Universidad de la República

Facultad de Ingeniería

ContikiWSN

Estudio, Análisis y Diseño de Redes de Sensores Inalámbricas con Contiki OS

Integrantes: Ignacio de Mula
Germán Ferrari
Gabriel Firme

Tutor: Leonardo Steinfeld

Acrónimos

CC	Consistency Check
CCA	Clear Channel Assessment
CPU	Central Processing Unit
CSMA	Carrier Sense Multiple Access
DAG	Directed Acycling Graph
DAO	Destination Advertisement Object
DIS	DODAG Information Object
DMA	Direct Memory Access
DSSS	Direct Sequence Spread Spectrum
DTSN	DAO Trigger Sequence Number
ETX	Expected Transmission Count
IEEE	Institute of Electrical and Electronics Engineers
ICMP	Internet Control Message Protocol
IETF	Internet Ingeering Task Force
IP	Interntet Protocol
LLN	Low-power and Lossy Network
LPM	Low Power Mode
LR-WPAN	Low-rate Wireles Area Network
MAC	Medium Access Control
PAN	Personal Area Network
RDC	Radio Duty Cycling
RPL	Routing Protocol for Low-power and Lossy Networks
RTT	Round Trip Time
TCP	Transmission Control Protocol

UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
uIP	micro IP
WSN	Wireless Sensor Network

Índice de contenido

Acrónimos....3
Capítulo 1:Introducción....11
1.1Resumen....11
1.2Objetivos....12
1.3Metodología....12
1.4Hardware 13
1.5Descripción de Estructura de Capas....15
1.5.1Capa física y Capa MAC (IEEE 802.15.4)....16
1.5.2Capa de Red....17
1.5.3Capa de Aplicación....17
1.6Requerimientos ContikiWSN....18
1.6.1Introducción....18
1.6.2Requerimientos a nivel de aplicación....18
1.6.3Requerimientos a nivel de Red....19
1.6.4Requerimientos a nivel de Enlace....19
1.6.5Requerimiento a nivel de topología....20
Capítulo 2:Sistema Operativo Contiki....21
2.1Introducción....21
2.2Arquitectura....21
2.2.1Procesos....21
2.2.2Eventos....21
2.2.3Protothreads....22
2.2.4Poll y Post Sincrónico....22
2.3Timers....22
2.3.1Etimers....22
2.3.2Ctimers....23
2.3.3Rtimers....23
Capítulo 3:Capa MAC....24
3.1Introducción....24
3.2ContikiMAC....24
3.2.1Descripción General....24
3.2.2Implementación Base de ContikiMAC....26
3.2.3Implementación de las CCA's....29
3.2.4Ejemplo relevado en el laboratorio....29
Capítulo 4:Capa de Red....31
4.1Introducción....31
4.2uIP – API (Application Program Interface)....31
4.2.1API orientada a eventos....31
4.2.2Loop de Control Principal....32
4.2.3Retransmisión de datos....33
4.2.4Cierre de Conexiones....33
4.2.5Reporte de errores....34
4.2.6Escucha de puertos....34
4.2.7Apertura de conexiones....34
4.3Protosockets....35
4.4Implementación de los protocolos en uIP....35
4.4.1Re-ensamblado de los fragmentos IP....35
4.4.2Manejo de TCP....36
4.4.3Ventana Deslizante....36

4.4.4Control de Flujo.....36
4.4.5Control de Congestión.....36
4.4.6Datos Urgentes.....36
4.4.7Cálculo de sumas de comprobación.....37
4.5IPv6 en Contiki.....37
4.6RPL - Ruteo en WSNs.....37
4.6.1Introducción al protocolo RPL.....38
4.6.2Rank.....39
4.6.3Procesamiento de Mensajes RPL.....41
4.6.4Envío de datos al sink.....43
4.6.5Ruteo.....43
4.6.6Rpl global repair.....44
4.6.7Rpl local repair.....44
4.7Bug RPL.....45
4.7.1Introducción.....45
4.7.2Alcance del Bug.....45
4.7.3Reproducción del Bug.....45
4.7.4El Bug en el código.....47
4.7.5Reparación del Bug.....49
4.8Algoritmo Trickle.....50
4.8.1Principales características del Algoritmo.....51
4.8.2Descripción del Algoritmo.....52
4.8.3Usos típicos de Trickle.....53
4.8.4Algunos comentarios.....53
Capítulo 5:Capa Aplicación.....54
5.1Introducción.....54
5.2Aplicación rpl-collect.....54
5.2.1Nodo sink.....54
5.2.2Nodo sender.....55
5.3Aplicación Shell.....56
Capítulo 6:Diseño e Implementación.....57
6.1Introducción.....57
6.2Compilación y MakeFiles.....57
6.3Optimización de memoria RAM y ROM.....58
Archivos del código modificados.....60
6.4Modificación y Uso de Trickle.....60
6.4.1 Trickle Original.....60
Envío de mensajes Trickle.....61
Recepción de mensajes Trickle.....61
6.4.2 Trickle para IPv6.....61
Inicialización de Trickle IPv6.....61
Envío de mensajes Trickle IPv6.....61
Recepción de mensajes Trickle IPv6.....61
Configuración de Parