de un mensaje	63
26	Enlazado de componentes	65
27	Pruebas de caso 1 y caso 2	81
28	Pruebas de caso 3 y caso 4	82
29	Estimación de gasto de energía	86
30	Tamaño de datos Incremental vs. No Incremental	87
31	Estimación de gasto de energía de forma Incremental	88
32	Estimación de gasto de energía de forma No Incremental	89
33	Programando dos nodos	90
34	Programando tres nodos	91
35	Comunicación de nodos	92
36	Programando un nodo	93
37	Programando un nodo a varios saltos	94

Lista de Figuras (Continuación)

Figura | Página

38 Programando cinco nodos 95
39 Programando varios nodos a cinco saltos 96

Lista de Tablas

Tabla Página

I Características principales de algunos nodos sensores 11
II Comparación de diferentes arquitecturas de reprogramación 16
III Número de paquetes Incremental y No Incremental 83
IV Energía utilizada en TmoteSky 84
V Parámetros para calcular energía 85
VI Ahorro de energía 87
VII Gasto de energía en nodos retransmirores 88
VIII Tiempo en reprogramar variando los saltos 94
IX Tiempo en reprogramar varios nodos a cinco saltos 96

Capítulo I

Introducción

Una red inalámbrica de sensores, es una red formada por pequeños dispositivos dotados de un componente de comunicación inalámbrica, algunos sensores, procesador para realizar algunas operaciones, y una memoria para almacenar algunos datos obtenidos de los sensores o de la aplicación corriendo en el nodo sensor. Gracias a la miniaturización y el bajo costo de los dispositivos electrónicos, es posible que distintos fabricantes de hardware puedan desarrollar dispositivos con las capacidades antes mencionadas.

I.1 Antecedentes

Como muchas áreas de investigación, el área de las redes inalámbricas de sensores empezó su actividad en la milicia norteamericana, sin embargo recientemente la industria y la comunidad científica de muchas partes del mundo se han dado cuenta de su potencial y esto ha dado un gran impulso dentro del cómputo móvil, protocolos de red y distintas áreas alrededor de las redes de sensores, incluyendo la electrónica.

Con el objetivo de proporcionar apoyo a la administración de una red de sensores, y de las aplicaciones con que cuentan los nodos se han propuesto diversos mecanismos o herramientas, entre las cuales se encuentra la actualización de los programas corriendo en los nodos. Es este es el tema principal de este trabajo, reprogramar a través del aire los nodos de una red de sensores y cuidar el consumo de energía al hacerlo.

I.2 Planteamiento del Problema

Actualmente los nodos sensores son relativamente de bajo costo, y existen en el mercado diferentes compañías que ofrecen este tipo de dispositivos, cada uno con algunas características propias, como puede ser diferentes tipos de sensores, o incluso con diferente procesador. Muchos de estos dispositivos, aunque sean de diferente fabricante e incluso con diferente arquitectura, son capaces de comunicarse y trabajar en conjunto.

La comunidad científica ha desarrollado algunas herramientas para reprogramar los nodos usando la unidad de radiocomunicación para transferir los nuevos programas. Sin embargo se ven limitadas a trabajar sobre una red homogénea, donde todos los nodos de la red cuentan con las mismas características de hardware, así como de la aplicación que tienen los dispositivos. De aquí que se observa una necesidad de contar con un esquema para reprogramar los nodos de una red de sensores, donde no todos los nodos de la red cuentan con las mismas características tanto de hardware como de software.

Al utilizar el medio inalámbrico para reprogramar los nodos de una red resulta bastante cómodo, y es un ahorro de tiempo el no estar reprogramando nodo por nodo físicamente, además de ser escalable. Sin embargo es importante tomar en cuenta el recurso mas preciado para estos dispositivos, que es la energía. Es importante que un esquema de reprogramación tome en cuenta este factor, y trate de minimizar su uso al máximo. Para esto un punto a tomar en cuenta es que generalmente al hacer una actualización, las dos versiones son muy parecidas. Es aquí donde surge un nuevo esquema que es la reprogramación incremental, donde se transmiten por la red solo aquellas porciones de código que no se encontraban en la versión anterior.

El presente trabajo pretende atacar estos problemas, brindando un mecanismo de reprogramación de redes de sensores, donde los nodos no necesariamente cuentan con las mismas características tanto de hardware como de software. También se minimiza el consumo de energía al transmitir solo aquellas porciones de código que no se encuentren ya en los programas instalados en los nodos, y al dejar al nodo a reprogramar la tarea de construir la nueva imagen con porciones de código del programa anterior.

I.3 Objetivos

El objetivo general de este trabajo es desarrollar un esquema de reprogramación para redes inalámbricas de sensores, utilizando las radiofrecuencias como medio de transporte de los programas y que se adapte a redes donde interactúan nodos con diferentes características, tanto físicas (distinto procesador), como lógicas (distinto comportamiento). Debido a que la comunicación a través de la unidad de radiocomunicación es uno de los procesos más costosos, se pretende disminuir la cantidad de paquetes a diseminar en la reprogramación. Para alcanzar este objetivo general es necesario lograr los siguientes objetivos específicos.

1. Analizar, diseñar e implementar un mecanismo para la formación de grupos *al vuelo*, que serán los diferentes grupos que compartirán algunas características. Un nuevo programa que será diseminado por la red tiene como objetivo reprogramar a un grupo de nodos sensores.

2. Analizar mecanismos existentes de enrutamiento, para formar las rutas por las cuales el nuevo programa será diseminado a los nodos interesados en reprogramarse. Adaptar el algoritmo de enrutamiento para darle apoyo a la reprogramación de los nodos en una red donde no todos los nodos ejecutan la misma

aplicación.

3. Diseñar e implementar un mecanismo que analice el nuevo programa a diseminar junto con los programas existentes en los nodos que se van a reprogramar, de tal forma que para diseminar el nuevo programa no se envíe completo, y cada nodo objetivo será capaz de formar el nuevo programa.

4. Diseñar e implementar el mecanismo para diseminar los paquetes que formarán al nuevo programa en cada nodo objetivo, aquellos que se van a reprogramar.

Para lograr los objetivos señalados es importante estudiar las implementaciones actuales, y los trabajos de investigación referentes a la reprogramación a través del aire, así como los diferentes mecanismos de enrutamiento multisaltos.

I.4 Contenido de la tesis

Este trabajo se presenta en seis capítulos. En el primer capítulo se muestra el planteamiento del problema y la forma en que se pretende darle solución. Esto último planteado en los objetivos específicos.

En el segundo capítulo se muestran las principales características de una red inalámbrica de sensores, se presentan algunos dispositivos utilizados para formar este tipo de redes y los principales sistemas operativos con que se está trabajando, tanto en la comunidad científica como en la industria. Además se presenta el estado del arte referente a la reprogramación de redes inalámbricas de sensores, mostrando las principales aportaciones y sus características.

En el tercer capítulo se muestra a mayor detalle el mecanismo de reprogramación en el que está basado este trabajo. En especial la manera en que disemina el nuevo

programa. Además se explica la forma en que un protocolo de enrutamiento multicast puede adaptarse al mecanismo de reprogramación para darle apoyo a la reprogramación de un conjunto de nodos, formando un árbol multicast por el cual se diseminarán las actualizaciones. Se entiende como multicast a la comunicación de un emisor a varios receptores.

En el capítulo cuatro se presenta la implementación realizada, junto con las principales características que brindarán apoyo para lograr los objetivos específicos.

En el quinto capítulo se muestran las pruebas realizadas a la propuesta de este trabajo, viendo su desempeño y las ventajas que produce el reprogramar una red de sensores de forma incremental. Se muestran las diferentes pruebas con una explicación sobre cada una de ellas.

En el capítulo seis se muestran las conclusiones finales de este trabajo, las principales aportaciones, así como las limitaciones con que cuenta. Además se muestran las posibles mejoras a la implementación realizada.

Capítulo II

Reprogramación de Redes Inalámbricas de Sensores

II.1 Redes Inalámbricas de Sensores

Las redes inalámbricas de sensores (WSN por sus siglas en inglés) están formadas por varios, incluso cientos o miles de nodos sensores que cuentan con unidad de procesamiento, poca memoria y capacidad de comunicación inalámbrica, y por supuesto dependiendo del fabricante cuentan con distintos tipos de sensores e incluso algunos cuentan con Sistema de Posicionamiento Global (GPS por sus siglas en inglés). Sin embargo estos recursos son muy limitados.

Dependiendo de la aplicación, los nodos obtienen lecturas de los sensores con que cuentan, procesan la información y son capaces de diseminar la información utilizando distintos nodos para enrutar los paquetes que recibirá un nodo central llamado *sink*. Una red de sensores es un tipo especial de una red *ad hoc*(Akyildiz *et al.*, 2002), sin embargo hay algunas diferencias significativas:

- El número de nodos en una red de sensores puede ser mucho mayor que en una red *ad hoc*.
- Los nodos sensores son emplazados densamente.
- Los nodos sensores son más propensos a fallas.
- La topología de una red de sensores puede variar con mayor frecuencia.

- La comunicación en una red de sensores está basada en comunicación por *broadcast*, en la mayoría de las redes *ad hoc* es comunicación punto a punto.
- Los nodos de una red de sensores cuentan con más restricciones en cuanto a energía, capacidades computacionales y memoria.
- Los nodos sensores pueden no tener un identificador global, debido al alto *overhead* y a la cantidad enorme de nodos sensores.

II.1.1 Aplicaciones

La investigación en redes de sensores fue originalmente motivada por aplicaciones para la milicia, sin embargo recientemente la industria se ha interesado cada vez más en ésta, entre algunas aplicaciones destacan(Chong y Kumar, 2003):

- Las redes de sensores pueden ser utilizadas para proteger construcciones importantes, aplicaciones antiterroristas para detectar ataques biológicos, quimicos o nucleares (Hills, 2001).
- Monitoreo ambiental y de hábitat. Algunos trabajos incluyen estudiar la vegetación o estar monitoreando cultivos, así como también obtener información importante de ciertas especies en el ambiente (Steere *et al.*, 2000).
- Sensado industrial.(Lee, 2001) La industria siempre se ha visto interesada en mantener sus equipos funcionando correctamente, es por eso que se utilizan ciertos tipos de sensores para estar revisando constantemente sus equipos y poder detectar posibles fallas antes de que sean mayores.
- Control de tráfico. Otra aplicación de una red de sensores es para monitorear el tráfico vehicular en ciertas avenidas, actualmente se utilizan videocámaras, sin embargo una red de sensores en ciertos lugares estratégicos podría minimizar los

costos y tanto el conteo de los vehículos como su velocidad podría ser calculado por los pequeños dispositivos(Estrin *et al.*, 1999).

Detección automática(Elson y Estrin, 2004), prevención y recuperación de desastres urbanos es uno de muchos usos potenciales para esta nueva tecnología.

II.1.2 Plataforma

Hardware

Existen varios dispositivos para redes de sensores de distintos fabricantes, algunos de ellos pueden comunicarse unos con otros, dependiendo de la unidad de radiocomunicación con que cuentan y al sistema operativo de soporte.

ESB/2

ESB (*Embedded Sensor Board*), diseñado por el grupo CST en Freie Universität (FU) en Berlin. Se pueden obtener en http://www.scatterweb.com. ESB es un dispositivo para redes de sensores basado en el chip MSP430 de Texas Instrument, está equipado con un radio transmisor-receptor TR1001 y un conjunto de sensores: un sensor detector de movimiento, un receptor infrarrojo, un transmisor infrarrojo, sensor de temperatura, sensor de vibración y un sensor de sonido (ver figura 1).

Figura 1: Nodo ESB/2

Tmote Sky

Es una plataforma de propósito general usado tanto en la investigación, como en la industria, es el sucesor de TelosA y TelosB desarrollados en UC Berkeley (ver figura 2). Actualmente se puede obtener en http://www.moteiv.com/.

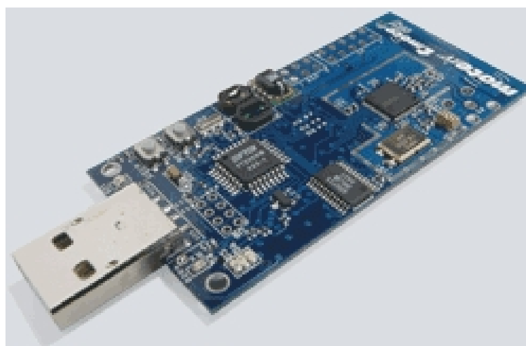

Figura 2: nodo Tmote Sky

Tmote Sky (Moteiv, 2006) está equipado con un microcontrolador MSP430 de TI. El dispositivo ofrece 10kb de RAM y 48kb de memoria flash, además tiene 1Mb de memoria flash externa. El radio es un chip CC2420 de Chipcon/Texas Instrument que permite mantener comunicación de 50m en interiores y hasta 125m en exteriores. Además del sensor de temperatura para el microcontrolador MSP430, el dispositivo permite agregar sensores de humedad, temperatura, así como también sensores de luminosidad. Tmote Sky obtiene su energía de dos pilas AA (1.5v), pero también puede obtener su energía a través del puerto USB con que cuenta.

BTnode

BTnode como se muestra en la figura 3 es una plataforma basada en la tecnología Bluetooth, se desarrolló para hacer investigación en redes móviles *ad hoc* y redes de sensores distribuidas.

BTNode fué desarrollado en ETH Zurich por el laboratorio de redes e ingenieria de computación y el grupo de investigación de sistemas distribuidos. La plataforma está equipada con un microcontrolador Atmel ATmega 128L, provee 128kb de memoria flash, 4kb de RAM. Se extendió la memoria SRAM a 64kb agregando un módulo de 60kb.

Además de contar con la tecnología Bluetooth, BTnode cuenta con un radio de bajo poder, el radio de bajo poder es un Chipcon CC1000 y trabaja a 868Mhz. Cada nodo cuenta con varios sensores y actuadores: un sensor de luminosidad y luz infrarroja, sensor de temperatura, dos LEDs, micrófono, sensor de aceleración en dos dimensiones.

Figura 3: Node BTnode

MicaZ

MicaZ es un producto de CrossBow Technologies Inc., basado en el chip Atmel ATmega 128L. Cuenta con un chip de radio CC2420 y tiene la capacidad de agregar varios sensores de manera fácil, entre los cuales están: acelerómetro, magnetómetro, temperatura, luz, acústica y la capacidad de agregar un GPS. MicaZ viene equipado con 128kb de memoria flash programable, una memoria flash externa de 512kb. El radio tiene la

capacidad de transmitir 250kbps y trabaja con dos pilas AA. En la tabla II.1.2 se comparan las características principales de los nodos que actualmente se pueden sonseguir en el mercado.

Tabla I: Características principales de algunos nodos sensores

Nodo	**CPU**	**Memoria**			**Energía**	**frecuencia de operación**	**tasa de transferencia**
		RAM	ROM	Otra			
ESB	+1Mhz	24kb	60kb	32kb	3 pilas AA	868Mhz	115.2kbps
Tmote Sky	8Mhz	10kb	48kb	1Mb Flash	2 pilas AA	2.4Ghz	250kbps
BTNode	8Mhz	4+(60)kb	128kb	4kb	2 pilas AA	868Mhz 2.4 Ghz	76.8kbps 723kbps
MicaZ	8Mhz	4Kb	128kb	512kb	2 pilas AA	2.4Ghz	250kbps

Sistema Operativo

Un aspecto importante en la investigación sobre redes de sensores es el sistema operativo disponible. El modelo de concurrencia y la abstracción que provee impacta en el proceso de diseño y desarrollo de una aplicación. Así como existe una gran variedad de dispositivos, también hay diversos sistemas operativos para ellos, no todos los sistemas operativos soportan todas las plataformas.

TinyOS 1.x

Es uno de los primeros ambientes diseñado específicamente conociendo los requerimientos de restricción de recursos, orientado a eventos y para sistemas de red empotrados. Desarrollado en la Universidad de Berkeley, California. Muchas de las características de TinyOS provienen de su lenguaje de implementación llamado nesC. NesC es una extensión de C, adaptado a las necesidades de dispositivos de red empotrados.

Una de las principales características del modelo de programación es la modularización de componentes. La forma de interactuar entre los componentes es por medio de interfaces, especificando puntualmente un enlace entre los componentes, las interfaces

que proveen y/o utilizan.

TinyOS provee dos niveles de concurrencia: tareas y eventos (*task*, *events*): Tareas son mecanismos para pequeños procesos computacionales donde no se relacionan operaciones de bajo nivel, las tareas no afectan la ejecución de otras tareas. Los Eventos pueden ser ejecutados mientras ocurre la ejecución de una tarea y/o evento. Gran parte del procesamiento de TinyOS inicia con eventos representados por interrupciones de hardware.

Un aspecto importante en TinyOS es el lenguaje estático, el cual no permite la asignación dinámica de memoria, esto para evitar en lo posible algunos errores y en tiempo de compilado nesC pueda realizar un análisis de seguridad y optimización de desempeño.

Los mensajes activos (*Active Message*) son la principal abstracción de comunicación. Consiste en un pequeño identificador agregado a cada mensaje, especificando la acción que necesita ejecutarse cuando se reciba el mensaje. Una analogía utilizada para definir un mensaje activo es el concepto de puerto para la pila de protocolos TCP/IP. Las operaciones de bajo nivel para enviar y recibir datos están encapsuladas en una abstracción llamada *GenericComm*, la cual provee mensajes *unicast* a un salto, y mensajes *broadcast*.

TinyOS 2.0

El predecesor TinyOS 1.x fue creciendo y madurando al mismo tiempo que se iba necesitando, se fue adaptando en el progreso, y al mismo tiempo los encargados de dar soporte se daban cuenta que algunos requerimientos de diseño eran más importantes

que otros. Por la dificultad de rectificar y readaptar tinyOS 1.x a las nuevas necesidades, se decidió reescribir gran parte para formar TinyOS 2.0 conociendo ya más a fondo los requerimientos y estructurarlo de mejor forma para facilitar la portabilidad y mejorar la robustez y confiabilidad.

TinyOS 2.0 tiene mayor flexibilidad en muchos aspectos. Introduce una arquitectura de abstracción de hardware de tres niveles (HAA por sus siglas en ingles). Se introduce el concepto de *chip* para permitir una descomposición vertical de la abstracción de hardware. Otra mejora de TinyOS 2.0 con respecto a su predecesor, es la utilización de la versión 1.2 del lenguaje nesC, el cual contiene nuevas características para permitir la comunicación en la red a través de múltiples plataformas. Esta versión introduce un tipo de dato de red, para especificar el formato de los paquetes y evitar hacer *marshalling/unmarshalling* en código, es decir, modificar el formato de datos para transmitirlos por la red y al llegar a su destino regresarlos a su formato original.

Contiki

Contiki OS es un proyecto de software libre, portable, sistema operativo multi-tareas para sistemas restrictivos en recursos. Contiki permite tres modelos de concurrencia: *events, threads y protothreads*, el kernel de Contiki está basado en eventos permitiendo que las aplicaciones puedan usar cualquiera de los tres modelos de concurrencia o una combinación de ellos.

El núcleo de Contiki provee abstracción para un gran número de dispositivos de hardware encontrados comúnmente en plataformas para redes de sensores, como: Leds, radio, flash ROM, EEPROM, UART, temporizadores y memoria externa.

BTnut

BTnut es un sistema operativo basado en C, provee el software necesario para la plataforma BTnode. BTnut fué creado encima de Nut/OS, un sistema operativo de propósito general para dispositivos embebidos. BTnut implementa multihilos cooperativos, ya que el procesador puede estar ejecutando un hilo a la vez, BTnut provee calendarización para hacer cambios de contexto. Los eventos son un concepto fundamental en el sistema operativo NutOS, cuando un hijo en ejecución necesita la ocurrencia de un evento, deja sus recursos en una cola de eventos. El hilo continuará su ejecución una vez que el evento esperado sea puesto en la cola de eventos por otro.

La ejecución de un hilo puede ser detenida por interrupciones del CPU, al terminar la rutina de interrupción éste continuará con su ejecución. BTnut provee temporizadores de un disparo (*one shot*) y periódicos. Provee asignación dinámica de memoria. No existen primitivas estándares para la comunicación, pero se proveen servicios para comunicación por radio Bluetooth y radio Chipcon.

II.2 Reprogramación de Redes Inalámbricas de Sensores

En un principio los pequeños nodos son programados de uno en uno, algunos cuentan con interfaz por el puerto paralelo, serial, incluso algunos nuevos nodos cuentan con interfaz USB. Con la ayuda de un HUB USB es posible programar más de un nodo a la vez, pero aún así, para redes con un número de nodos considerablemente grande, resulta una tarea muy tediosa. Existen varias características(Wang *et al.*, 2006) deseables en una arquitectura de reprogramación de una red inalámbrica de sensores, entre las principales se encuentran:

- **Control de versiones**: Un sistema de control de versiones mantiene una base de datos con la información de las distintas versiones de los programas para los nodos sensores, esto sirve para poder generar parches cuando sea necesario.

- **Selección de nodos**: Una red de sensores puede estar formada por nodos que realizan distintas actividades, incluso con distinto programa, esto influye al momento de reprogramar los nodos, ya que es necesario localizar los nodos que se van a reprogramar.

- **Preparar/Identificar información de programas**: Antes de enviar un nuevo programa, en la estación base se preparan los paquetes que se enviarán a los nodos, tanto de anuncios del nuevo programa, como paquetes de datos que forman el programa. Además se añade información necesaria para que los nodos escriban los datos contenidos en cada mensaje en la dirección correcta para almacenar el nuevo programa.

- **Diseminación de código**: El código se va diseminando a través de los nodos, si es un mecanismo multi saltos, para esto es necesario un protocolo de diseminación del nuevo programa y de los mensajes de control de la reprogramación.

- **Validación al completar la recepción**: Al momento de recibir el nuevo programa, es necesario verificar que es un código correcto antes de realizar alguna otra actividad como reiniciar con el nuevo programa.

- **Solicitud de programa**: Cuando un nodo detecta la existencia de un nuevo programa en la red, por lo general por anuncios, el nodo debe establecer comunicación con un nodo próximo a él que contenga el nuevo programa para solicitárselo.

- **Intercambio de programa**: Al recibir el nuevo programa, el nodo usualmente lo almacena en una memoria externa, el nodo debe ser capaz de reiniciar con el

Tabla II: Comparación de diferentes arquitecturas de reprogramación

Nombre	**Envio del programa**	**selección**	**Saltos**	**Pipeline**
XNP	Programa completo	Toda la red	Un salto	NO
Trabajo de Reijers	Parche independiente de plataforma	Toda la red	Un salto	NO
Incremental	Parche dependiente de plataforma	Toda la red	Un salto	NO
Trickle	Scripts de Maté	Toda la red	Multisaltos	NO
MOAP	Programa completo	Toda la red	Multisaltos	NO
Deluge	Programa completo	Toda la red	Multisaltos	SI
MNP	Programa completo	Toda la red	Multisaltos	SI
Acqueduct	Programa completo	Selección de nodos	Multisaltos	SI
FlexCup	Actualización modular	Selección de nodos	Multisaltos	NO

nuevo programa o cualquier programa con que cuente en su memoria externa.

II.2.1 Mecanismos de Reprogramación

Existen distintos trabajos enfocados a la reprogramación de los nodos en una red inalámbrica de sensores, con algunas diferencias entre ellos en cuanto a la manera en representar la información, la forma en que retransmiten los nuevos programas o simplemente en la plataforma de trabajo donde se emplean, que indudablemente hace que el mecanismo de reprogramación sea distinto.

Deluge(Hui y Culler, 2004) es un mecanismo de reprogramación desarrollado bajo TinyOS (Hill *et al.*, 2000). Para manejar el gran tamaño de los datos, Deluge divide el nuevo programa en páginas de un tamaño definido. Esta página es la unidad básica que se transfiere. El nodo debe conocer los paquetes que ha recibido y los que necesita para completar el objeto. Se maneja un vector de bits de los paquetes de las páginas que se han recibido para poder detectar paquetes que no llegaron. Los paquetes y las páginas incluyen 16 bits de CRC (cycling Redundancy Check).

Un nodo opera en uno de los tres estados siguientes: Mantenimiento, RX (Recibiendo), TX (Transmitiendo)(ver figura 4).

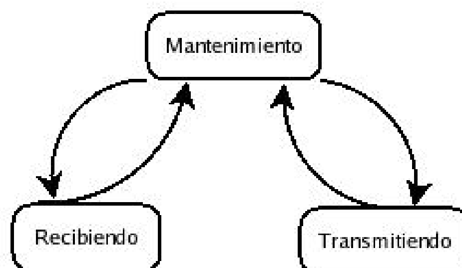

Figura 4: Estados de Deluge

Deluge se basa en *Trickle* (Levis *et al.*, 2004) para controlar la transmisión de mensajes potencialmente redundantes. *Trickle* divide el tiempo en series de *rounds* y los nodos eligen si enviar o no por broadcast un anuncio en cada *round*.

Un nodo al contener una página completa puede enviar anuncios, sin importar que todavía no cuente con toda la imagen completa del nuevo programa , esto permite realizar un *pipeline*, agilizando la transmisión del nuevo programa a todos los nodos.

En MNP (Kulkarni y Wang, 2005) se usan dos tipos de mensajes para la selección del nodo fuente: peticiones de *advertisement* y *download*. Un mensaje de *advertisement* contiene información del nuevo programa y del nodo fuente, éste tiene dos objetivos: anunciar la llegada de un nuevo programa y prevenir a los nodos que han recibido menos peticiones en convertirse en retransmisores.

El mensaje *download* además contiene el valor de *ReqCtr* que se envió a él en la fase

de *advertisement*. Mientras la petición de *download* está destinada a k[1], Se manda el mensaje por broadcast, así cuando otros nodos (e.g l) reciban la petición de *download* se pueden dar cuenta de que k es un transmisor potencial.

Cuando un nodo termina de transmitir el nuevo código, se quita de la competencia y se pone a *dormir* por un rato, así otros nodos tienen oportunidad de volverse *senders*.

MAOP (Stathopoulos *et al.*, 2003) disemina el nuevo programa de vecindario en vecindario en un solo salto, y al final de cuentas se reduce a muchos saltos de manera recursiva, en un vecindario se suprimen los retransmisores al momento en que se detecta que ya existe uno, de esta manera sólo se utiliza un solo nodo retransmisor en cada vecindario. Se divide el programa en segmentos igual que en MNP(Kulkarni y Wang, 2005). El protocolo de diseminación es llamado Ripple, la política de retransmisión es unicast y para la administración de segmentos se utilizan ventanas deslizantes. El procedimiento de MAOP es el siguiente:

- Se empaqueta el programa a diseminar en segmentos.
- Un mote conectado a una PC es el primer fuente.
- Envía mensajes de publicación anunciando el nuevo código
- Los nodos revisan la versión y se suscriben.
- Cuando un nodo tiene la imagen completa, envía mensajes de publicación, y éste se convierte en fuente.
- Después de transmitir, se espera un determinado tiempo para posibles retransmisiones y después invoca al *bootloader* para arrancar con el nuevo código.

[1] j recibe un mensaje *advertisement* de k

- Cuando un nodo detecta un segmento perdido usando ventanas deslizantes, pregunta al fuente por retransmisión usando mensajes unicast.

- Si no recibe respuesta, envía por broadcast una petición de reparación.

- Todos los nodos envían periódicamente mensajes de publicación. por si hay un nodo nuevo que no tiene la nueva imagen o si se recuperó de algún error.

FlexCup (Marrón *et al.*, 2006) es un mecanismo de reprogramación modular utilizado en el *Framework TinyCubus*(Marrón *et al.*, 2005). FlexCup divide el nuevo programa en módulos, lo que permite una reprogramación incremental. El nodo a reprogramarse realiza trabajo extra al ligar los componentes binarios, debido a que utiliza la tabla de símbolos para lograrlo. FlexCup debe ser la única aplicación corriendo ya que absorbe grandes cantidades de memoria en ese momento.

Aqueduct (Phillips, 2005) es una modificación al funcionamiento de Deluge(Hui y Culler, 2004) agregando un estado más, el de retransmisor, para dar soporte a redes heterogéneas, con este nuevo estado es posible formar ductos por los cuales se diseminarán los programas.

En el estado de mantenimiento, igual que en Deluge, se envían anuncios con información de la página disponible, la versión y se agrega un campo más, distancia al nodo miembro del grupo a reprogramar, el estado de transmitiendo y recibiendo son idénticos a Deluge. El estado retransmitiendo es un intercambio entre estado de transmitiendo y recibiendo, almacenando en cache algunos paquetes para posibles mensajes de solicitud de paquetes. Otra modificación a Deluge es el agregar a los mensajes de anuncios una lista de vecinos para formar los ductos sobre enlaces cortos simétricos, y evitar caminos que pueden ser más cortos pero con enlaces no muy confiables.

II.2.2 Programación Incremental

Dentro de los mecanismos de reprogramación, existen algunos trabajos para disminuir la transferencia de la información, entre algunos trabajos se encuentran las máquinas virtuales, y sólo se diseminan pequeños módulos de aplicaciones, o la generación de parches con algoritmos cómo rsync o diff.

Reijers et al(Reijers y Langendoen, 2003) solo distribuyen los cambios del código actual corriendo en los nodos. El nuevo programa se crea usando un *script* de comandos que son fáciles de procesar por los nodos sensores. Algunos requerimientos a tomar en cuenta al momento de crear un esquema de actualización de los nodos sensores son:

- El esquema de distribución del código debe actualizar todo el código en el nodo sensor.
- El esquema debe asegurar que todos los paquetes lleguen a todos los nodos, y utilizar esquemas de recuperación de paquetes perdidos no muy costosos.
- El esquema de distribución debe minimizar el uso del radio al máximo.
- El procesado en los nodos no debe ser excesivo.
- Cuando se actualice código se desea que la aplicación corriendo en el nodo se detenga el menor tiempo posible.

Se utiliza un algoritmo como el diff de Unix para obtener el script que será enviado a los nodos para que rehaga el nuevo programa, un problema importante es que pequeños cambios en código fuente producen grandes cambios al código binario, esto es debido al direccionamiento de funciones. Una manera de solucionar o disminuir este problema es crear espacios entre las funciones al momento de crear el binario, para que así al momento de alguna modificación, ya sea que crezcan las funciones o disminuyan, haya

espacio suficiente para que el direccionamiento cambie lo menos posible.

El diseño presentado en el trabajo por Jeong y Culler (2005) no asume ningún conocimiento de la estructura del programa y puede ser aplicado a cualquier plataforma de hardware.

El mecanismo de reprogramación envía la nueva versión de un programa transmitiendo la diferencia entre dos imagenes de programas, comparando el código del programa en niveles de bloques sin ningún conocimiento previo de la estructura de la aplicación. Lo anterior lo hace utilizando el algoritmo Rsync (Trigdell, 1999) para generar las diferencias, rsync busca bloques de código compartidos entre dos programas y permite distribuir solo los cambios del programa.

Jeong y Culler (2005) modificaron el algoritmo rsync para adaptarse a redes de sensores. Primero se hacen operaciones costosas como construir la tabla hash en una computadora base, y el nodo solo necesita leer o escribir bloques de código en la memoria flash para reconstruir la nueva imagen.

La estación base tiene el historial de las versiones de los programas que tienen los nodos, entonces compara el nuevo programa con el programa anterior utilizando una tabla hash.

Se entiende como tabla hash a una estructura de datos que asocia llaves con valores. Estas llaves se generan utilizando una función hash que resume un conjunto de datos dando como resultado una llave de dimensiones más pequeña, con la característica que dada la llave, es imposible generar el valor de entrada, y dos llaves hash provienen de dos valores distintos.

La comparación de los dos programas se hace en base a lo siguiente (ver figura 5):

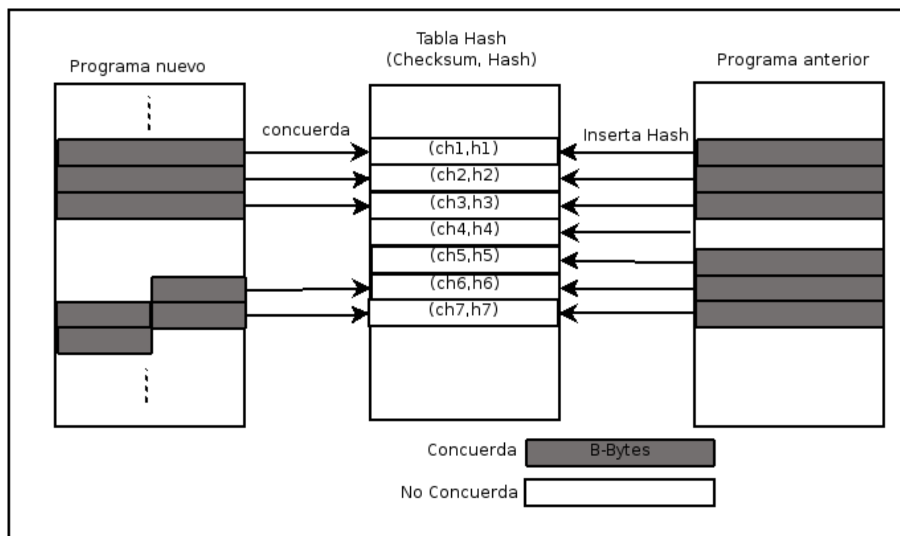

Figura 5: Implementación de rsync (Jeong y Culler, 2005)

- El algoritmo rsync calcula un par checksum (checksum, hash) para cada bloque de tamaño definido (B Bytes) de la versión anterior y el checksum par es insertado en una tabla hash

- Rsync lee la versión actual y calcula el checksum de B bytes en cada byte. Si encuentra un checksum igual en la tabla hash, rsync calcula el hash del bloque y lo compara con el correspondiente en la tabla hash. Si el hash es igual se considera que el bloque es común en las dos versiones.

- Rsync se mueve al siguiente byte para comparar, si el bloque no concuerda en el checksum o en la tabla hash. Una región de bytes que no concuerda es etiquetada

como *non-matching block* y necesita ser enviado

El programa en el host describe la diferencia con dos operaciones *copy* y *download*. El tamaño del bloque de un mensaje *copy* es múltiplo del tamaño de la linea SREC, y el del bloque de un mensaje *download* pude ser de cualquier dimensión.

Las actividades básicas de los nodos son las siguientes:

- El nodo sensor pone en cola todos los scripts comandos que le llegan
- El nodo sensor revisa si hay algún registro perdido para solicitarlo
- El nodo sensor decodifica el script después de recibir un mensaje *decode*

Otro trabajo de reprogramación incremental es utilizando lo que le llaman ligado incremental remoto(Koshy y Pandey, 2005). En este trabajo se trata de minimizar el envío de nuevos programas enviando actualizaciones *diff-like*. Las actualizaciones son codificadas en *diff-scritp* y son inyectadas a la red. Modificaron el procedimiento de ligado de objetos al compilar un nuevo programa para facilitar la generación de scripts *diff* más concisos llamados *deltas*.

Este modelo se acerca mejor a la intuición de que la reprogramación es esencialmente el religado y reemplazamiento de los módulos modificados, por lo tanto se tiene un conocimiento de la estructura del programa.

Al modificar el compilador para lograr una reprogramación incremental se minimiza el número de paquetes que se envían de un nuevo programa, por lo tanto se minimiza el uso del componente de radiocomunicación y el gasto de energía. Además el acceso a la memoria flash se reduce debido a que las funciones que no cambiaron no se mueven,

esto es gracias a que el espacio para cada función se incrementa dejando espacios vacios entre una función y otra, logrando que una función que si se modifica pueda crecer sin afectar el direccionamiento de las demás funciones. Si una función cree mas allá del espacio vacio, se coloca en uno más grande.

En este esquema las instrucciones *diff* son generadas solo para las páginas modificadas. Un ligado incremental religa solo esos módulos que han cambiado desde el religado anterior. Debido a que el ligado es costoso en procesamiento, el ligado se hace de forma remota, es decir en una computadora, y se generan pequeños *Deltas* que son distribuidos a los nodos. Para generar los *Deltas* se utiliza el algoritmo ***XDelta***.

FlexCups(Marrón *et al.*, 2006) implementa un algoritmo de actualización que permite retransmitir sólo los componentes de un programa que fueron modificados, para esto FlexCups genera meta-información al momento de compilar que describen los componentes del programa a diseminar, esta meta-información es necesaria para actualizar los componentes en la aplicación corriendo, reenlazando llamadas a funciones a su dirección correspondiente. Con la ayuda del compilador nesC (nesC 1.2) es posible compilar un conjunto de componentes en objetos para generar un componente binario. Estos componentes binarios pueden ser enlazados como los componentes tradicionales de nesC. FlexCup usa el concepto de componentes binarios y lo mejora de tal forma que el proceso de ligado se realice en el nodo sensor. Para que FlexCup integre los nuevos componentes es importante que cuente con meta-información, que almacena en la memoria Flash externa, esta meta-información consiste de tres partes: información general del programa, la tabla de símbolos, y una tabla de reasignación para cada componente binario del programa. El algoritmo puede ser separado en cinco pasos:

- *Almacenamiento de código y meta-información:* Esta fase incluye la transmisión

de los cambios entre los nuevos componentes binarios y los almacenados en los nodos. También se disemina y almacena la meta-información.

- *Combinar la tabla de símbolos:* Esta fase involucra el combinar la nueva tabla de símbolos enviada con la tabla de símbolos almacenada en memoria, un problema con este proceso es que absorbe 3kb de memoria de los 4kb con que cuentan la plataforma MicaZ donde realizaron los experimentos. Por esta razón es importante que FlexCup sea la única aplicación corriendo en los nodos al momento de actualizar los programas.

- *Reasignación de la tabla de redireccionamiento:* Se envía también por cada componente binario su tabla de redireccionamiento, este paso solo involucra reemplazar la tabla de redireccionamiento anterior por la nueva.

- Parchado de referencias: Este paso involucra revisar las tablas de redireccionamiento de todos los componentes y ver cuales referencias necesitan actualizare.

- Instalación y reinicio: El último paso consiste en copiar el programa de la memoria flash externa a la memoria interna, donde se almacena el programa a ejecutarse, y reiniciar.

Bombilla, antes Maté (Levis y Culler, 2002) es un interprete *bytecode* pequeño que corre encima de TinyOS (Hill *et al.*, 2000), y los programas hechos con bombilla son muy cortos. El código está dividido en cápsulas de 24 instrucciones, y cada instrucción es un *long byte*, programas grandes pueden estar compuestos por múltiples cápsulas. El límite de 24 instrucciones permite que cada cápsula pueda estar empaquetada en un simple mensaje de TinyOS y evita utilizar un *buffer* para almacenar porciones de cápsulas en RAM. Cada cápsula incluye además su tipo y su versión. Existen cuatro

tipos de cápsulas: Cápsulas de envío de mensajes, cápsulas de recepción de mensajes, cápsulas de temporizadores (*timer capsules*), y cápsulas de subrutinas.

Se cuenta además con un inyector de cápsulas, que las inyecta a un nodo conectado a la PC; si la cápsula está marcada para que se auto envíe, se diseminará a través de la red.

El sistema de reenvío de cápsulas se ejecuta aproximadamente cada 700 milisegundos, si Bombilla decide reenviar una cápsula, elige una de manera aleatoria y revisa si es una cápsula marcada como auto envío; si es así transmite la cápsula por *broadcast*.

Cada cápsula contiene un número de versión, cuando un nodo con bombilla recibe una cápsula, verifica si es más nueva a la que tiene instalada; si es así, bombilla detiene su ejecución e instala la nueva cápsula.

La primera implementación de bombilla satura fácilmente la red, debido que a pesar de que cada nodo cuenta con la misma versión de cápsula, constantemente la retransmite por *broadcast*.

II.2.3 Diseminación de nuevos programas

Un proceso importante en toda reprogramación o actualización de programas en los nodos sensores, es cómo se envían los programas o en que protocolos se basan para lograr que todos los nodos en una red, sea la topología que fuese, lleguen a recibir el nuevo programa o sus actualizaciones para poder así, reprogramarse.

II.3 Resumen

A pesar de existir muchas soluciones para reprogramar una red inalámbrica de sensores, es muy dificil que una sola de ellas solucione todos los problemas o se adapte a cualquier implementación de una red. Esto es debido a que el diseño de la aplicación con que va a contar la red depende mucho, la topología de la red, o inclusive el tipo de hardware y sistema operativo dependen mucho del problema que se quiere solucionar. En el siguiente capítulo se explica con más detalle la forma en que se diseminan nuevos programas, en especial con la implementación Deluge. Además se muestran algunos protocolos de enrutamiento que podrán ser adaptados a un mecanismo de diseminación de nuevos programas, como Deluge.

Capítulo III

Diseminación de Código

III.1 Introducción

En este capítulo se estudia Deluge, el cual es un mecanismo de diseminación de actualizaciones, la manera en que diseminan los paquetes y sus características principales. Además se estudia un protocolo de enrutamiento multicast que podrá ser adaptado a una red de sensores para reprogramar no toda la red, sino seleccionar grupos de nodos. Con el protocolo de enrutamiento multicast se formará el árbol de enrutamiento por el cual van a pasar los paquetes de la nueva actualización para así llegar al grupo de nodos seleccionados a reprogramar.

En particular, se estudia un protocolo de enrutamiento multicast llamado ADMR, el cual, por sus características se puede adaptar a un mecanismo de reprogramación para diseminar nuevos programas a un grupo de nodos de la red de sensores.

III.2 Diseminación de actualizaciones

Un proceso importante en toda reprogramación o actualización de programas en los nodos sensores, es cómo se envían los programas o en que protocolos se basan para lograr que todos los nodos en una red, sea la topología que fuese, lleguen a recibir el nuevo programa o sus actualizaciones para poder así, reprogramarse.
Firecracker (Levis y Culler, 2004) utiliza una combinación de enrutamiento y envío de paquetes por *broadcast* para enviar rápidamente información a cada nodo en una red.

El protocolo tiene dos fases: la primera se enruta la información a algunos nodos en la red, los nodos intermedios de esas rutas almacenan la información. La segunda fase se ejecuta cuando el nodo recibe la información, utiliza un protocolo de diseminación como Trickle (Levis *et al.*, 2004), un protocolo basado en broadcast, para diseminar la información a todos los nodos.

Para disminuir el consumo de energía, Firecracker busca minimizar los conflictos y contención en la red. El envío por broadcast empieza segundos después del enrutamiento.

Trickle (Levis *et al.*, 2004) es un algoritmo de propagación de código y mantenimiento en redes inalámbricas de sensores, Entre sus características principales es que envía constantemente metadatos por *broadcast* para detectar si un nodo en la red necesita actualizarse. Cuando un nodo escucha metadatos de un código más viejo del que él tiene, éste envía el nuevo código para mantener al otro nodo actualizado.

Cuando los nodos no escuchan mensajes con información nueva, los nodos envían los metadatos de forma menos frecuente, pero en cuanto reciben nueva información, empiezan los nodos a enviar los mensajes con mayor frecuencia.

Para evaluar el funcionamiento de trickle se implementó sobre Maté (Levis y Culler, 2002) ahora Bombilla. Sin embargo el protocolo de diseminación de actualizaciones Deluge (Hui y Culler, 2004) utiliza también la mayoría de sus características.

III.3 Diseminación de Actualizaciones en Deluge

DelugeHui y Culler (2004) es un protocolo epidémico y trabaja como una máquina de estados, como se puede ver en la figura 6, donde cada nodo mantiene un conjunto de reglas para lograr el comportamiento deseado.

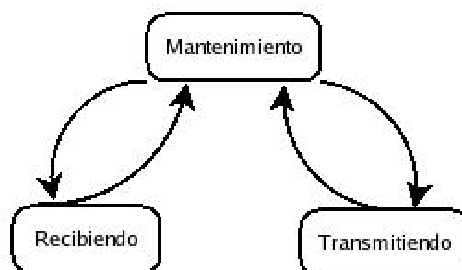

Figura 6: Estados de Deluge

Cada nodo envía ocasionalmente anuncios de su última versión de los programas que contiene mediante broadcast; si S recibe un anuncio de un nodo R con versión más vieja, S responde con un anuncio que contiene el perfil de sus programas. Con el perfil, R determina que porción de información necesita para actualizar y envía una solicitud al nodo S que envió el anuncio. S envía la información solicitada por broadcast para que cualquier nodo a su alcance que necesite esa información, incluyendo R la reciban. Cuando R recibe completa la información de su solicitud, envía anuncios de su perfil para propagar la información a los nodos que la requieran.

El perfil que disemina un nodo contiene:

- Identificador único de la imágen.
- Versión de la imágen.

- Número de la imagen, volúmen donde se encuentra almacenada la imagen en EEPROM.
- Número de páginas que se encuentran almacenadas.
- Número de páginas de la imagen completa.

El perfil se almacena en una estructura llamada `ImageDesc`. La información del perfil almacenado en la estructura se representa como se puede ver en la figura 7.

Perfil de la Imágen
uid: 4 bytes vNum: 2 bytes imgNum: 1 byte numPgs: 1 byte numPgsComplete: 1 byte

Figura 7: Perfil de una imágen

III.3.1 Representación de la información en Deluge

Para manejar grandes cantidades de información, como es la imagen de un programa completo, Deluge divide la imagen en páginas de tamaño fijo. La página es la unidad básica de transferencia y una de las principales ventajas es limitar el tiempo que un nodo se encuentra en el estado de recibiendo información, otra ventaja es el poder diseminar cierta página, aún cuando no se cuenta con la imagen del programa entero, siempre y cuando se cuente con la página completa como se ve en la figura 8.

La información de un programa de tamaño S_{img} es dividida en páginas de tamaño fijo $S_{pag} = N \cdot S_{pkt}$, donde N es el número de paquetes, cada paquete es de tamaño fijo S_{pkt}, como se ve en la figura 9. Para verificar cuáles paquetes se necesitan para completar la imagen, el nodo cuenta con un vector de bits de la página con la que está

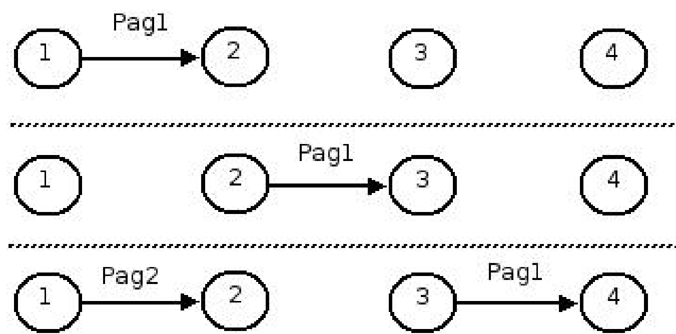

Figura 8: Diseminación de páginas

trabajando, y cada bit representa un paquete de dicha página. El vector de bits debe ser solo de tamaño N-bits, donde N es el número de paquetes en una página.

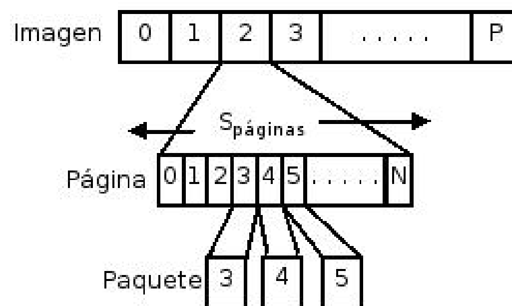

Figura 9: Representación de la información de un programa

Los paquetes y las páginas cuentan con control de redundancia cíclica (CRC's). Si un paquete o una página falla la revisión de CRC, toda la información representada por ella se descarta y necesita ser recibida de nuevo. Esta revisión es importante para Deluge debido a la diseminación epidémica de la naturaleza del protocolo para evitar que un pequeño error se vaya diseminando a todos los nodos.

Deluge puede soportar enviar perfiles de varias imágenes de programas contenidas en

memoria EEPROM, dependiendo de la arquitectura sobre la cual está corriendo Deluge y la capacidad que se tiene en memoria. Al momento de iniciar un programa con Deluge, éste verifica la memoria EEPROM para obtener la descripción de las imágenes que están almacenadas y envía anuncios por cada imagen.

III.3.2 El protocolo

Un nodo opera en uno de los tres estados posibles en un momento dado: Mantenimiento, Recibiendo y Transmitiendo.

Mantenimiento

Un nodo en estado de Mantenimiento es responsable de asegurar que todos los nodos en su rango de comunicación tienen (i) la última versión del perfil de la imagen y (ii) toda la información necesaria para la última versión. Para mantener esta propiedad, cada nodo anuncia información del perfil y el número de páginas disponibles para retransmisión, i.e, las páginas que tiene que están completas.

Una contribución importante de Deluge es poder diseminar información de imagen sin importar que no tenga la imagen completa, siempre y cuando contenga en memoria al menos la página completa y que haya sido revisado el CRC, Deluge envía anuncios de la última página completa aún antes de tener la imagen lista. El rendimiento de procesamiento total es incrementado debido al *pipeline* transfiriendo páginas a través de la red.

Recibiendo

Un nodo en estado recibiendo es responsable de solicitar los paquetes requeridos para completar la página $p = \gamma + 1$, donde p es la página a recibir y γ es la página disponible

que se puede retransmitir. Cada solicitud trabaja como un *selective negative acknowledgement* (SNACK) donde un vector de bits especifica cual paquete en la página es necesario. La solicitud se hace en tiempo aleatorio para que disminuya la probabilidad de que ocurran colisiones. La respuesta a la solicitud es enviada por broadcast para permitir que nodos en el mismo rango que necesiten la misma página, aprovechen y capturen los paquetes necesarios.

Transmitiendo

Un nodo en estado transmitiendo es responsable de enviar por broadcast todas las solicitudes de paquetes de cierta página, y regresa al estado de mantenimiento cuando termina de atender la solicitud.

III.4 Enrutamiento sobre demanda

Como se ha visto en este capítulo, un proceso importante para reprogramar es la forma en que se diseminarán las actualizaciones.

En lo que resta de este capitulo se estudia el enrutamiento sobre demanda y se plantea de que forma se puede adaptar un enrutamiento con estas características a la tarea de realizar actualizaciones y las ventajas que ésto ofrece.

Los protocolos de enrutamiento sobre demanda están clasificados dentro de los protocolos reactivos, y su principal característica es la capacidad de formar las rutas sólo cuando es necesario.

Cuando un nodo fuente necesita enviar cierta información a un nodo en particular,

envía un mensaje por broadcast haciendo una solicitud de ruta. En cada nodo intermedio, cuando se recibe un mensaje de solicitud de ruta, se forma la trayectoria al nodo fuente. Si el nodo no había recibido ese mensaje, no es el nodo destino, y no tiene una ruta hacia el destinatario, reenvía la solicitud. Si el nodo que recibe el mensaje es el destinatario, envía un mensaje de contestación al nodo que originó el mensaje de forma *unicast*, el mensaje de contestación se va reenviando salto por salto hasta llegar al primer nodo que originó el mensaje, el nodo fuente. Si se reciben varios mensajes de solicitud de ruta, se elige la ruta con el menor número de brincos al nodo fuente.

Entre los protocolos de enrutamiento reactivos existentes se encuentra *Ad-hoc On-demand Distance Vector (AODV)* de Perkins y Royer (1999) para redes *ad-hoc*, y puede ser utilizado para enrutar mensajes *unicast* o *multicast* (como se mencionó anteriormente el componente de enrutamiento entra a operar solo cuando hay información para enviar).

Una de las principales ventajas de este tipo de protocolos de enrutamiento es que pueden ser empleados en redes con cualquier topología, donde la movilidad de los nodos podrían afectar tablas de enrutamiento estáticas.

Para el protocolo de enrutamiento *Dynamic Source Routing(DSR)*, en cada mensaje de solicitud de ruta va además de la dirección del nodo fuente y del nodo destino, un registro de los saltos por los cuales se va enrutando ese paquete; de esta forma, cuando llegue el paquete de solicitud de ruta al nodo, éste responde a la solicitud reenviando el registro por los cuales va a pasar el paquete. Cuando el mensaje de respuesta llega al nodo que originó la solicitud de ruta, éste agrega en el encabezado del paquete de información que desea enviar, los saltos que debe contener el paquete para llegar al objetivo.

Paralelamente mientras se hace una solicitud de ruta para enviar cierta información, los nodos van guardando en caché la información necesaria para enrutar nuevos paquetes. Así cuando un nodo desee enviar información, si ya cuenta en cache con información para enrutar paquetes, no necesita mandar mensajes de descubrimiento de rutas.

Para lograr el objetivo principal de esta tesis y reprogramar sólo un grupo de nodos en lugar de la red entera, es necesario identificar las rutas por las cuales se diseminarán los nuevos programas, es por ello que se estudiaron algunos protocolos de enrutamiento.

Se identificó que al ser la reprogramación una actividad poco frecuente, además que no necesariamente siempre se reprograman los mismos nodos, un protocolo de enrutamiento multicast sobre demanda se adecúa perfectamente a nuestro problema, ya que una vez logrado el objetivo principal (reprogramación) se pueden desechar las tablas de enrutamiento y cuando se requiera reprogramar de nuevo otro conjunto de nodos, se vuelve a hacer la solicitud de rutas.

El protocolo de enrutamiento multicast que se seleccionó para estudiar más a detalle, es ADMR (Adaptive Demand-driven Multicast Routing), debido a sus características en comparación con otros protocolos sobre demanda. En la siguiente sección se profundiza dicho protocolo.

III.4.1 ADMR: Enrutamiento Multicast Adaptativo Sobre Demanda

El protocolo de enrutamiento multicast adaptativo sobre demanda ADMR (por sus siglas en inglés) es un protocolo diseñado para trabajar sobre redes *ad-hoc*.

La inicialización de ruta en redes *ad-hoc* puede ser de dos maneras:

Iniciado por el transmisor : Cuando un nodo necesita enviar información otro o a un grupo de nodos en particular, el nodo fuente empieza a enviar mensajes por *flood* controlado para llegar al grupo de nodos destino que recibirán la información. Este caso aplica al momento de querer reprogramar un conjunto de nodos de una red.

iniciado por el receptor : Cuando un nodo necesita información de algún nodo fuente en particular, empieza a mandar mensajes para que el nodo fuente pueda enviarle la información necesaria.

Debido a que para reprogramar un conjunto de nodos, el descubrimiento de rutas se inicia por el nodo fuente, solo se entrará en detalle en la formación de rutas iniciada por el transmisor o nodo fuente.

Descubrimiento de rutas

Cuando un nodo fuente S necesita enviar información a un grupo de nodos G y no cuenta información en sus tablas para llegar a ese grupo de nodos, envía un mensaje de solicitud de ruta agregando el valor del grupo G al mensaje y envía el mismo por broadcast. Cada nodo en la red que recibe el mensaje lo retransmite a menos que ya lo haya hecho anteriormente. Además el nodo registra en su Tabla de Nodos la dirección de quien envió el mensaje, y el número de secuencia, para evitar retransmitir el mismo paquete.

Después de retransmitir el mensaje si era necesario, y si el nodo pertenece al grupo G, envía un mensaje para unirse a la ruta del nodo que retransmitió el mensaje de solicitud de ruta. El mensaje de unión a la ruta es reenviado por el camino más corto desde donde se originó hasta el nodo fuente S. Cada nodo que retransmite el mensaje de unión de ruta, agrega un registro en la Tabla de Membresía para el fuente S y el

grupo G, indicando que es retransmisor para dicho grupo. El conjunto de rutas con estado de retransmisor entre S y los receptores para G constituyen el árbol multicast de retransmisores como se muestras en la figura 10.

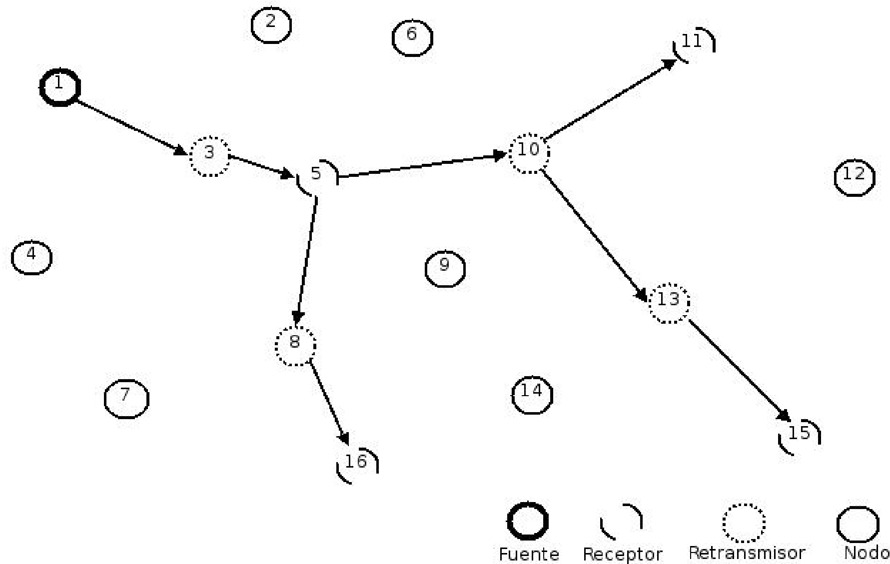

Figura 10: Ejemplo de árbol multicast formado por ADMR

Retransmisión de paquetes multicast

Un nodo en el cual su Tabla de Membresía muestra una entrada para el grupo G y fuente S es ertransmisor de todos los paquetes enviados por S al grupo G. Si el nodo además de ser transmisor, es receptor o miembro del grupo G, procesa el paquete dependiendo de la aplicación que utiliza el protocolo.

Estado Mantenimiento

Durante el estado de operación del protocolo, las condiciones de la red pueden ir variando, es por eso que se requiere un estado de mantenimiento para que los nodos del árbol multicast sigan conectados al nodo fuente S de alguna manera.

El estado de mantenimiento inicia al momento de solicitar la formación de ruta y continúa mientras existan nodos pertenecientes al grupo G que no han recibido toda la información.

Para detectar enlaces rotos, dentro del encabezado del mensaje de datos ADMR va un tiempo estimado en que debe llegar el siguiente paquete, este tiempo es generado por el nodo S al momento de enviar los mensajes de datos. Si el nodo S no tiene paquetes por enviar, pero requiere que se mantenga establecido el enlace entre el árbol multicast, envía mensajes de *Keep-Alive*.

Cuando un nodo C detecta que se ha perdido un cierto número de paquetes de datos o de mensajes *Keep-Alive* entra a un estado de reparación de enlace. Envía un mensaje de reparación a los nodos que se encuentran por debajo de él en el árbol, ésto para evitar que otros nodos inicien también un estado de reparación de enlace. Después de esperar un tiempo, si C escucha un mensaje de reparación de ruta de un nodo D más próximo al fuente S, C cancela su reparación de ruta, ya que D se encuentra más cerca del enlace roto y del nodo fuente S.

Después de esperar un tiempo establecido y no recibir un mensaje de reparación de ruta de un nodo más próximo al fuente S, C decide mandar un mensaje para reconectar; el mensaje se reenvía sólo un número límite de saltos (e.g 3 saltos), de esta forma el mensaje sólo llega a nodos relativamente cercanos a C. Si llega el mensaje de reconectar a un nodo E que no ha recibido un mensaje de reparación, y además es retransmisor del nodo fuente S para el grupo G, se supone que es un nodo que de alguna forma está conectado a S.Entonces E reenvía el mensaje de reconectar de forma unicast hasta llegar al nodo fuente S, si el mensaje llega con éxito, S contesta con un mensaje de

respuesta a la reconección (*Reconnect Reply)* hasta llegar al nodo C.

Organización de la información

Para apoyar la funcionalidad descrita anteriormente, ADMR mantiene tres tablas en cada nodo: la Tabla Nodo, Tabla Membresía, y Tabla de Transmisores o Fuentes.

La Tabla Nodo contiene un registro por cada nodo en la red del cual este nodo ha recibido un mensaje de descubrimiento de rutas. Cada registro incluye los siguientes valores.

- El número de secuencia del encabezado del mensaje ADMR
- La dirección del nodo anterior

La Tabla Membresía contiene una entrada por cada combinación de identificadores de grupos y de fuentes de los cuales éste nodo es receptor o retransmisor. Cada entrada en esta tabla contiene:

- Un campo que indica si este nodo es receptor.
- Un campo que indica que el nodo está conectado al árbol multicast para ese grupo
- Un campo que indica si el nodo es retransmisor
- El tiempo actual entre paquetes para el nodo fuente transmitiendo a este grupo.
- El valor actual del campo *keep-alive*, para mantener la conexión con el árbol.
- La dirección del nodo anterior.

Ls Tabla de Transmisores o Fuente en un nodo contiene una entrada por cada grupo de los cuales este nodo es un fuente activo. La tabla es consultada cuando el

protocolo necesita hacer un descubrimiento. Además cuenta con algunos campos que le permiten conocer la actividad de la red, tal como la movilidad e información sobre cuantas veces se ha entrado a un estado de recuperación de enlace.

III.5 Diseminación de actualizaciones sobre demanda

III.5.1 Protocolo de Formación de Rutas

Cuando se notifica al nodo en la estación base sobre un nuevo programa a diseminar, primero se envía a través de un flood controlado mensajes para actualizar las tablas de enrutamiento para retransmitir el nuevo programa Este mensaje contiene información sobre el nuevo programa y las reglas para elegir los nodos que se van a reprogramar.

Descubrimiento de Rutas

Para reprogramar solamente un cierto grupo de nodos, es importante poder llegar a esos nodos desde la estación base, incluso utilizando nodos que no pertenecen al grupo a reprogramar. Viendo la necesidad anterior y estudiando algunos protocolos de enrutamiento multicast, se puede utilizar un protocolo de este tipo, utilizando el árbol multicast para diseminar el nuevo programa, un ejemplo de formación de rutas se puede ver en la figura 11.

- El nodo S en la estación base envía por *broadcast* un mensaje RREQ
- Cuando un nodo R recibe un mensaje *RREQ* almacena en la Tabla Nodos la dirección del nodo fuente S , la dirección del nodo que reenvió el mensaje RREQ, el número de saltos a partir del último nodo emisor y el peor valor de LQI [1] detectado a través de ese camino.

[1] LQI Link Quality Indicator (Indicador de calidad de enlace)

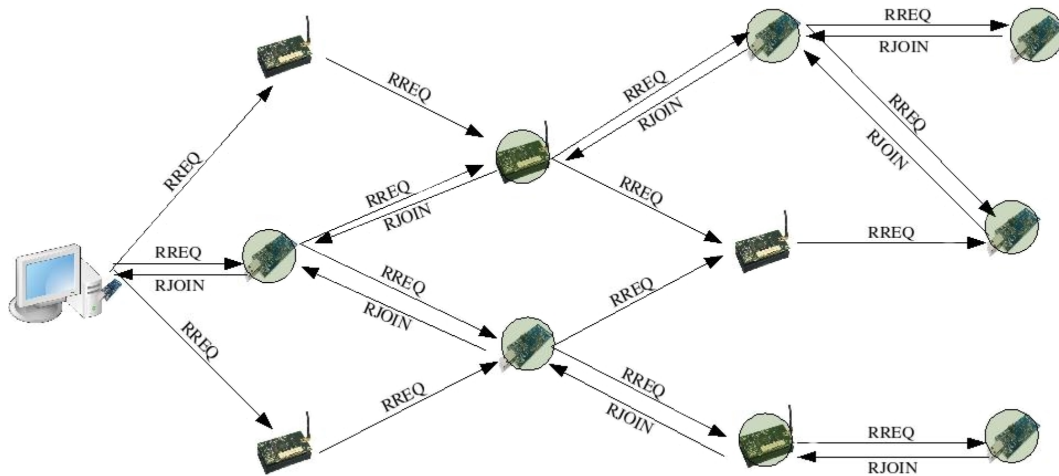

Figura 11: Mensajes para formar ruta

- El nodo R reenvía el mensaje RREQ a sus vecinos, actualizando los valores de *hop count y WLQI*

- Si el nodo R está interesado en el nuevo programa, envía un nuevo mensaje RREQ por broadcast modificando el valor de *hop count*,espera un tiempo considerable (e.g 200 ms) y manda un mensaje *RJOIN* al mejor nodo de la Tabla Nodo (explicado en la sección IV.4.1) y agrega un registro en la Tabla Membresía para saber que es un receptor de la nueva imagen. El tiempo esperado es debido a que no necesariamente el primer *RREQ* recibido proviene de la mejor ruta.

- Cuando un nodo recibe un mensaje RJOIN y no contiene un registro en la Tabla Membresía, se lo agrega para saber que es un retransmisor de ese programa, y reenvía el *RJOIN* al mejor nodo de la Tabla Nodo.

- Si recibe un *RJOIN* y ya contenía un registro en la Tabla Membresía, no retransmite el mensaje y modifica el registro de dicha tabla agregando que es un retransmisor, si no lo tenía.

Elección de la mejor ruta

En el momento que un primer mensaje RREQ llega a un nodo, y si este está interesado en la nueva reprogramación, se espera un tiempo considerable antes de contestarle con un mensaje *RJOIN*, ésto es para dar oportunidad a que lleguen más mensajes. La información de los mensajes es guardada en la Tabla Nodo. Transcurrido el tiempo de espera se evalúan los mensajes *RREQ* para determinar cual es el mejor padre y se le envía el mensaje directamente a él. Debido a que en una red de sensores, todos los mensajes son a través de *broadcast*, cuando un nodo reciba un mensaje de tipo *RJOIN*, evaluará si va dirigido hacia él. La forma de determinar cual es el mejor padre se hace evaluando los valores de *hop count y WLQI* en la Tabla Nodo.

Hop Count es el número de nodos que se van a reprogramar por la ruta definida por el mensaje recibido. *WLQI* es el peor valor del indicador de calidad de enlace por esa ruta. En este caso, de la lista de entradas que se tienen en la tabla nodo, se elije el mejor WLQI para asegurar que es la ruta por la cual hay mejor calidad de enlace.

III.5.2 Retransmisión del nuevo programa

Una vez formada la ruta, el nodo *EB* en la estación base recibe los paquetes del nuevo programa a diseminar y los envía por *broadcast*. Aquellos que tienen en la Tabla Nodo la dirección de *EB* aceptan los paquetes, si es receptor los escribe en memoria, y si es retransmisor, reenvía los paquetes por broadcast cómo se ve en el diagrama de la figura 12.

III.6 Resumen

Para lograr la reprogramación de un grupo selecto de nodos de una red inalámbrica de sensores, es importante determinar las rutas por las cuales pasarán los paquetes de los

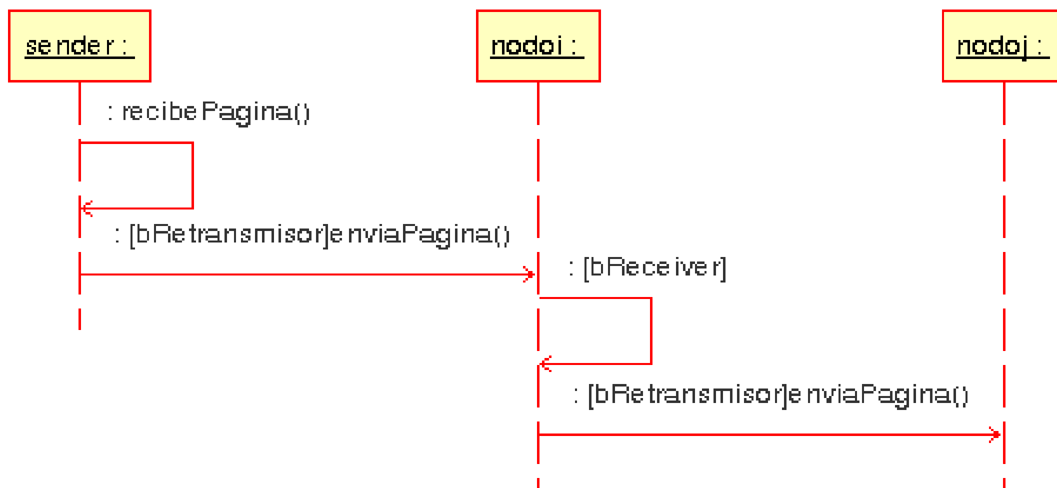

Figura 12: Comportamiento de la retransmisión de un programa

nuevos programas. Es aquí donde un protocolo de enrutamiento multicast sobre demanda se adecúa para lograr el objetivo. El protocolo multicast permite encontrar un mejor camino para disminuir en lo posible que nodos que no se vayan a reprogramar participen en la retransmisión. Sin embargo en ocasiones es necesario que participen para que los nodos que se van a reprogramar logren estar en el árbol multicast. En el siguiente capítulo se muestra la adaptación realizada de un protocolo de enrutamiento multicast al mecanismo de reprogramación Deluge. Además se muestra la implementación realizada para disminuir la información enviada del nuevo programa reprogramando de forma incremental.

Capítulo IV

Implementación

IV.1 Introducción

En el presente capítulo se muestran los detalles de la implementación para lograr el objetivo principal de este trabajo, que es reprogramar de forma incremental un conjunto de nodos sensores. Se tratan con detalle algunos puntos importantes para lograr dicho objetivo, como es identificar los nodos que se van a reprogramar, la descripción de los diferentes mensajes que se utilizan y la manera en que se organiza la información. Se ve con detalle la forma en que se analiza el nuevo programa a enviar para hacerlo de forma incremental, como es que se preparan los bloques de datos y los mensajes para que los nodos copien un bloque de datos del programa anterior a un programa nuevo. Por último se muestra de forma general, cómo se enlazan los componentes desarrollados que forman la implementación.

IV.2 Implementación

La implementación se realizó utilizando como base un mecanismo de reprogramación llamado Deluge (Hui y Culler, 2004) para nodos corriendo TinyOS (Hill *et al.*, 2000) como sistema operativo.

Deluge permite reprogramar a todos los nodos en la red que estén ejecutando un programa que tenga el componente de Deluge activado. Dependiendo de la plataforma se pueden almacenar en EEPROM varios programas, permitiendo enviar un mensaje a

toda la red para que reinicien con un programa en específico. Deluge no reprograma de forma incremental, disemina el nuevo programa a todos los nodos de la red enviando cada página salto por salto.

Deluge está escrito en nesC (Gay *et al.*, 2003), el cual es una extensión del lenguaje de programación C que abarca el modo de estructurar los conceptos y modelo de ejecución de tinyOS.

Para depurar la implementación se utilizó TOSSIM (Levis *et al.*, 2003), un simulador para TinyOS que permite simular una red completa ejecutando tinyOS. Una ventaja de TOSSIM es que utiliza el mismo código que será compilado para los nodos, con algunas pequeñas diferencias.

Para probar su funcionalidad en nodos reales, se utilizaron nodos TmoteSky de Moteiv (2006) que están equipados con un microcontrolador MSP430 de TI. El dispositivo ofrece 10kb de RAM y 48kb de memoria flash interna, además tiene 1Mb de memoria flash externa. En la memoria externa es posible almacenar hasta 6 programas.

IV.3 Elección de grupo a reprogramar

Para realizar pruebas al mecanismo de reprogramación incremental a través de un árbol de enrutamiento multicast, primero se debe identificar el grupo de nodos a reprogramar, para esto en la estación base donde se encuentra el nuevo programa se preparan las reglas que se van a enviar para formar el árbol multicast.

Para identificar un grupo se tiene que:

- Un nodo puede pertenecer a más de un grupo.

- Cada programa en el nodo corresponde a un posible grupo.
- Un nodo puede cambiar de grupo cuando:
  - Se cambia un programa del nodo.
  - Se actualiza un programa del nodo.
  - Se elige el nodo con base en cierto valor de sus lecturas de sensado.

Para identificar cuando un nodo se va a reprogramar se definió el mensaje que se muestra en la figura 13. Este mensaje contiene algunos campos descritos en la definición del protocolo ADMR para enviar una petición de ruta. Además de algunos campos específicos de este trabajo para poder determinar si un nodo se va a reprogramar o no. A continuación se muestran con detalle los campos del mensaje:

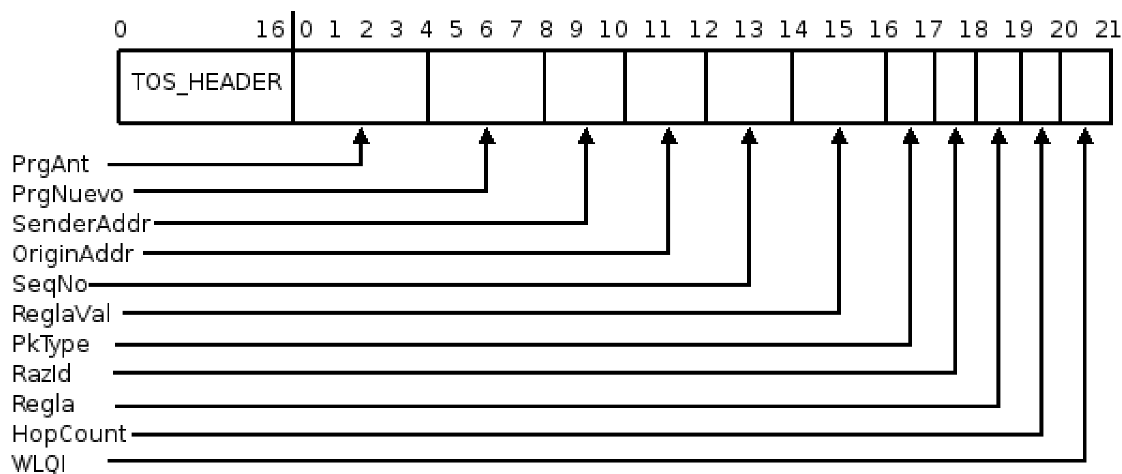

Figura 13: Estructura del mensaje para formar rutas

PrgAnt : 4 bytes que contienen el identificador del programa que debe estar en el nodo que se va a reprogramar.

PrgNuevo : 4 bytes que contienen el identificador del nuevo programa que se va a diseminar.

SenderAddr : 2 bytes que contienen la dirección del último nodo en la ruta que se va a reprogramar.

OriginAddr : 2 bytes que contienen la dirección del nodo que envía el mensaje.

SeqNo : 2 bytes que contienen el número de secuencia del mensaje.

ReglaVal : 2 bytes conteniendo el valor de la lectura del sensor que se evaluará.

PK_Type : 1 byte que contiene el tipo de mensaje (*RREQ* o *RJOIN*).

RazID : 1 byte con el identificador del sensor, o REPR si sólo se va a reprogramar sin verificar las lecturas de los sensores.

Regla :1 byte con el Operador Relacional con el cual se comparará la lectura del sensor con el valor de *ReglaVal*. El operador puede ser:

1. MORETHAN: El valor del campo *ReglaVal* debe ser mayor que el valor de la lectura obtenida del sensor identificado por *RazID*.
2. LESSTHAN: El valor del campo *ReglaVal* debe ser menor que el valor de la lectura obtenida del sensor identificado por *RazID*.
3. EQUAL: El valor del campo *ReglaVal* debe ser igual al valor de la lectura obtenida del sensor identificado por *RazID*.
4. UNEQUAL: El valor del campo *ReglaVal* debe ser diferente al valor de la lectura obtenida del sensor identificado por *RazID*.

Hopcount : 1 byte mostrando el número de nodos que no se reprograman a partir del último nodo que si se reprograma a través de la ruta por la cual ha pasado el mensaje.

WLQI : 1 byte que muestra el peor valor de calidad de enlace detectado por la ruta por donde ha pasado ese mensaje.

Para determinar si un nodo se va a reprogramar se toman en cuenta los siguientes campos: *PrgAnt, RazID, Regla, ReglaVal.*

Si $RazID = REPR$ significa que no va a tomar en cuenta el valor de la lectura de sensor, no toma en cuenta el valor que tienen los campos $Regla$ ni $ReglaVal$ y sólo verifica si en memoria se encuentra el programa identificado con el valor enviado en el campo $PrgAnt$. En la figura 15 se ven los elementos que conforman el descriptor de una imagen, es aquí donde verifica si ya cuenta con algun programa en específico. Si cumple ambas condiciones, el nodo será parte del grupo de nodos a reprogramar y pertenecerá al árbol multicast.

Si $RazID = sensorID$, donde $sensorID$ es el identificador de un sensor en particular, se evalúa el valor de $ReglaVal$ en base al Operador Relacional definido en el campo $Regla$ con el valor que se toma de una lectura del sensor especificado. Si la evaluación es verdadera, el nodo se reprogramará y formará parte del árbol de enrutamiento multicast.

En cualquiera de los dos casos anteriores, si se cumplen las condiciones el nodo se va a reprogramar. Un punto importante a tomar en cuenta es que sólo se reprograma un grupo a la vez, es decir, en ningún momento estarán circulando mensajes de formación de rutas por la red con distintas reglas. El protocolo de ADMR permite que circulen por la red mensajes de distintos grupos y con diferentes nodos fuente, esa funcionalidad se quitó en este trabajo para darle prioridad a la reprogramación.

La reprogramación no es un proceso que esté en ejecución en cada momento, es

más que nada para apoyar en la corrección de errores o agregar nueva funcionalidad a la red. Es por eso que en este trabajo se contempla que sólo un grupo de nodos se va a reprogramar en un momento, y así sólo un árbol multicast se mantendrá en cada momento y ningún paquete de otro programa interferirá con la reprogramación en curso.

IV.3.1 Representación de la información

Al iniciar la ejecución del nodo se realiza una búsqueda de los distintos programas almacenados en EEPROM. El componente que realiza esto es DelugeMetadataC que provee la interfaz DelugeMetadata. En la figura 14 se puede apreciar cómo estan conectados los componentes.

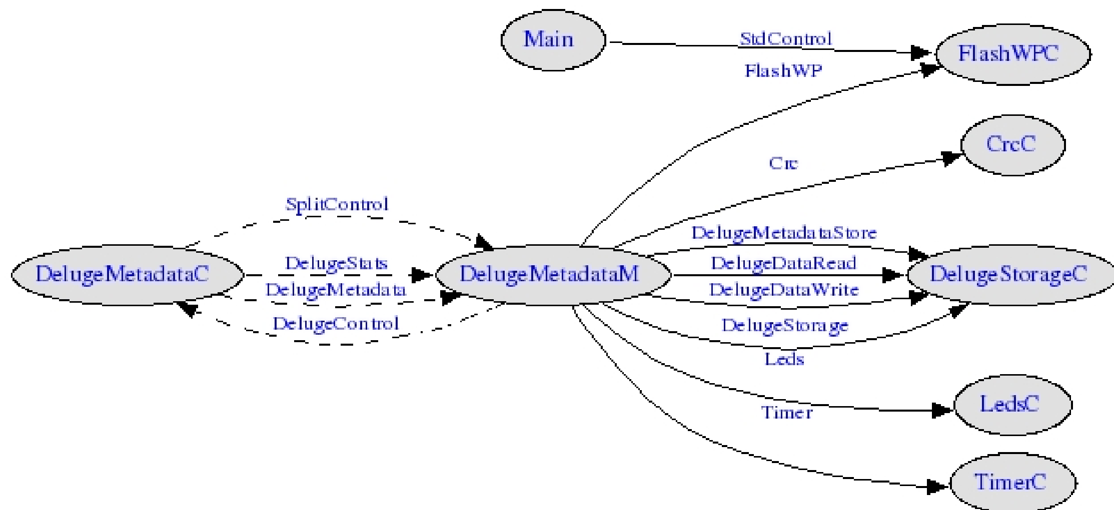

Figura 14: Enlazado de componentes de DelugeMetadata

El componente LedsC se utiliza para propósitos de *debugging* y ver los estados en los que se encuentra el componente, el componente TimerC permite realizar tareas después de transcurrido un tiempo especificado. CrcC verifica el control de redundancia cíclica de una página una vez recibida (ver sección A.1 para mayor detalle). FlashWPC

permite desbloquear un sector protegido, correspondiente al primer sector donde se almacena una imagen llamada *GoldenImage* que será cargado en caso de que ocurra un error al momento de querer reiniciar con una imagen corrupta. DelugeStorageC permite leer y escribir en EEPROM la descripción de los programas. DelugeMetadataM es la implementación del componente y donde se mandan llamar todas las rutinas.

La descripción de los programas está definida por la estructura DelugeImgDesc que se puede apreciar en la figura 15.

DelugeImgDesc
uid: uint32_t vNum: int16_t imgNum: uint8_t numPgs: uint8_t crc: uint16_t numPgsComplete: uint8_t reserved: uint8_t

Figura 15: Estructura de DelugeImgDesc

- **uid**: es el identificador único de la imagen, este valor se genera en la computadora donde se compila el programa.
- **vNum**: es el número de versión de la imágen.
- **imgNum**: es el número de la imagen a la cual hace referencia esta información.
- **numPgs**: es el número de páginas que constituyen la imagen completa.
- **crc**: es el valor del control de redundancia cíclica de los valores de **uid** hasta **numPgs**.
- **numPgsComplete**: es el número de páginas completas que se tiene almacenado en memoria.
- **reserved**: un byte reservado para futuro uso.

IV.4 Formación de Rutas

Una vez que en la estación base se tienen identificadas las reglas para reprogramar un grupo de nodos, se envía un mensaje, tal como está descrito en la figura 13, al broadcast en un flood controlado. Ver figura 16.

Flood Controlado : Es un mecanismo de diseminación de información en el cual se envían los mensajes por broadcast, pero en base a ciertos criterios, como puede ser el número de saltos o el ya haber procesado un mensaje con la misma información, se dejan de reenviar los mensajes.

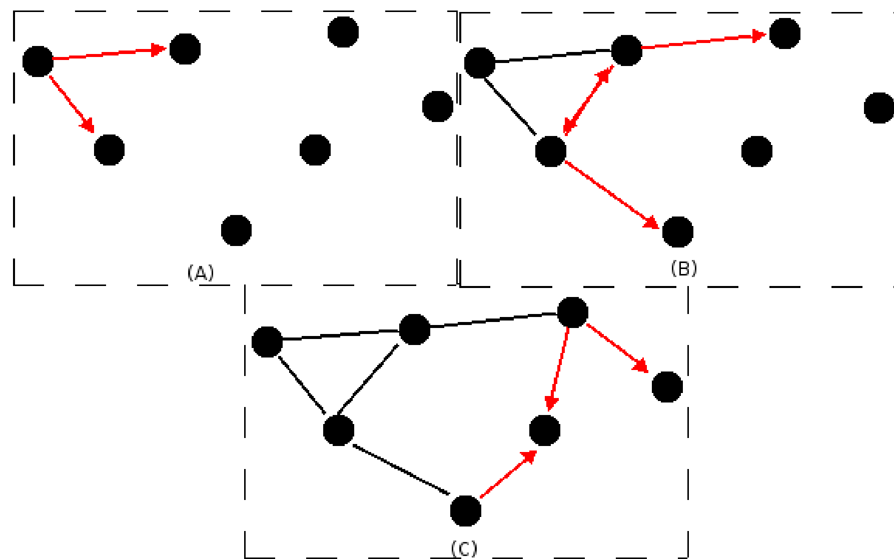

Figura 16: Diseminación de mensajes por flood

Cuando un nodo recibe un mensaje de formación de ruta (i.e $Pk_type = RREQ$) verifica las reglas que vienen en el mensaje como se explicó en la sección IV.3. Si se va a reprogramar se espera un tiempo especificado (e.g. 1 sec), verifica en la tabla Nodo (ver figura 18) el mejor nodo padre y le contesta con un mensaje con $PK_type = RJOIN$. Pero sólo hace eso si no había enviado antes un mensaje con $RJOIN$ y esto se verifica en

la tabla membresía (ver figura 17), si es *Receptor* y/o *Retransmisor* (variables *fReceiver y fForwarder*).

Tabla Membresia
progId: uint_16 senderAddr: uint_16 fReceiver: bool fForwarder: bool

Figura 17: Estructura de la tabla Membresía

Tabla Nodo
senderAddr: uint_16 prev_hop: uint_16 seq_no: uint_16 hopCount: uint_8 wlqi: uint_8

Figura 18: Estructura de la tabla Nodo

Al momento de recibir un mensaje con $PK_type = RREQ$. Es decir un mensaje solicitando una formación de ruta se hace lo siguiente:

- Verifica si ya había visto ese mensaje en base al número de secuencia. Si lo había visto, descarta el mensaje, si no, continúa procesando el mensaje.
- Verifica si se va a reprogramar en base a las reglas que contiene el mensaje como se explicó en la sección IV.3.
- Verifica si el nodo va a retransmitir el mensaje *RREQ*. Si no encuentra en la tabla Nodo una entrada con mejor padre como se vé en la sección IV.4.1, retransmite el mensaje por *broadcast* en un tiempo determinado (e.g. 200ms).
- Para reenviar el mensaje *RREQ* modifica el mensaje, agrega la dirección del nodo en el campo *originAddr*, agrega la dirección del mejor nodo padre en *SenderAddr*

y el número de nodos que se reprograman por esa ruta en $hopCount$.

- Si se va a reprogramar el nodo inicializa el componente de reprogramación y envía un mensaje $RJOIN$ al mejor padre en caso de que no lo haya hecho.

Cuando un nodo recibe un mensaje con $PK_type = RJOIN$. Es decir un mensaje para unirse a la ruta, se hace lo siguiente:

- Verifica si el mensaje era destinado al nodo que lo recibió, el mensaje se envía de forma *unicast*, pero esta verificación es necesaria debido a que todos los nodos escuchan todos los mensajes que transitan por la red, sin importar a quien estén destinados.
- Si no está marcado en la tabla Membresía que el nodo es retransmisor, lo marca como tal.
- Si además de no ser retransmisor, tampoco es receptor, reenvía el mensaje $RJOIN$ al mejor padre localizado en la tabla Nodo.
- Si ya era retransmisor y además receptor, descarta el mensaje ya que esto implica que envió el mensaje $RJOIN$ a su nodo padre con anterioridad.

En la figura 19 se explica gráficamente el funcionamiento del componente de formación de rutas. Cuando un nodo S envía una petición de formación de rutas, mensaje $RREQ$, un nodo i escucha el mensaje, actualiza el mensaje con sus datos y reenvía el mensaje que ahora es escuchado por un nodo j, si el nodo i está interesado en reprogramarse (i.e unirse a la ruta), le contesta un mensaje $RJOIN$ a S. Cuando i escucha un mensaje $RJOIN$ de j retransmite el mensaje a S si y solo si no lo había hecho antes.

Figura 19: Formación de Ruta

IV.4.1 Elección de ruta

Un paso importante en la formación de rutas es la métrica utilizada para elegir la ruta. Jetcheva y Johnson (2001) muestran el protocolo de enrutamiento ADMR para redes *ad-hoc*. Este protocolo está pensado para redes que trabajan bajo IP y no tienen restricciones de energía o ancho de banda. La métrica para elegir la ruta en ADMR para redes *ad-hoc* es el número de saltos. Se escoge la ruta con menor número de saltos. En la implementación de Chen *et al.* (2006) para redes de sensores del protocolo ADMR se evaluan algunas métricas, entre ellas MIN_HOP y MAX_LQI.

MIN_HOP : Esta métrica consiste en ir llevando un contador del numero de saltos que va teniendo cada paquete. Cuando un nodo elige una ruta lo hace eligiendo al nodo que envió el mensaje con número menor de saltos.

MAX_LQI : Esta métrica busca encontrar el camino con el mejor cuello de botella. Sin embargo, no existe en esta métrica una forma de diferenciar entre dos caminos con el mismo cuello de botella pero con diferentes caracteristicas en los enlaces que conforman el camino. Para lograr esto cuando un mensaje va de salto en salto, se va actualizando en el campo *WLQI* el peor valor de calidad del enlace que se halla detectado

Para obtener el mejor cuello de botella cada nodo va retransmitiendo en cada mensaje de formación de rutas el peor valor lqi detectado. De esta forma cada nodo va almacenando una lista de los mensajes que llegan de distintos nodos, junto con los peores valores lqi's detectados. Cuando un nodo decide unirse a una ruta elige el mejor valor lqi almacenado. Así se asegura que la calidad del enlace entre dos nodos cualquiera de la ruta seleccionada, es mejor que al menos un enlace entre dos nodos de cualquier otra ruta definida por un mensaje de formación de rutas.

En la figura 20 se puede ver un ejemplo de los valores lqi de los enlaces en una red. En este caso el nodo 7 recibe mensajes de los nodos 5 y 6. Cada mensaje contiene un peor valor lqi encontrado en el camino. El nodo 5 envía el peor valor lqi detectado de 84, mientras que el nodo 6 envía un valor lqi de 83. Cuando el nodo 7 decide unirse a una ruta lo hace a través del nodo 5, puesto que por esta ruta se encontrará un mejor cuello de botella. El peor valor de calidad de enlace a través del nodo 5 es mejor que al menos un enlace a través del nodo 6.

En la figura 21 se muestran dos ejemplos de diferentes enlaces. en 21-A se elige el camino A, debido a que se utiliza la métrica *MIN_HOP* y sólo se requieren tres saltos, mientras que en el camino B se necesitan 4 saltos. En 21-B se puede detectar que en el camino A puede haber más perdida de paquetes debido a un enlace pobre entre un par de nodos. Utilizando la métrica *MAX_LQI* en este caso se elige el camino B.

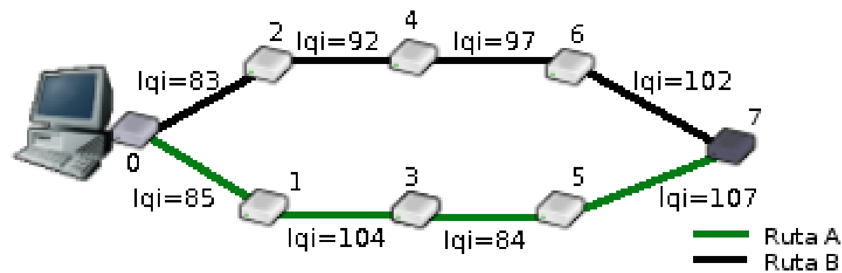

Figura 20: Métrica MAX_LQI

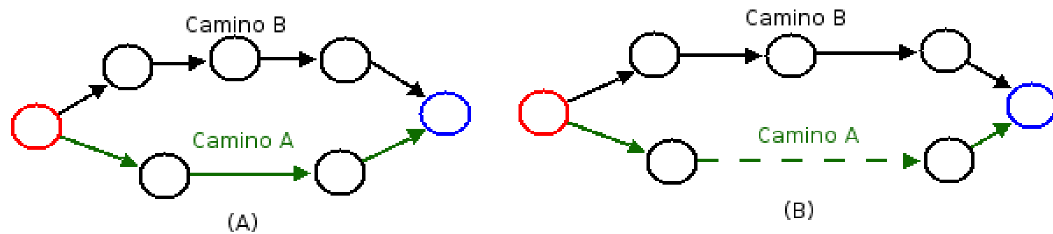

Figura 21: (A) Min_Hop (B) Max_Lqi

En el presente trabajo, para elegir la ruta se hace una combinación de las dos opciones. Sin embargo, debido a que la ruta que se elegirá será utilizada para transmitir grandes cantidades de información (e.g un programa entero) se redefine el número de saltos, como el número de saltos a partir del último nodo que se va a reprogramar a través de esa ruta; entonces se elegirá la ruta con menor valor *Hopcount*. Sin embargo dependiendo de la topología de la red, puede ocurrir que un nodo i reciba un mensaje de j y k teniendo estos últimos $HopCount_j = Hopcount_k$. En este casi se elige el camino con un mejor cuello de botella (MAX_LQI), quedando la elección del camino como sigue:

```
eleccionCamino(){
   mejores=MAXHOP(Tabla Nodo)
   if(count(mejores)>1)
       mejor = MAX_LQI(mejores)
   else mejor = mejores[1]
   return mejor
 }
```

Para calcular el número de saltos se tiene lo siguiente:

$$hopCount_{reg} = \sum_{i=j+1}^{k-1} 1$$

Donde k es el número de nodos que se encuentran por la ruta definida por el registro reg en la tabla Nodo (i.e el número de saltos que recorrió el mensaje hasta llegar al nodo actual) y j es el último nodo que se reprograma a través de esa ruta.

La función $MAXHOP$ regresa el registro de la tabla Nodo con el valor $HopCount$ más bajo, si encuentra más de uno con el valor más bajo, regresa una lista con todos los nodos con el valor más bajo.

IV.4.2 Representación de la información

El presente trabajo se basa en la implementación del mecanismo de reprogramación desarrollado por Hui y Culler (2004) llamado Deluge. Los cambios en la representación de las imagenes en EEPROM fueron pocos, para evitar modificar el gestor de arranque (*bootloader*). Se modificó el tamaño de la página para que los nodos que no se vayan a reprogramar no almacenaran en memoria mucha información, sólo el equivalente a una página.

Hay varios puntos importantes sobre los programas en Deluge como se puede ver en la figura 22.

- Una imagen se divide en páginas

- Cada página consta de 552 bytes. En la implementación de Deluge. cada página constaba de 1104 bytes.
- En la primera página se guardan los CRC's de todas las páginas completas del programa
- Cada página se divide en 24 paquetes de 23 bytes cada uno.
- Cada nodo mantiene un buffer de 552 bytes donde almacena los paquetes de la página que está enviando/recibiendo.

Se redujo el tamaño de las páginas a 552 bytes con la finalidad de que los nodos que no se vayan a reprogramar y por consiguiente no vayan a escribir una página en EEPROM, puedan almacenar una página en un buffer en RAM, para retransmitirlo. Debido a que los nodos cuentan con una memoria RAM pequeña y limitada se decidió disminuir el tamaño de la página para ahorrar un poco de memoria.

IV.5 Preparación de paquetes a enviar

Para lograr una reprogramación incremental, se necesita analizar el programa que se va a enviar y compararlo con el que se encuentra en los nodos.

Se tomó como base el algoritmo *rsync* propuesto por Trigdell (1999) que fué creado para actualizar de forma eficiente archivos binarios, en específico programas.

A diferencia del trabajo propuesto por Trigdell (1999), se tiene contemplado que en una computadora se conocen los diferentes programas que se encuentran en los nodos, así como también se cuenta con el nuevo programa que se quiere enviar.

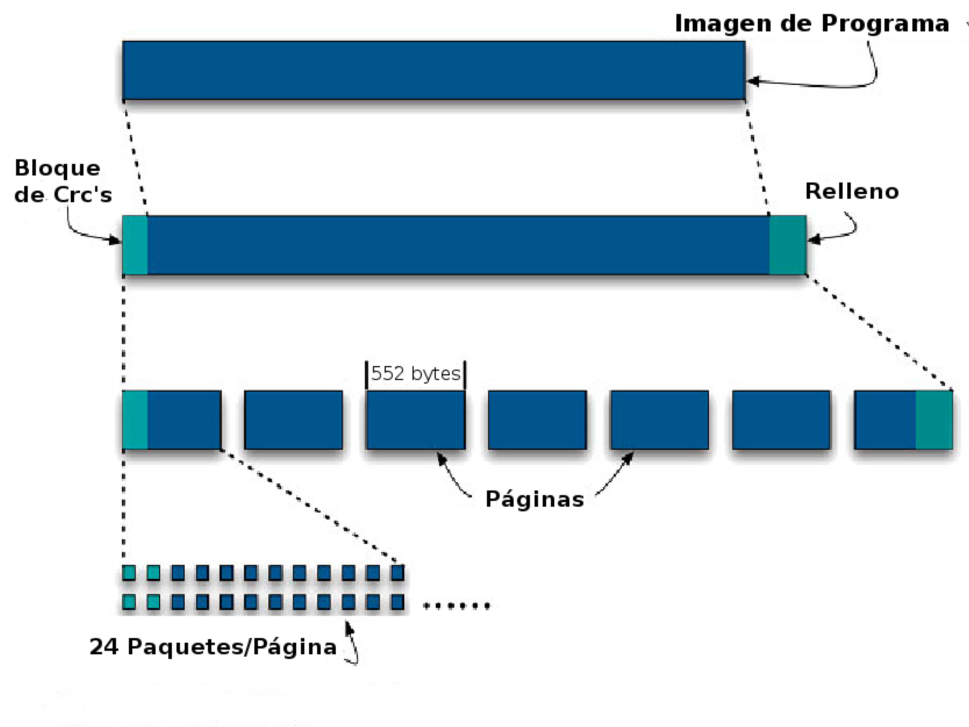

Figura 22: Representación de una imagen

Para la implementación y la generación de las diferencias se utilizó un conjunto de clases llamadas jarsync en lenguaje Java.

Teniendo el programa anterior y el nuevo programa, se realiza una búsqueda de bloques de datos en común de los dos programas de la siguiente manera:

- Se genera una lista de valores *checksum MD4* de 128 bits y *checksum* débil de 32 bits por cada bloque de 23 bytes del programa anterior, aquel que se encuentra en el nodo a reprogramar.
- Se almacena el par generado anteriormente en una tabla hash.
- Se calcula el *checksum* de 32 bits por cada bloque de 23 bytes en el programa nuevo, con un *offste* de 1 byte.

- Se almacenan en una lista ligada llamada *deltas* el conjunto de bloques de datos (clase DataBlock) y de comandos (clase Offsets).

La clase *DataBlock* contiene la siguiente información:

data es un arreglo de bytes donde se almacena un bloque de datos que no se encuentra en el programa anterior.

offset es la dirección donde se escribirá este bloque

La clase *Offsets* Contiene la siguiente información:

blockLength es el tamaño del bloque de datos que se encuentra en el programa anterior y en el nuevo programa.

newOffset es la dirección en el nuevo programa donde se encuentra el bloque de datos.

oldOffset es la dirección en el programa anterior donde se encuentra el bloque de datos.

Los objetos almacenados en la lista ligada *deltas* se tienen que preparar para poder ser enviados a la red de sensores. Como se mencionó en la sección IV.4.2 cada paquete de datos del programa consta de 23 bytes, para eso cada objeto *DataBlock* que se encuentre en la lista *deltas* se divide en bloques de 23 bytes y se agrega a una instancia de una clase llamada Paquete, cada instancia se agrega en una lista y recordando lo que aparece en la sección IV.4.2 cada 24 paquetes conforman una página. La figura 23 muestra la clase Paquete.

Para enviarse a los nodos estos paquetes se almacenan en una estructura del mensaje DelugeData que se puede apreciar en la figura 24. Para manejar más fácil la estructura establecida por Deluge en cuanto al envío de nuevos programas, dividiéndolos en páginas de 24 paquetes, cada uno con 23 bytes.

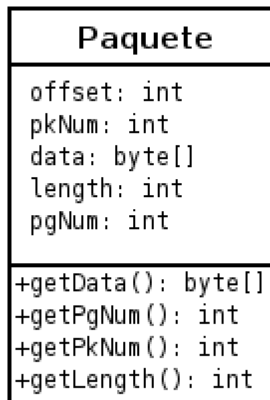

Figura 23: Clase Paquete

El mensaje comprendido con la estructura DelugeData contiene los siguientes elementos:

offset es la dirección donde se escribirá el bloque de datos, en caso de que la información en el campo *data* sea un bloque de datos y no un conjunto de comandos. Este campo se utiliza también para diferenciar entre un bloque de datos o un conjunto de comandos.

length es el tamaño del bloque de datos que se está enviando.

imgNum es el número de la imagen que se está enviando, recordando que Deluge puede almacenar más de una imagen en EEPROM dependiendo del espacio disponible.

pgNum es el número de página que se está enviando.

pkNum es el número de paquete correspondiente a la página *pgNum*. Nos permite detectar paquetes perdidos y no solicitar y/o reenviar una página hasta que se tengan todos los paquetes de la página completa.

data es el bloque de datos que se está enviando, o un conjunto de comandos que reconocerá el nodo que se reprogramará

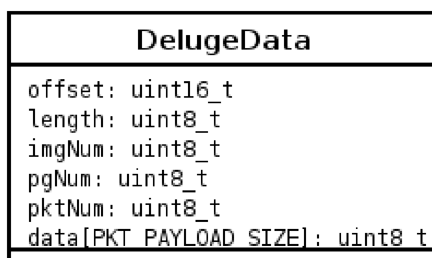

Figura 24: Estructura del mensaje de datos

Los comandos se envían dentro de la misma estructura de la figura 24, para asignar los comandos en la estructura DelugeData se hace como se puede apreciar en la figura 25.

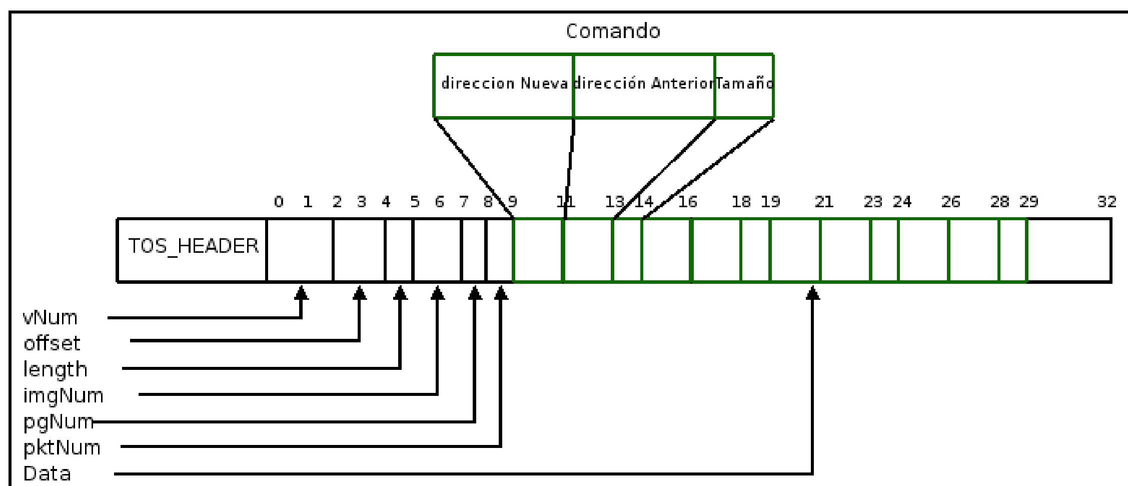

Figura 25: Comandos dentro de un mensaje

- En un paquete de datos se agregan hasta 4 comandos
- Cada comando consta de tres elementos:

dirección Anterior : Es la dirección donde se encuentra un bloque de datos que se va a copiar del programa anterior al nuevo programa, esta función la realiza cada nodo que se va a reprogramar.

dirección Nueva : Es la dirección donde se va a copiar el bloque de datos en la imagen del programa nuevo.

tamaño : Es el tamaño del bloque de datos que se va a copiar de la imagen anterior a la imagen nueva.

- Los comandos se organizan dentro de las páginas, para lograr una administración completa de los comandos, detectando paquetes de comandos perdidos y realizar la petición cuando sea requerido.

- Para diferenciar un bloque de datos de un conjunto de comandos, se utiliza el campo *offset*. Si $offset = 0xFFFF$ significa que el campo *data* contiene comandos. Se utiliza el valor $0xFFFF$ debido a que es imposible que un bloque de datos se escriba en esa posición.

IV.6 Arquitectura

En general, los componentes principales en la implementación realizada en este trabajo se enlazan como se muestra en la figura 26.

El componente llamado *SensorProg* es el encargado de identificar cuando un nodo se va a reprogramar o no, lo determina en base a las reglas que vienen en el mensaje recibido en el componente *Router*, con información obtenida en el componente *DelugeMetadata* que es el encargado de leer los metadatos de los programas almacenados, y con información obtenida al momento de tomar lecturas de algún sensor.

El componente *PMRoute* es el encargado de recibir y enviar los paquetes de formación de rutas, así como también provee una interfaz que será utilizada por otros componentes para determinar si el nodo se va a reprogramar, además busca en la Tabla

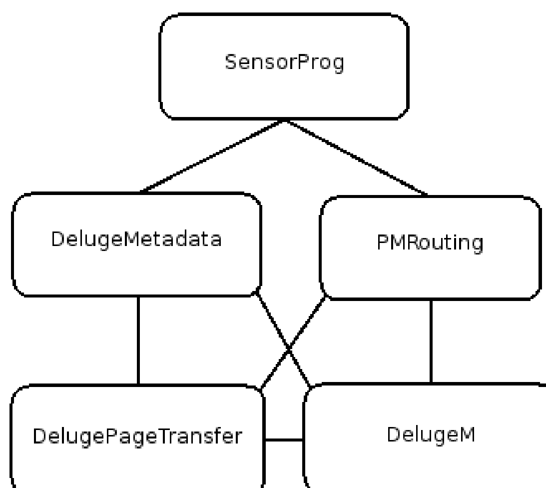

Figura 26: Enlazado de componentes

Nodo la dirección del nodo anterior a quien se le solicitarán paquetes. También determina cuando un nodo es retransmisor y receptor. Cuando se está formando el árbol multicast se almacenan esos datos que después simplemente se consultarán por otros componentes.

El componente *DelugePageTransfer* es el encargado de procesar los paquetes de información, tanto bloques de datos de un nuevo programa como los comandos enviados. También es el responsable de detectar cuando le hacen falta paquetes de la página en curso y hace la solicitud al nodo que aparece en las tablas de enrutamiento.

El componente *DelugeM* es el encargado de anunciar a otros nodos cuando tiene disponible una nueva página para retransmitir. Esto lo hace sólo si es retransmisor consultándolo a través de la interfaz que provee el componente *PMRoute*.

El componente *DelugeMetadata* es el encargado de leer la información de la memoria

externa, donde se encuentran los programas y mantiene un arreglo de metadatos con la información de cada programa, este arreglo es accedido por diferentes componentes.

IV.7 Detalles técnicos de implementación

En el desarrollo de este trabajo se encontraron varios problemas que hicieron que su implementación tomara más tiempo de lo planeado. A continuación se mencionan algunos detalles técnicos que se tuvieron que enfrentar para lograr este trabajo:

- La curva de aprendizaje en el lenguaje nesC para TinyOS es lenta.
- El sistema operativo TinyOS 1.x que fué utilizado para la realización de este trabajo ya no es mantenido por sus autores. Actualmente estan trabajando con una versión 2.x la cual difiere en muchos aspectos y no son compatibles.
- Se tuvo que estudiar a fondo el código de la implementación de Deluge para poder modificarse. Su documentación es escasa.
- En ocasiones fué necesario modificar componentes de bajo nivel, como el gestor de arranque.
- A pesar de que se utilizó el simulador de TinyOS TOSSIM, algunos componentes no se podían simular, y esto hacía más complicado depurar el programa.
- Para depurar el programa en los nodos, en ocasiones se tuvieron que usar los LEDs para poder observar cual era la actividad que estaba realizando el nodo. Esto es una tarea dificil, puesto que sólo se cuentan con tres LEDs.
- Otro problema fué modificar la implementación de Deluge, puesto que está conformada por más de 30 archivos de código, incluyendo archivos de implementación

de componentes, interfaces, encabezados y archivos de configuración. Muchos de estos archivos están ligados entre si.

- En ocasiones los nodos no reaccionaban de la forma esperada. En particular con el componente de radiocomunicación. Lo cual dificultaba el formar topologías específicas para hacer pruebas de funcionamiento.

- Era necesario un conocimiento previo sobre el funcionamiento y característica de algunos componentes de hardware, para entender algunos componentes del código de Deluge. En particular con la memoria externa.

IV.8 Resumen

En este capítulo se muestró la implementación realizada tomando como base un mecanismo de reprogramación existente, se pudo comprobar que los objetivos principales podían adecuarse a un mecanismo de reprogramación ya implementado y bastante probado. Se tuvo que modificar bastante el modelo de reprogramación Deluge para reprogramar de forma incremental y que soportara el seleccionar sólo un grupo de nodos y no la red de sensores entera.

En el siguiente capítulo se describen los experimentos realizados con esta implementación.

Capítulo V

Pruebas y resultados

V.1 Introducción

En el capítulo anterior se presentaron detalles de la implementación que permiten comprender mejor su funcionamiento.
En el presente capítulo se muestran las diferentes pruebas realizadas a la implementación. Está organizado de la siguiente manera: primero se explican a detalle ocho experimentos diseñados para cuantificar el ahorro del uso del componente de radiocomunicación al reprogramar de forma incremental y el gasto de energía, seguido de los resultados obtenidos de los ocho experimentos. Después se muestran dos experimentos y su resultado para mostrar el comportamiento de la formación del árbol multicast por el cual se diseminarán las actualizaciones. Después se muestran diferentes experimentos reprogramando nodos reales tanto incremental como no incremental, midiendo el tiempo en concluir cada experimento y se muestra un análisis de los resultados.

Para las diferentes pruebas se muestra la forma en que se prepararon los nodos y los resultados obtenidos, así como un análisis de los resultados. Todas las pruebas, a excepción de la prueba a la formación de rutas, se realizan reprogramando tanto de forma incremental como diseminando el programa entero utilizando nodos reales.

V.2 Programación Incremental

Para probar la reducción de información de un programa a diseminar reprogramando de forma incremental, se hicieron cuatro casos de prueba, cada caso cuenta con dos pruebas. Cada prueba consiste en enviar un programa, el cual se hace transmitiendo el programa completo y de forma incremental. Para diseminarlo de forma incremental el nodo a reprogramar ya debe contar con otro programa en EEPROM, que es con el cual se generarán las diferencias, y de donde copiará bloques de datos para formar la imagen en el nodo.

Los cuatro casos de prueba son los siguientes:

Caso 1 Cambio de una constante.

Caso 2 Cambio en la implementación.

Caso 3 Cambio de un programa a otro.

Caso 4 Cambio en la configuración

En seguida se dan detalles sobre cada caso y sus respectivas pruebas, mientras que en la sección V.3 se muestran los resultados de su realización.

V.2.1 Cambio de constante

Este consiste en hacer solo una pequeña modificación a un programa almacenado en los nodos que se van a reprogramar. Este pequeño cambio podría consistir en cambiar el valor de un dato que rige el funcionamiento de la aplicación, por ejemplo la periodicidad en que se ejecutan algunas instrucciones.

Prueba 1

Esta prueba consiste en cambiar una constante de una aplicación llamada *Blink* que se encuentra en $TOSROOT/apps/Blink, siendo $TOSROOT el directorio donde se instaló TinyOS-1.x. Teniendo el archivo original *BlinkM.nc* como sigue:

```
command result_t StdControl.start() {
  // Start a repeating timer that fires every 1000ms
  return call Timer.start(TIMER_REPEAT, 1000);
}
event result_t Timer.fired()
{
  call Leds.redToggle();
  return SUCCESS;
}
```

Este programa al iniciar, ejecuta el evento `Timer.fired()` cada 1000ms. El cambio que se realizó da como resultado el siguiente código:

```
command result_t StdControl.start() {
  // Start a repeating timer that fires every 500ms
  return call Timer.start(TIMER_REPEAT, 500);
}
event result_t Timer.fired()
{
  call Leds.redToggle();
  return SUCCESS;
}
```

En la nueva versión se cambió el tiempo en que se ejecutará el evento `Timer.fired()`, que será cada 500ms. Esta aplicación lo único que hace es prender el LED rojo cada vez que se ejecuta el evento `Timer.fired()`.

Prueba 2

Esta prueba consiste en cambiar una constante de una aplicación llamada *Oscilloscope*, que se encuentra en $TOSROOT/apps/Oscilloscope.

Esta aplicación realiza una lectura de un sensor y lo transmite por el puerto serial a la computadora conectada. El archivo original del archivo *Oscilloscope.nc* es como sigue:

```
OscilloscopeM.CommControl -> Comm;
OscilloscopeM.ResetCounterMsg -> //
    Comm.ReceiveMsg[AM_OSCOPERESETMSG];
OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
```

Siendo `AM_OSCOPERESETMSG` y `AM_OSCOPEMSG` constantes con valores `32` y `10` respectivamente. Estas constantes determinan el identificador del mensaje activo que utilizará el programa.

Estas constantes están definidas en el archivo *OscopeMsg.h* de la siguiente forma:

```
enum {
AM_OSCOPEMSG = 10,
AM_OSCOPERESETMSG = 32
};
```

El cambio realizado en esta prueba consiste en modificar el valor del mensaje activo `AM_OSCOPEMSG`. El código resultante es:

```
enum {
AM_OSCOPEMSG = 15,
AM_OSCOPERESETMSG = 32
};
```

V.2.2 Cambio en la implementación

Una actualización típica de un programa podría ser para corregir algún error, para agregar mayor funcionalidad a la aplicación o simplemente porque cambia la lógica del programa.

Este caso de prueba consiste en hacer pequeños cambios a una aplicación, para realizar una actualización hipotética entre dos aplicaciones. Estos cambios se realizan modificando un poco la lógica de la aplicación principal para que haya diferencias entre las dos aplicaciones, para poder compararlas y realizar una reprogramación incremental.

Prueba 3

La primera prueba en este caso consiste en modificar la aplicación original *Oscilloscope*. Como se mencionó anteriormente esta aplicación toma lecturas de un sensor y cuando completa un número de valores envía las lecturas a la computadora a la que está conectado el nodo. Cabe señalar que esta aplicación utiliza el componente llamado `LedsC` que controla los LEDs que contiene el nodo sensor, y cuando ocurren ciertos eventos en el nodo se enciende algún LED. La aplicación en el nodo recibe un mensaje definido por el mensaje activo `AM_OSCOPERESETMSG`, que al recibirlo reinicia un contador a cero.

```
configuration Oscilloscope { }
implementation
{
  components Main, OscilloscopeM
             , TimerC
             ,LedsC
             , DemoSensorC as Sensor
             , UARTComm as Comm,
              DelugeC;

  Main.StdControl -> DelugeC;
  Main.StdControl -> OscilloscopeM;
  Main.StdControl -> TimerC;
  OscilloscopeM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeM.Leds -> LedsC;
  OscilloscopeM.SensorControl -> Sensor;
  OscilloscopeM.ADC -> Sensor;
  OscilloscopeM.CommControl -> Comm;
  OscilloscopeM.ResetCounterMsg -> //
```

```
        Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
}
```

Para evaluar una reprogramación incremental con esta aplicación se hicieron los siguientes cambios

- Se eliminó el componente `LedsC`, que solo servía para identificar cuando una lectura sobrepasaba cierto umbral, pero debido a que los datos se envían a la computadora, ahí se puede identificar si las lecturas sobrepasan el umbral sin necesidad de utilizar el componente.

- Se eliminó el mensaje activo que definía un mensaje enviado de la computadora al nodo, el cual inicializaba una variable a cero para identificar el número de secuencia de la lectura. Se eliminó para poder llevar el control de todas las lecturas en el orden en que se tomaron.

El código del archivo de configuración queda como sigue:

```
configuration Oscilloscope { }
implementation
{
  components Main, OscilloscopeM
             , TimerC
             , DemoSensorC as Sensor
             , UARTComm as Comm,
              DelugeC;

  Main.StdControl -> DelugeC;
  Main.StdControl -> OscilloscopeM;
  Main.StdControl -> TimerC;
  OscilloscopeM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeM.SensorControl -> Sensor;
  OscilloscopeM.ADC -> Sensor;
  OscilloscopeM.CommControl -> Comm;
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
}
```

En el código de la implementación, archivo *OscilloscopeM.nc*, se eliminaron las instrucciones a las llamadas al componente `LedsC`, como se muestra enseguida:

```
if (data > 0x0300)
  call Leds.redOn();
else
  call Leds.redOff();
```

También se eliminó un evento que se ejecutaba al recibir un mensaje de la computadora para reiniciar una variable y la definición de la interfaz del componente:

```
 uses{
  interface ReceiveMsg as ResetCounterMsg;
 }

event TOS_MsgPtr ResetCounterMsg.receive(TOS_MsgPtr m) {
   atomic {
     readingNumber = 0;
   }
   return m;
}
```

Prueba 4

La siguiente prueba consiste en modificar el funcionamiento de la aplicación *SurgeTelos.* Esta aplicación realiza una lectura de uno de sus sensores y la disemina utilizando el protocolo de enrutamiento multihopLQI, siendo la particularidad de esta aplicación que por cada lectura envía el mensaje conteniendo la última lectura que realizó.

La modificación que se hizo en esta aplicación es que se envía un mensaje con lecturas, cada vez que realice 5 lecturas, almacenando en un buffer cada lectura hasta completar las 5 y entonces enviarla. Este cambio requiere modificar un poco la implementación, así como la definición del mensaje, puesto que antes solo se enviaba una lectura y ahora envía un arreglo de las lecturas obtenidas.

Los bloques de código de la aplicación original que se modificaron son los siguientes:

Del archivo *surge.h*

```
typedef struct SurgeMsg {
  uint16_t type;
  uint16_t reading;
  uint16_t parentaddr;
  uint32_t seq_no;
} SurgeMsg;
```

Del archivo *SurgeM.nc* función `SendData`

```
pReading->type = SURGE_TYPE_SENSORREADING;
pReading->parentaddr = call RouteControl.getParent();
pReading->reading = gSensorData;
pReading->seq_no = seqno++;
```

Función `ADC.DataReadt(uint16_t data)`, al momento en que realiza una lectura del sensor

```
atomic {
    if (!gfSendBusy) {
      gfSendBusy = TRUE;
      gSensorData = data;
      post SendData();
    }
 }
```

Como resultado de la modificación, el código resultante quedó como sigue:

Del Archivo *Surge.h*

```
typedef struct SurgeMsg {
  uint16_t type;
  uint16_t reading[5];
  uint16_t parentaddr;
  uint32_t seq_no;
} SurgeMsg;
```

Del archivo *SurgeM.nc* función *SendData*

```
pReading->type = SURGE_TYPE_SENSORREADING;
pReading->parentaddr = call RouteControl.getParent();
for(i =0;i<5;i++)
   pReading->reading[i] = gSensorData[i];
pReading->seq_no = seqno++;
```

Función ADC.DataReady(uint16_t data)

```
if(gfSendBusy)
  return SUCCESS;
atomic gSensorData[numReadings++] = data;
if(numReadings >5){
  gfSendBusy = TRUE;
  post SendData();
}
```

V.2.3 Cambio de un programa a otro

Este caso es uno de los más drásticos. El objetivo es determinar si entre dos aplicaciones diferentes es posible encontrar bloques de código similares, y si al reprogramar de forma incremental se obtiene un beneficio al disminuir el número de paquetes a enviar por la red.

Prueba 5

Esta prueba consiste en cambiar el programa *testDrip* por el programa *SurgeTelosb*. Cabe mencionar que aparte de que las dos aplicaciones tienen diferente funcionalidad, ambas cuentan con el esquema de reprogramación de Deluge modificado en este trabajo, el cual permite reprogramación incremental.

Prueba 6

En esta prueba se cambia, como en la prueba anterior, el programa *testDrip* por el programa *SurgeTelosb*. Sin embargo, ahora ninguna aplicación cuenta con el componente que permite reprogramar de forma incremental. Hay que tomar en cuenta que el diseminar un nuevo programa, ya sea incremental o no, no significa que inmediatamente el

nodo va a cambiar de programa. Para ésto, tiene que recibir otro mensaje enviado desde la computadora, y diseminado por broadcast, para reiniciar con el programa deseado.

V.2.4 Cambio en la configuración

Una aplicación hecha para TinyOS consiste en al menos un archivo de configuración y varios archivos de implementación. En el archivo de configuración se enlazan los componentes utilizados en la aplicación. Los componentes se enlazan de tal manera que un componente utilice una *interfaz* que otro componente provee.

Prueba 7

Esta prueba consiste en cambiar un componente en una aplicación. Se eligió la aplicación `CntToLeds`, que es una aplicación muy sencilla cuya única función es desplegar el valor de un contador en sus tres LEDs. Se eligió la aplicación, debido a que utiliza un componente llamado `IntToLeds` y un componente llamado `Counter`. El componente `Counter` utiliza un componente que provee la *Interfaz* llamada `IntOutput`. El componente `IntToLeds` provee dicha Interfaz.

El código del archivo de configuración es como sigue:

```
configuration CntToLeds {
}
implementation {
  components Main, Counter, IntToLeds, TimerC,DelugeC;
  Main.StdControl -> DelugeC;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  Counter.IntOutput -> IntToLeds.IntOutput;
}
```

Para realizar esta prueba se utiliza un componente llamado `IntToRfm`, el cual provee

la misma *Interfaz* que utiliza el componente `Counter`, así que solo es necesario realizar los cambios en el archivo de configuración para que quede el código como sigue:

```
configuration CntToLeds {
}
implementation {
  components Main, Counter, IntToRfm, TimerC,DelugeC;
  Main.StdControl -> DelugeC;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  Counter.IntOutput -> IntToRfm.IntOutput;
}
```

Como se puede ver, este es el único cambio que hubo: en donde se encontraba el componente `IntToLeds` se reemplazó por `IntToRfm`

Prueba 8

Esta prueba se realizó con la aplicación *Oscilloscope*. Esta aplicación utiliza un componente llamado `LedsC` para poder visualizar algunos eventos que ocurren en el nodo, al recibir un mensaje o al momento de tomar una lectura de un sensor, ver si el valor sobrepasa un umbral ya establecido.

El código del archivo de configuración de esta aplicación está dado como sigue:

```
configuration Oscilloscope { }
implementation
{
  components Main, OscilloscopeM
             , TimerC
             ,LedsC as LedsC
             , DemoSensorC as Sensor
             , UARTComm as Comm,
              DelugeC;
  Main.StdControl -> DelugeC;
  Main.StdControl -> OscilloscopeM;
  Main.StdControl -> TimerC;
```

```
  OscilloscopeM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeM.Leds -> LedsC;
  OscilloscopeM.SensorControl -> Sensor;
  OscilloscopeM.ADC -> Sensor;
  OscilloscopeM.CommControl -> Comm;
  OscilloscopeM.ResetCounterMsg -> Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
}
```

El cambio que se hizo a esta aplicación es cambiar el componente `LedsC` por el componente `NoLeds`. El componente `NoLeds` provee los mismos métodos y las mismas Interfaces que el componente `LedsC`. Sin embargo no activa los leds de los sensores.

El código de la aplicación modificada queda como sigue:

```
configuration Oscilloscope { }
implementation
{
  components Main, OscilloscopeM
            , TimerC
            ,NoLeds as LedsC
            , DemoSensorC as Sensor
            , UARTComm as Comm,
             DelugeC;
  Main.StdControl -> DelugeC;
  Main.StdControl -> OscilloscopeM;
  Main.StdControl -> TimerC;
  OscilloscopeM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeM.Leds -> LedsC;
  OscilloscopeM.SensorControl -> Sensor;
  OscilloscopeM.ADC -> Sensor;
  OscilloscopeM.CommControl -> Comm;
  OscilloscopeM.ResetCounterMsg -> Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
}
```

V.3 Resultados de la Programación Incremental

A continuación se muestran los resultados tras la aplicación de las pruebas para cada uno de los cuatro casos, tal como se describieron en la sección V.2.

En los primeros dos casos, se modifica la implementación. En el caso uno se cambian algunos valores de unas constantes, que en cierta manera repercuten en el funcionamiento general de la aplicación. En el segundo caso se modifica el funcionamiento de algunos componentes.

Recordar que en la prueba uno se cambió una constante que indicaba la periodicidad en que ocurre cierto evento. En la segunda prueba se cambió la constante que indica el identificador del mensaje activo con el que trabaja la aplicación.

En la tercera prueba, se elimina un componente llamado `LedsC` y se elimina un mensaje activo y el componente que lo recibe. En la cuarta prueba se cambia un poco mas el funcionamiento de la aplicación. Antes se obtenía una lectura de un sensor y se transmitía de inmediato. Después de modificar, la aplicación se espera a obtener 5 lecturas antes de enviar el mensaje.

Los resultados obtenidos al reprogramar de forma incremental con las pruebas mencionadas se puede apreciar en la gráfica de la figura 27. Se puede apreciar en las primeras dos pruebas que el número de páginas enviadas para reprogramar son mucho menor en comparación con enviar la imagen completa, debido a que el cambio en la aplicación fue muy poco.

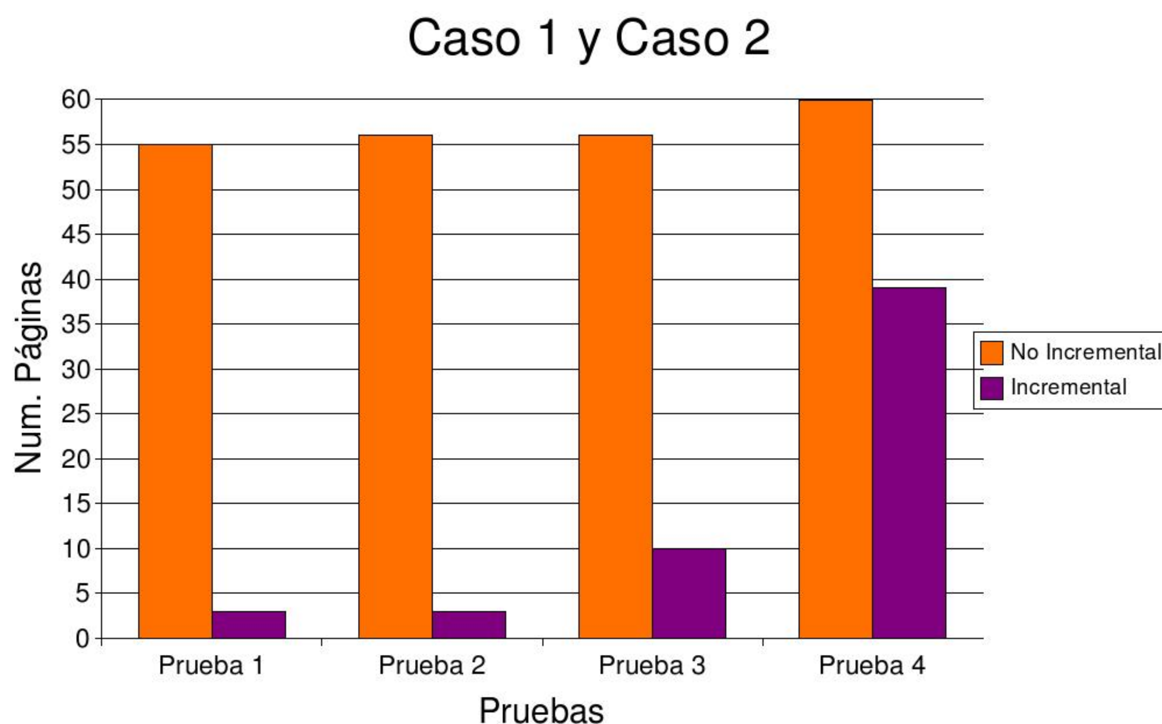

Figura 27: Pruebas de caso 1 y caso 2

En las pruebas tres y cuatro se ve que se envían mucho más paquetes que en las dos pruebas anteriores. En la prueba tres no se modificó la implementación, sólo se eliminaron componentes, por lo tanto era considerable la cantidad de información compatible entre las dos versiones de los programas.

En la prueba cuatro hubo mayores cambios en la implementación. No se eliminaron componentes, sin embargo se aumentó el código en algunas funciones para lograr la funcionalidad deseada. Debido a que al realizar una comparación a nivel de *bytes*, como lo hace nuestra implementación, no se toman en cuenta ciertos elementos que se encuentran en la tabla de símbolos y el direccionamiento de las funciones, se realizan desplazamientos en el direccionamiento de las funciones y variables, las cuales no son detectadas al comparar a nivel de *bytes*. Es por esto que el número de páginas

y por consiguiente de paquetes se incrementa en comparación con las pruebas anteriores.

En el caso tres y cuatro se cambiaron los componentes de las aplicaciones. En las pruebas cinco y seis se cambiaron todos los componentes, es decir, se cambia completamente la aplicación, cambia incluso el nombre de ellas. Son aplicaciones totalmente diferentes, sin embargo en la prueba cinco, ambas aplicaciones cuentan con el componente de reprogramación, eso hace que haya más bloques de datos en común entre los dos programas, a diferencia de la prueba seis, donde cambian los mismos programas que en la prueba cinco, pero sin el componente de reprogramación. Esto se puede ver claramente en la gráfica de la figura 28.

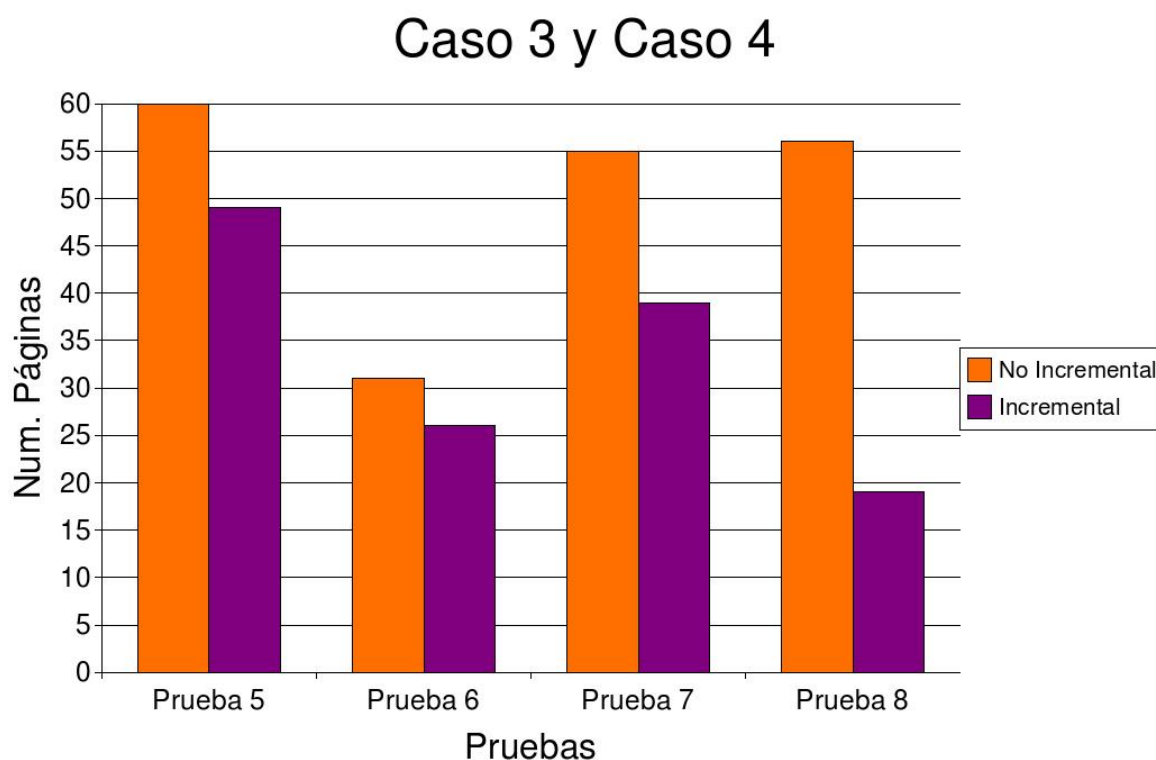

Figura 28: Pruebas de caso 3 y caso 4

Un punto importante que se puede observar en la prueba cinco y seis, es que a pesar

Tabla III: Número de paquetes Incremental y No Incremental

Prueba	**No Incremental**	**Incremental**
1	1320 paquetes	72 paquetes
2	1344 paquetes	72 paquetes
3	1344 paquetes	240 paquetes
4	1440 paquetes	936 paquetes
5	1440 paquetes	1176 paquetes
6	744 paquetes	624 paquetes
7	1320 paquetes	936 paquetes
8	1344 paquetes	456 paquetes

de que se utilizan para ambas pruebas las mismas aplicaciones, el hecho de agregar el componente de reprogramación, incrementa bastante el número de bytes. Es el precio que se tiene que pagar para permitir que los nodos se puedan reprogramar a través del aire.

En la prueba siete y ocho, se cambia sólo un componente, sin embargo se puede apreciar en la gráfica de la figura 28 que en la prueba siete se diseminaron mucho más paquetes que en la prueba ocho. Esto es debido a que en la prueba siete se cambia un componente llamado `IntToLeds` por uno llamado `IntToRfm`, y el segundo componente a pesar de que cuenta con la misma *interfaz* que el primero requiere de otros componentes para controlar el radio y poder enviar los mensajes. Mientras que en la prueba ocho que se cambia el componente `LedsC` por `NoLeds`, ambos con la misma interfaz, el último no utiliza internamente ningún componente, ni siquiera el que controla los LEDs.

En la tabla III se puede apreciar el número de paquetes que se envía en cada prueba, en la primera columna los que se envían al programar de forma convencional, sin el componente de programación incremental, y en la segunda columna los que se envían al reprogramar de forma incremental. Se puede apreciar que en todas las pruebas realizadas se disminuyó el número de paquetes a enviar. Aún cuando se cambia una aplicación

Tabla IV: Energía utilizada en TmoteSky

Operación	**Cantidad**	**Unidad**
Leer 1 byte de EEPROM	0.13	μJ
Escribir 1 byte en EEPROM	0.243	μJ
Leer EEPROM (instrucción)	0.52	μJ
Escribir EEPROM (instrucción)	0.972	μJ
Transmitir 1 byte por radio	1.891	μJ
Recibir 1 byte por radio	1.67	μJ

por otra es posible encontrar bloques de datos en común entre las dos aplicaciones, ésto se debe principalmente a que todas las aplicaciones comparten componentes del sistema operativo utilizado, en este caso TinyOS, además todos los programas utilizados en estos experimentos, a excepción de la prueba 6, cuentan con el componente de reprogramación implementado en este trabajo, lo cual hace que haya más información en común entre todas las versiones de las aplicaciones.

En la siguiente sección se cuantifica el gasto de energía por cada nodo en los ocho experimentos aquí mostrados.

V.3.1 Energía al reprogramar

Una característica importante que se tiene que tomar en cuenta al desarrollar aplicaciones en este tipo de dispositivos es la cantidad de energía que se consume. La utilización de la unidad de radiocomunicación es uno de los procesos mas costosos, seguidos por la escritura/lectura en la memoria externa. En la tabla IV se ve la corriente utilizada por operación.

La energía utilizada se calculó con la fórmula $E = P \cdot T$, donde T es el tiempo y P es la potencia calculada como $P = V \cdot I$ donde V es el voltaje. Debido a que los nodos sensores se alimentan con dos baterias de 1.5V cada una, entonces $V = 3.0$. I es la

Tabla V: Parámetros para calcular energía

Descripción	**Cantidad**	**Unidad**
Voltaje	3.0	V
Transmision (corriente)	19.7	mA
Transmitir (velocidad)	250	kbps
Recibir (corriente)	17.4	mA
Recibir (velocidad)	250	kbps
Escribir (corriente)	15.0	mA
Escribir 1 byte (tiempo)	5.405	μs
Escribir instrucción (tiempo)	21.622	μs
Leer (corriente)	8.0	mA
Leer 1 byte (tiempo)	5.405	μs
Leer instrucción (tiempo)	21.622	μs

corriente utilizada en cada operación en mA. Los parámetros para obtener la energía se pueden ver en la tabla V y se obtuvieron en el archivo de descripción del chip, tanto de radio (Chipcon, 2004) como de la memoria externa (STMicroeletronics, 2004).

La estimación del consumo de energía se hizo con base en la tabla IV para las ocho pruebas que se realizaron en la sección V.2 de este capítulo. Esta es sólo una aproximación debido a que no se toman en cuenta los mensajes de anuncios ni las retransmisiones de paquetes de datos debido a su pérdida. Esto tanto reprogramando de forma incremental, como diseminando el programa entero.

Como se vé en la gráfica de la figura 29, programando de forma incremental se reduce el consumo de energía comparado con enviar el programa entero. Este consumo de energía corresponde a un nodo que es receptor y al mismo tiempo es retransmisor.

Si se compara la gráfica de la figura 29 con la gráfica 30 se puede ver que el consumo de energía es proporcional al tamaño de la imagen a diseminar.

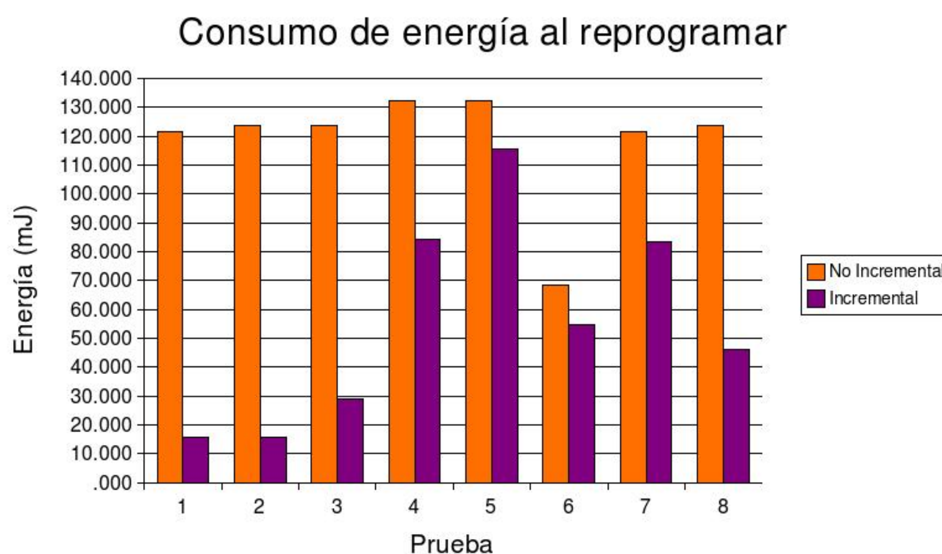

Figura 29: Estimación de gasto de energía

Ahora comparando la energía gastada reprogramando de forma incremental, pero comparando a nodos que son sólo retransmisores con nodos que son retransmisores y se van a reprogramar se obtiene la gráfica 31. Se puede ver claramente que los nodos intermedios, al no escribir el nuevo programa en EEPROM consumen menos energía que aquellos que si se van a reprogramar.

En la gráfica 32 se muestra el consumo de energía de nodos que no se van a reprogramar, pero son retransmisores, con la de aquellos que si se van a reprogramar y además también son retransmisores. De la misma manera que con la gráfica 31 el consumo de energía de los nodos que no se van a reprogramar pero intervienen en la reprogramación es menor.

En la tabla VI se muestra el porcentaje de ahorro de energía obtenido mediante las pruebas de la sección V.3.1 al reprogramar de forma incremental, tomando como el 100% reprogramar diseminando el programa entero. Este ahorro de energía es por

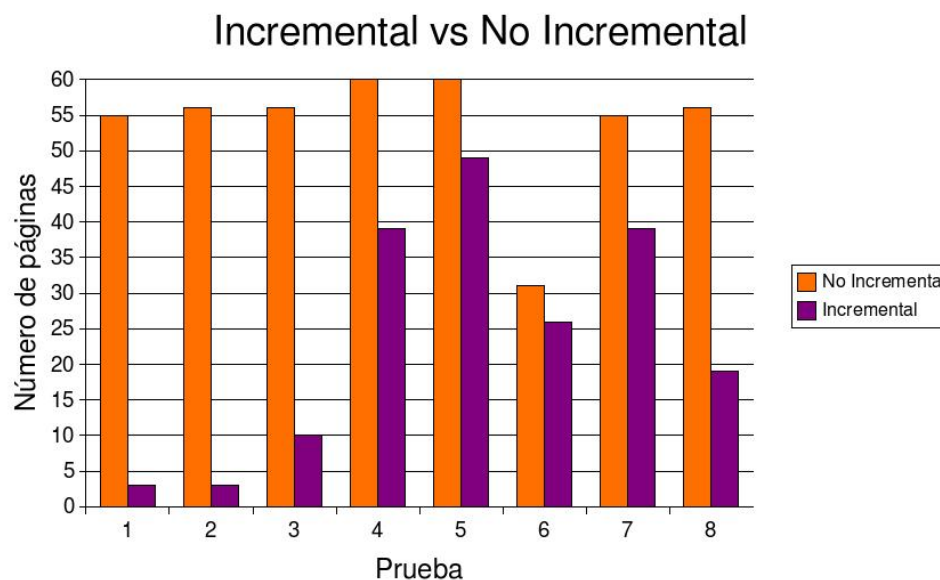

Figura 30: Tamaño de datos Incremental vs. No Incremental

nodo que además de reprogramarse es retransmisor.

Como se puede observar, en todas las pruebas hubo un ahorro de energía, incluso en aquellas donde se intercambia un programa por otro totalmente diferente, como son las pruebas 5 y 6 con un ahorro de energía de 12.66 % y 20.14 % respectivamente . En las primeras dos pruebas donde sólo se hace un cambio de una constante el ahorro de energía sobrepasa el 87 %.

Tabla VI: Ahorro de energía

Prueba	**Ahorro de Energía**
1	87.03 %
2	87.13 %
3	76.42 %
4	36.24 %
5	12.66 %
6	20.14 %
7	31.1 %
8	62.67 %
Promedio	**51.67 %**

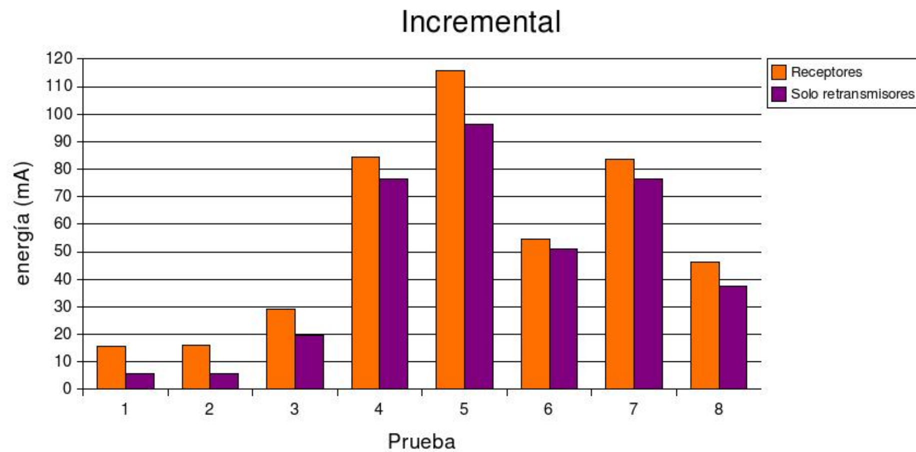

Figura 31: Estimación de gasto de energía de forma Incremental

Tabla VII: Gasto de energía en nodos retransmirores

Prueba	**Gasto de Energía**
1	37.46 %
2	37.06 %
3	67.44 %
4	90.78 %
5	83.27 %
6	93.52 %
7	91.65%
8	80.94%
Promedio	**72.76%**

En la tabla VII se muestra el porcentaje de gasto de energía de los nodos que fungen como retransmisores. Este ahorro de energía es tomando en cuenta el 100% a los nodos que aparte de ser retransmisores son receptores, es decir, se van a reprogramar, y 0% obviamente gastan los nodos que no se van a reprogramar ni intervienen como retransmisores.

Este gasto de energía es contemplando solamente los mensajes de datos y no los mensajes de formación de rutas y tampoco mensajes anunciando un nuevo programa.

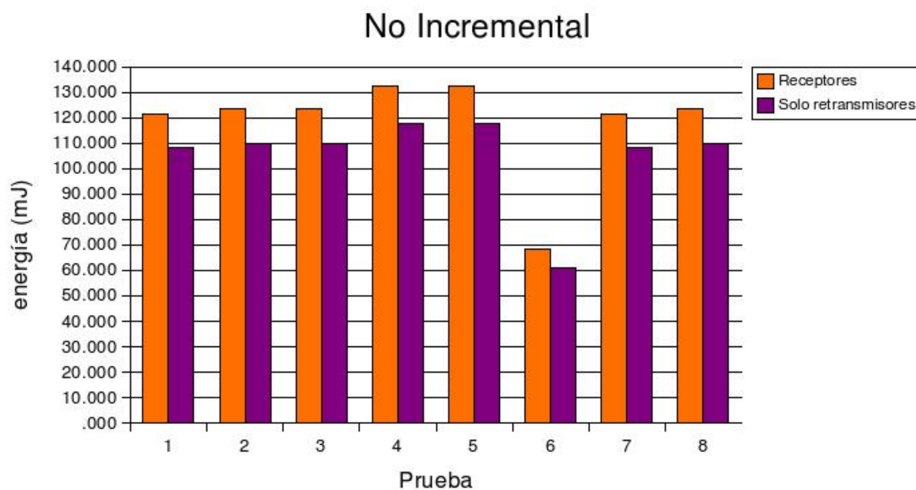

Figura 32: Estimación de gasto de energía de forma No Incremental

Como era de esperarse, los nodos que intervienen en la reprogramación sólo como retransmisores gastan energía al utilzar el componente de radiocomunicación, pero en menor cantidad de los nodos que se van a reprogramar, que escriben en la memoria externa. Es por esta razón que es importante hacer una buena selección de rutas por donde van a pasar los paquetes de los nuevos programas, para evitar en lo posible que nodos que no se vayan a reprogramar actúen como retransmisores.

V.4 Formación de Rutas

A continuación se muestran dos experimentos realizados para ver el comportamiento del mecanismo de reprogramación implementado en este trabajo al seleccionar sólo un grupo de nodos para reprogramar y no la red entera. Estos experimentos se realizaron utilizando TOSSIM, que es un simulador de TinyOS para poder observar las tablas de enrutamiento de cada nodo y ver las rutas seleccionadas por cada uno de ellos.

En la figura 33 se muestran las rutas para reprogramar dos nodos, el número 10 y

el número 11. En este caso el protocolo implementado en este trabajo seleccionó dos rutas para llegar a los nodos, las cuales son 0,1,4,7,10 y 0,3,6,9,11. Como se puede ver claramente sería más eficiente seleccionar las rutas 0,2,5,8,{10,11}. Pero debido a que el protocolo desconoce la topología de la red, no se seleccionó esa ruta. Debido a la incapacidad de simular la calidad de enlace con TOSSIM, todos los nodos contaban con la misma calidad del enlace. Eligiendo así sólo las rutas que satisfacían la condición de estar más próximo a un nodo que fuese a reprogramarse.

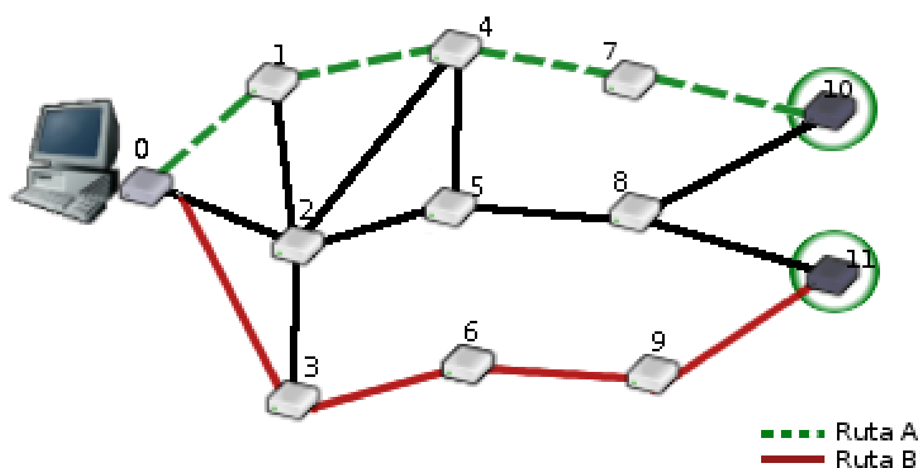

Figura 33: Programando dos nodos

Tomando la misma topología que en la prueba anterior, pero ahora programando los nodos 5, 10 y 11 se obtiene un árbol como se puede ver en la figura 34. En este caso se elige la ruta A para reprogramar los nodos 5,10,11. Esto es debido a que cuando llega un mensaje RREQ a 10 y a 11 del nodo 8, ambos eligen la ruta definida por el nodo 8, ya que hay un nodo más próximo que se va a reprogramar, en este caso el nodo 5.

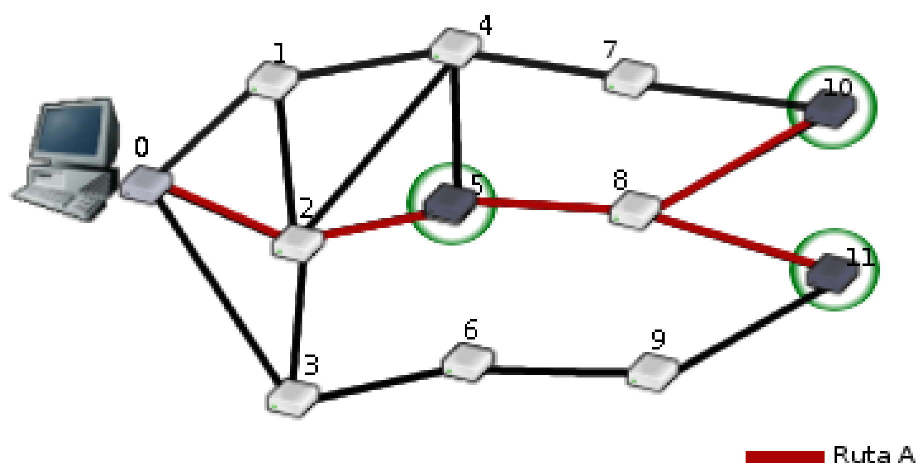

Figura 34: Programando tres nodos

V.5 Pruebas de Reprogramación

En esta sección se describen las pruebas que se realizaron al mecanismo de reprogramación y se tomaron tiempos programando un cierto número de nodos, variando los saltos necesarios para reprogramar a los nodos o variando el número de nodos a reprogramar manteniendo el número de saltos constante.

Los tiempos obtenidos son solo una aproximación, debido a que se hicieron a través de observaciones utilizando un cronómetro controlado manualmente para tomar el tiempo. Para determinar cuando se termina la reprogramación, se programaron los nodos para que encendieran un LED al finalizar.

Cabe señalar que estas pruebas se hicieron en un espacio cerrado (i.e. en un laboratorio de cómputo) con nodos reales. Para evitar la formación de otra topología y no la deseada, se restringieron los nodos para que sólo pudieran escuchar un nodo antes y un nodo después de él, como se puede ver en la figura 35.

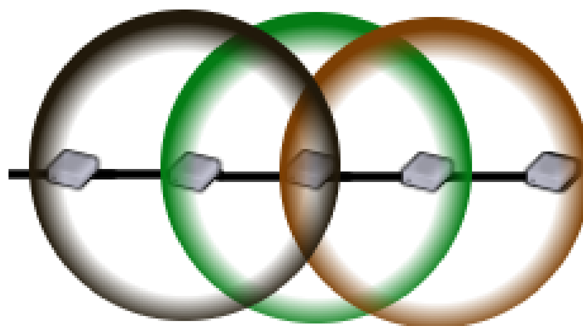

Figura 35: Comunicación de nodos

V.5.1 Reprogramando un nodo a varios saltos

El número de nodos a reprogramar se mantuvo constante para la primera prueba, se fijó en uno el número de nodos a reprogramar y el número de saltos variaba de 1, 2, 3, 4 y 5 saltos.

En la figura 36 se ve la topología de los nodos en cada prueba, en la figura (a) se reprograma un nodo a un sólo salto, no tomando en cuenta el pasar de la computadora al primer nodo como un salto. En la figura (b) se reprograma un nodo a dos saltos. En la figura (c) un nodo se reprograma a tres saltos. En la figura (d) un nodo se reprograma a cuatro saltos. En la figura (e) se reprograma un nodo a cinco saltos. Los nodos color gris claro son retransmisores mientras que los nodos gris oscuro se van a reprogramar.

El programa que se envió cuenta con 60 páginas, y es el mismo que se envió en la prueba 4 en la sección V.2.2. Al enviarlo de forma incremental el número de páginas se redujo a 40.

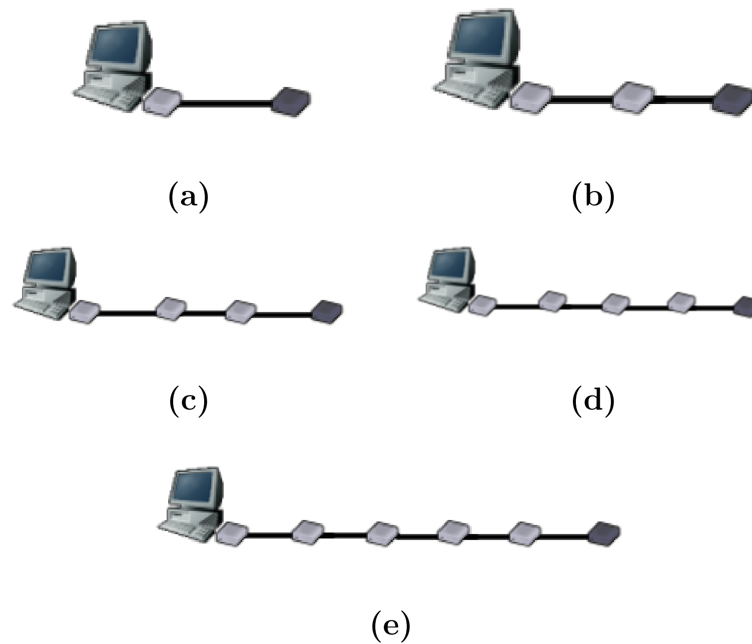

Figura 36: Programando un nodo

Cabe recordar que el tamaño de cada página es de 24 paquetes de 23 bytes cada uno, teniendo un total de 552 bytes por página.

En esta prueba se tomó el tiempo en que se termina de diseminar el programa a un nodo, usando como intermediarios a varios nodos, de uno hasta 5. En la gráfica de la figura 37 se ve la forma en que a más saltos obviamente el tiempo se va incrementando, sin embargo de forma incremental a cinco saltos el tiempo sigue siendo menor que reprogramando de forma no incremental a un solo salto.

La diferencia en el tiempo de reprogramación en los primeros 3 saltos es mínima, mientras que cuando se agrega un nodo más, el tiempo en concluir la reprogramación se incrementa. Esto es debido a que a pesar de que los nodos son forzados a nivel de aplicación para atender mensajes de un nodo adelante de ellos y un nodo atras, existe interferencia a nivel de la capa de enlace con otros nodos transfiriendo información.

Tabla VIII: Tiempo en reprogramar variando los saltos

Saltos	No Incremental	Incremental
1	02:14.59	01:29.43
2	02:21.37	01:31.12
3	02:24.43	01:32.36
4	02:42.14	01:53.99
5	02:59.31	02:05.27

Cuando se tienen solamente 3 nodos, en cualquier tiempo sólo se está llevando a cabo una transferencia de una página, mientras que con 4 o más nodos, puede haber más de un nodo transfiriendo una página causando ruido en la comunicación de otro par de nodos transfiriendo otra página.

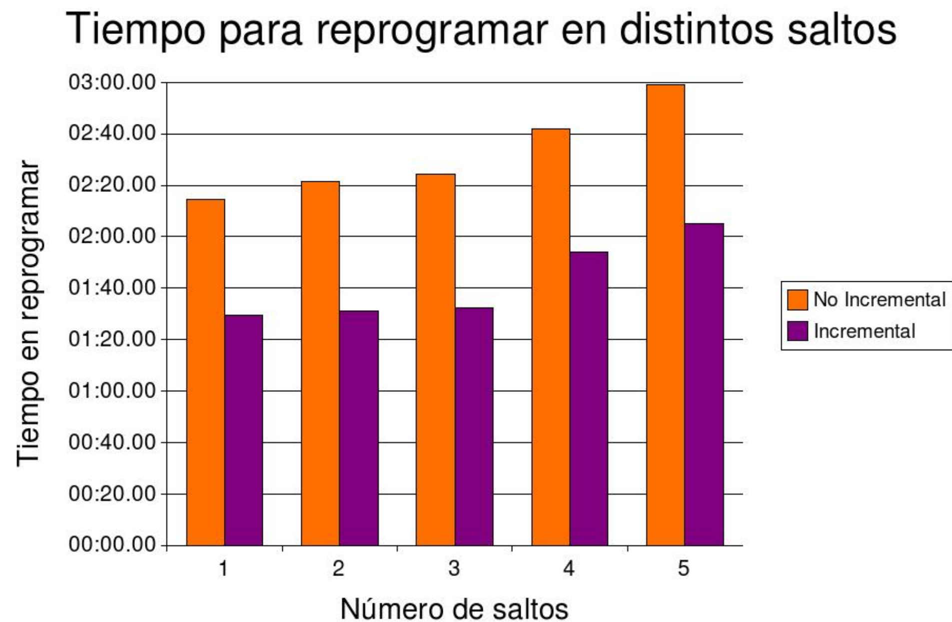

Figura 37: Programando un nodo a varios saltos

En la tabla VIII se muestran los tiempos logrados al reprogramar un nodo en diferentes saltos.

V.5.2 Reprogramando varios nodos

Para esta prueba se eligió ir reprogramando de uno hasta cinco nodos, pero siempre un nodo se encuentra a cinco saltos. En la figura 38 se puede ver la topología utilizada en cada prueba, teniendo a los nodos en color gris bajo como retransmisores y los nodos gris oscuro como nodos que se van a reprogramar.

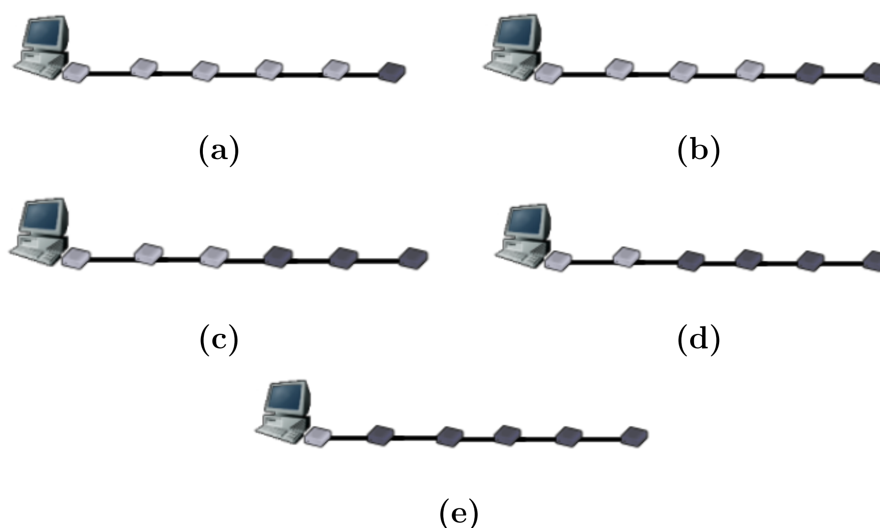

Figura 38: Programando cinco nodos

En la gráfica de la figura 39 se puede apreciar el resultado al reprogramar varios nodos, de uno a cinco, utilizando siempre cinco saltos para alcanzar al último nodo a reprogramar utilizando la topología que se muestra en la figura 38. A pesar de que es la misma cantidad de información que se transmite en cada prueba, y cada uno de ellos realiza siempre el mismo número de saltos, se puede apreciar que a mayor número de nodos a reprogramar mayor es el tiempo en concluirse la reprogramación. Esto es debido a que los nodos que se van a reprogramar realizan un procesamiento extra al escribir en memoria EEPROM el nuevo programa, que además es una operación costosa en energía. En contraste, los nodos que no se reprograman solamente toman los

Tabla IX: Tiempo en reprogramar varios nodos a cinco saltos

Nodos	No Incremental	Incremental
1	02:59.31	02:05.27
2	03:12.64	02:10.46
3	03:25.86	02:16.46
4	03:27.14	02:19.11
5	03:31.58	02:31.66

paquetes, los almacenan en un buffer en RAM y los reenvían sin utilizar la mamoria EEPROM.

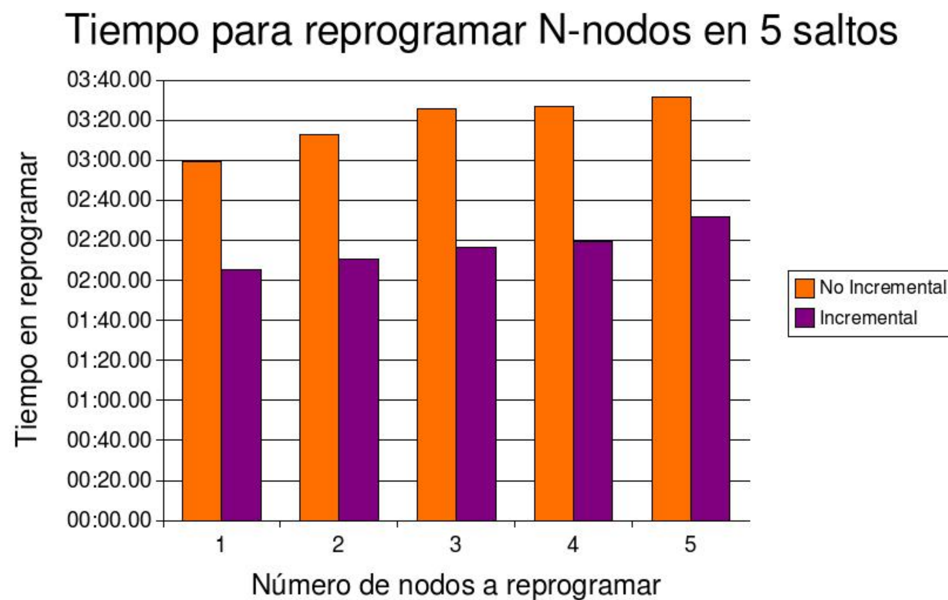

Figura 39: Programando varios nodos a cinco saltos

En la tabla IX se muestran los tiempos logrados al reprogramar de uno a cinco nodos con la topología mostrado en la figura 38. Se puede apreciar que el tiempo va aumentando de forma lineal, ya que sólo el número de nodos a reprogramarse varía, y el tamaño de la red se mantiene constante.

V.6 Resumen

En este capítulo se describieron las pruebas realizadas al mecanismo de reprogramación desarrollado en este trabajo. En general, el objetivo de dichas pruebas fué comprobar el beneficio obtenido al reprogramar de forma incremental. Los resultados muestran claramente que reprogramando de forma incremental se reduce considerablemente la energía consumida por los nodos a reprogramar. Sin embargo para lograr reprogramar los nodos deseados, en ocasiones se necesita que otros nodos intervengan en el proceso fungiendo como retransmisores, los cuales al ser nodos intermediarios, escuchar paquetes y retransmitirlos gastan energía. La energía gastada por los nodos que sólo son retransmisores es menor a la energía consumida por los nodos que se van a reprogramar. Esto es debido a que no escriben los paquetes de datos en EEPROM ni ejecutan los comandos de copiado embebidos en algunos paquetes de datos.

Capítulo VI

Conclusiones, aportaciones y trabajo a futuro

VI.1 Conclusiones

En este trabajo se presenta una propuesta para reprogramar un conjunto de nodos en una red inalámbrica de sensores minimizando el costo de reprogramar al hacerlo de forma incremental. Aunque ya ha habido propuestas anteriores para reprogramar de forma incremental, ninguna trata de reprogramar sólo un conjunto de nodos y se centran en reprogramar toda la red. En este trabajo se selecciona un conjunto de nodos para reprogramar en base a los programas que se encuentran en cada nodo. Otra opción para seleccionar los nodos podría ser en base a alguna lectura de uno de sus sensores.

Como se explica en los capítulos anteriores la energía consumida es un recurso importante en una red inalámbrica de sensores, es por esto que en este trabajo se buscó minimizar el costo de energía al reprogramar un conjunto de nodos. En base a las pruebas realizadas se puede observar que, efectivamente, reprogramando de forma incremental se disminuye el consumo de energía.

Además teniendo como unos de los objetivos específicos el reprogramar sólo un conjunto de nodos se pudieron aplicar técnicas de enrutamiento multicast, minimizando en lo posible el uso de nodos que no se vayan a reprogramar. Sin embargo en ocasiones es imprescindible la utilización de nodos que no se vayan a reprogramar para lograr

llegar a aquellos que si. Estos nodos que no se reprograman, al no almacenar el nuevo programa en EEPROM y utilizar sólo un buffer, minimizan su gasto de energía y utilizan la unidad de radiocomunicación sólo para retransmitir los paquetes que formarán el nuevo programa.

Una red de sensores puede estar constituida por nodos corriendo distintas aplicaciones pero que de alguna manera puedan comunicarse entre si, o inclusive corriendo la misma aplicación, pero siendo los nodos de distinto fabricante. Ésto permite tener una mayor flexibilidad y evita restringir una red de sensores a nodos de un sólo fabricante, además distribuir tareas entre los nodos ejecutando diferentes programas.

Por ello, es importante en este tipo de redes contar con un mecanismo de reprogramación que seleccione un conjunto de nodos, además de minimizar el consumo de energía realizando una reprogramación incremental para aprovecharla en la actividad principal de la red de sensores y no en una tarea secundaria como es reprogramarse.

Con este trabajo se comprobó que es posible seleccionar un grupo de nodos de una red de sensores y reprogramar de forma incremental para disminuir el consumo de energía. En base a las pruebas realizadas, se pudo observar que aún cuando no se realice una actualización, sino un cambio de un programa a otro, es posible encontrar bloques de datos en común permitiendo ahorrar energía al sólo retransmitir las diferencias de los dos programas.

VI.2 Aportaciones

En el desarrollo de este trabajo se obtuvieron varias aportaciones. A continuación se explican cada una de ellas.

- Se modificó el mecanismo de reprogramación Deluge para brindar apoyo a reprogramación incremental.

- Se modificó el mecanismo de reprogramación Deluge para reprogramar un conjunto de nodos seleccionados en base a ciertas reglas.

- Se comprobó que una reprogramación incremental ayuda en minimizar el consumo de energía en una red de sensores cuando es necesario reprogramar un conjunto de nodos.

- Se comprobó que es posible reprogramar un conjunto de nodos en una red inalámbrica de sensores utilizando conceptos de enrutamiento multicast.

- Se diseñó un mecanismo de reprogramación incremental para reprogramar sólo un conjunto de nodos, en lugar de reprogramar toda la red, como presentan en algunos trabajos de reprogramación incremental.

VI.3 Trabajo a futuro

En el desarrollo de este trabajo de investigación se observaron varias posibles mejoras que se proponen explorar como trabajo a futuro. A continuación se describe cada una de ellas.

- Hacer el protocolo de enrutamiento multicast más robusto, de tal manera que se pueda conseguir una recuperación de rutas en caso de que un nodo falle.

- Desarrollar una herramienta para administrar las diferentes versiones de programas con que cuentan los nodos en la red de sensores.

- Evaluar distintos protocolos de enrutamiento para reprogramar un conjunto de nodos.

- Realizar pruebas de desempeño con distintas métricas utilizadas para formar el árbol de enrutamiento.

- Modificar el esquema de reprogramación desarrollado en este trabajo para que el proceso de reprogramación no inicie con mensajes enviados de una PC conectada a un nodo, sino de un nodo que en cualquier momento pueda ser agregado a la red con el propósito de reprogramar un conjunto de nodos.

VI.4 Limitaciones

Este trabajo cuenta con algunas limitaciones en comparación con otros mecanismos de reprogramación. A continuación se describe cada uno de ellos.

- Sólo se puede reprogramar un conjunto de nodos a la vez, no tiene apoyo para mantener distintos árboles de enrutamiento multicast.

- El protocolo de enrutamiento desarrollado no permite recuperar enlaces rotos. Cuando se detecta un enlace perdido algunos nodos quedarían con la imagen del programa incompleta por la imposibilidad de buscar una ramificación del árbol multicast sin enlaces rotos.

- El proceso de reprogramación inicia sólo a través de una PC conectada a un nodo y no puede ser iniciada por un nodo que contenga una versión de un programa más nuevo, como lo hace Deluge.

Bibliografía

Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, y E. Cayirci 2002. “Wireless sensor networks: a survey”. Comput. Networks, 38(4):393–422 p.

Chen, B.-R., K.-K. Muniswamy-Reddy, y M. Welsh 2006. “Ad-hoc multicast routing on resource-limited sensor nodes”. En: “REALMAN ’06: Proceedings of the second international workshop on Multi-hop ad hoc networks: from theory to reality”, Florencia, Italia. Mayo 2006. ACM Press. 87–94 p.

Chipcon 2004. “Cc2420 datasheet, 2.4ghz ieee 802.15.4/zigbee ready rf transceiver datasheet”. Internet. www.chipcon.com. Junio 2004.

Chong, C.-Y. y S. P. Kumar 2003. “Sensor networks: Evolution, opportunities, and challenges.”. En: “Proceedings of the IEEE”. IEEE. 1247-1256 p.

Elson, J. y D. Estrin 2004. “Wireless sensor networks: A bridge to the physical world”. En: Raghavendra, Sivalingam, y Znati (eds). Wireless sensor networks. Kluwer Academic Publishers. Norwell, MA, USA. 3-20 p.

Estrin, D., R. Govindan, J. Heidemann, y S. Kumar 1999. “Next century challenges: scalable coordination in sensor networks”. En: “MobiCom ’99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking”, New York, NY, USA. ACM. 263–270 p.

Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer, y D. Culler 2003. “The nesc language: A holistic approach to networked embedded systems”. En: “PLDI ’03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation”, San Diego, California, USA. ACM. 1–11 p.

Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler, y K. Pister 2000. “System architecture directions for networked sensors”. SIGPLAN Notices, 35(11):93–104 p.

Hills, R. 2001. “Sensing for danger”. online. http://llin.gov/str/JulAug/01/Hills.html. Julio 2001.

Hui, J. W. y D. Culler 2004. “The dynamic behavior of a data dissemination protocol for network programming at scale”. En: “SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems”, Baltimore, MD, USA. ACM Press. 81–94 p.

Jeong, J. y D. Culler 2005. “Incremental network programming for wireless sensors”. Tesis de Maestría, EECS Department, University of California, Berkeley. 69 p.

Jetcheva, J. G. y D. B. Johnson 2001. “The adaptive demand-driven multicast routing protocol for mobile ad hoc networks (admr)”. INTERNET-DRAFT. http://tools.ietf.org/html/draft-ietf-manet-admr-00. Julio 2001.

Koshy, J. y R. Pandey 2005. “Remote incremental linking for energy-efficient reprogramming of sensor networks”. En: “Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on”. Estambul, Turquía. Febrero 2005. IEEE Computer Society. 354- 365 p.

Kulkarni, S. S. y L. Wang 2005. “Mnp: Multihop network reprogramming service for sensor networks”. En: “ICDCS 2005. Proceedings. 25th IEEE International Conference on”, Columbus, OH, USA. Junio 2005. IEEE Computer Society. 7-16 p.

Lee, K. 2001. “Wireless sensing and ieee 1451”. presented at Sensor Conf./Expo. Chicago,Il.

Levis, P. y D. Culler 2002. “Maté: a tiny virtual machine for sensor networks”. En: “ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems”, San Jose, California. Octubre 2002. ACM Press. 85–95 p.

Levis, P. y D. Culler 2004. “The firecracker protocol”. En: “EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC”, Lovania, Bélgica. Septiembre 2004. ACM Press. 3 pp.

Levis, P., N. Lee, M. Welsh, y D. Culler 2003. “Tossim: accurate and scalable simulation of entire tinyos applications”. En: “SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems”, Los Angeles, California, USA. Noviembre 2003. ACM Press. 126–137 p.

Levis, P., N. Patel, D. Culler, y S. Shenker 2004. “Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks”. En: “NSDI’04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation”, Berkeley, CA, USA. USENIX Association. 2–2 p.

Marrón, P. J., M. Gauger, A. Lachenmann, D. Minder, O. Saukh, y K. Rothermel 2006. “Flexcup: A flexible and efficient code update mechanism for sensor networks”. En: “Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)”. Zúrich, Suiza. Febrero 2006. 212-227 p.

Marrón, P. J., A. Lachenmann, D. Minder, J. Hähner, R. Sauter, y K. Rothermel 2005. “TinyCubus: A flexible and adaptive framework for sensor networks”. En: “Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)”. Estambul, Turquía. Febrero 2005. 278-289 p.

Moteiv 2006. “Tmote sky”. online. http://www.moteiv.com/products/tmotesky.php. Febrero 2006.

Perkins, C. E. y E. M. Royer 1999. “Ad-hoc on-demand distance vector routing”. En: “Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications”, New Orleans, LA, USA. Febrero 1999. 90-100 p.

Phillips, L. 2005. “Aqueduct: Robust and Efficient code propagation in heterogeneous Wireless Sensor Networks”. Tesis de Maestría, Univ. Colorado. 53 p.

Reijers, N. y K. Langendoen 2003. “Efficient code distribution in wireless sensor networks”. En: “WSNA ’03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications”, San Diego, CA, USA. Septiembre 2003. ACM Press. 60–67 p.

Stathopoulos, T., J. Heidemann, y D. Estrin 2003. “A remote code update mechanism for wireless sensor networks”. Reporte técnico , CENS. 15 p.

Steere, D. C., A. Baptista, D. McNamee, C. Pu, y J. Walpole 2000. “Research challenges in environmental observation and forecasting systems”. En: “MobiCom ’00: Proceedings of the 6th annual international conference on Mobile computing and networking”, Boston, Massachusetts, USA. Agosto 2000. ACM Press. 292–299 p.

STMicroeletronics 2004. “M25p80 datasheet, 8 mbit, low voltage, serial flash memory with 40mhz spi bus interface”. Internet. http://www.st.com/stonline/books/pdf/docs/8495.pdf. Agosto 2004.

Trigdell, A. 1999. “Efficient algorithms for sorting and synchronization”. Tesis de Maestría, Australian National University. 106 p.

Wang, Q., Y. Zhu, y L. Cheng 2006. “Reprogramming wireless sensor networks: challenges and approaches.”. IEEE Network, 20(3):48-55 p.

Apéndice A

Algoritmos utilizados

En este apéndice se presenta el algoritmo original de rsync, así como el algoritmo para obtener el hash (*message digest*) MD4 de 128 bits. Estos algoritmos se mencionan en este trabajo, sin embargo no se implementaron debido a que ya se encuentran en algunas clases del framework de Java, o en el caso de rsync, en un conjunto de clases llamadas jarsync.

A.1 Algoritmo CRC

El control de redundancia cíclica es un tipo de función que toma como entrada un flujo de datos de cualquier dimensión y produce como saida un valor de tamaño fijo. Regularmente es utilizado para detectar alteraciones accidentales en la transmisión de información.

El algoritmo utilizado en el control de redundancia cíclica es el siguiente:

- Se añaden r bits "0" a la derecha del mensaje (esto es, se añaden tantos ceros como grado tenga el polinomio generador).
- Se divide el polinomio obtenido por el polinomio generador. La división se realiza en módulo 2, que es igual que la división binaria, con dos excepciones:
    1. $1 + 1 = 0$ (no hay acarreo) y

2. 0 - 1 = 1 (no hay acarreo)

- Después se añade el resto de la división al polinomio original.

La elección del polinomio generador es esencial si queremos detectar la mayoría de los errores que ocurran. Uno de los polinomios generadores que más se suelen utilizar es el estándar CCITT:

$$x^{16} + x^{12} + x^5 + 1.$$

Este polinomio permite la detección de:

1. 100% de errores simples.

2. 100% de errores dobles.

3. 100% de errores de un número impar de bits.

4. 100% de errores en ráfagas (en una serie sucesiva de bits) de 16 o menos bits.

5. 99.99% de errores en ráfagas de 18 o más bits.

A.2 Algoritmo Rsync

El algoritmo rsync fué diseñado para actualizar un archivo en una computadora con la finalidad de que sea identico a otro archivo que se encuentra en otra máquina dentro de una red. El algoritmo identifica partes del archivo fuente que son identicas al archivo original, y solo envía aquellos bloques de datos donde no concuerdan.

A.2.1 Algoritmo

Suponer que se tienen dos computadoras α y β. La computadora α tiene acceso al archivo A, mientras que la computadora β tiene acceso al archivo B. Los archivos A y

B son "similares".

El algoritmo rsync consiste en los siguientes pasos:

1. β divide el archivo B en un conjunto de bloques de S bytes. El último bloque puede ser menor que S bytes.

2. Para cada uno de los bloques β calcula dos checksums: un checksum débil de 32 bits llamado *rolling checksum* y un checksum fuerte de 128 bits (MD4).

3. β envía los checksum a α

4. α busca en A en bloques de tamaño S (a cualquier *offset* no necesariamente múltiplos de S) que tengan el mismo checksum débil y fuerte.

5. α envía a β una secuencia de instrucciones para construir una copia de A. Cada instrucción hace referencia a un bloque de B o es un bloque de datos. El bloque de datos que se envía es para aquellas secciones de A que no se encontraron en ningún bloque de B.

El resultado final es que β obtiene una copia de A, pero sólo piezas de A que no se encuentran en B (más el envío de los checksums y los indices de los bloques) son enviados.

A.2.2 Rolling Checksum

El checksum débil utilizado en este algoritmo necesita tener la prioridad de ser rápido en calcular el checksum de un buffer $X_2 \ldots X_{n+1}$ dado el checksum del buffer $X_1 \ldots X_n$ y el valor de los bytes $X_1 \ldots X_{n+1}$.

Este algoritmo de checksum débil está definido por:

$$a(k,l) = \left(\sum_{i=k}^{l} X_i \right) mod\ M$$

$$b(k,l) = \left(\sum_{i=k}^{l} (l-i+1) X_i \right) mod\ M$$

$$s(k,l) = a(k,l) + 2^{16} \cdot b(k,l)$$

Donde $s(k,l)$ es el *rolling checksum* de los bytes $X_k \ldots X_l$. Para simplicidad y rapidez se define $M = 2^{16}$.

Una propiedad importante de este checksum es que valores sucesivos pueden ser calculados de forma eficiente y rápida usando recurrencia:

$$a(k+1, l+1) = (a(k,l) - X_k + X_{l+1}) mod\ M$$

$$b(k+1, l+1) = (b(k,l) - (l-k+1) X_k + a(k+1, l+1)) mod\ M$$

De esta manera el checksum puede ser calculado para bloques de longitud S en todos los *offsets* posibles en un archivo con muy poco procesamiento. El checksum fuerte es calculado para cada bloque donde el checksum debil concuerde. Esto se hace debido a que existe una pequeña probabilidad que aunque concuerden los checksum debiles, el bloque de datos no sea igual.

A.3 Algoritmo MD4

El algoritmo MD4 recibe un mensaje de entrada de tamaño arbitrario y produce una salida de 128 bits (*message digest*). Se considera que es computacionalmente inviable que dos mensajes produzcan el mismo *message digest*. El algoritmo MD4 es utilizado en aplicaciones de firmas digitales.

Se realizan los siguientes pasos para calcular el "*message digest*" de un mensaje:

Paso 1 El mensaje de entrada es extendido para que su tamaño en bits sea múltiplo de 512 menos 64.

Paso 2 Una representación de 64 bits b (el tamaño del mensaje antes de extenderlo) se adjunta al paso anterior. En caso de que b sea mayor de 2^{64}, sólo los 64 bit de bajo orden son usados. En este paso, después de extender el mensaje más los 64 bits de b, el mensaje resultante es exactamente múltiplo de 512.

M[0...N-1] son los *words* del mensaje resultante. N es múltiplo de 16.

Paso 3 Se usa un buffer de cuatro *words* (A,B,C,D) para calcular el *message digest*. Donde A,B,C,D son de 32 bits. Estos registros son inicializados con los siguientes valores hexadecimales:

A:01 23 45 67

B:89 ab cd ef

C:fe dc ba 98

D:76 54 32 10

Paso 4 Se definen tres funciones auxiliares, donde cada una de ellas recibe como parámetro tres *words* de 32 bits y producen como salida un *word* de 32 bits.

F(X,Y,Z) = XY v not(X) Z

G(X,Y,Z) = XY v XZ v YZ

H(X,Y,Z) = X xor Y xor Z

Se hace lo siguiente:

```
For i = 0 to N/16-1 do
        /* copia el bloque i en X. */
        For j = 0 to 15 do
          Set X[j] to M[i*16+j].
        end /* fin del ciclo j */

        /* guarda A como AA, B como BB, C como CC, y D como DD. */
        AA = A
        BB = B
        CC = C
        DD = D

        /* Round 1. */
        /*  [abcd k s] define la siguiente operación
             a = (a + F(b,c,d) + X[k]) <<< s. */
        /* Hacer las siguientes 16 operaciones. */
        [ABCD  0  3]  [DABC  1  7]  [CDAB  2 11]  [BCDA  3 19]
        [ABCD  4  3]  [DABC  5  7]  [CDAB  6 11]  [BCDA  7 19]
        [ABCD  8  3]  [DABC  9  7]  [CDAB 10 11]  [BCDA 11 19]
        [ABCD 12  3]  [DABC 13  7]  [CDAB 14 11]  [BCDA 15 19]

        /* Round 2. */
        /* [abcd k s] define la siguiente operación
             a = (a + G(b,c,d) + X[k] + 5A827999) <<< s. */

 /* Hacer las siguientes operaciones. */
        [ABCD  0  3]  [DABC  4  5]  [CDAB  8  9]  [BCDA 12 13]
        [ABCD  1  3]  [DABC  5  5]  [CDAB  9  9]  [BCDA 13 13]
```

```
    [ABCD  2  3]  [DABC  6  5]  [CDAB 10  9]  [BCDA 14 13]
    [ABCD  3  3]  [DABC  7  5]  [CDAB 11  9]  [BCDA 15 13]

    /* Round 3. */
    /* [abcd k s] define la siguiente operación
         a = (a + H(b,c,d) + X[k] + 6ED9EBA1) <<< s. */
    /* Hacer las siguientes 16 operaciones. */
    [ABCD  0  3]  [DABC  8  9]  [CDAB  4 11]  [BCDA 12 15]
    [ABCD  2  3]  [DABC 10  9]  [CDAB  6 11]  [BCDA 14 15]
    [ABCD  1  3]  [DABC  9  9]  [CDAB  5 11]  [BCDA 13 15]
    [ABCD  3  3]  [DABC 11  9]  [CDAB  7 11]  [BCDA 15 15]

    /* Realizar las siguientes operaciones */
    A = A + AA
    B = B + BB
    C = C + CC
    D = D + DD

  end /* fin del ciclo i */
```

Paso 4 Salida. El *message digest* producido es A,B,C,D. Concatenados en ese orden produciendo un mensaje de 128 bits.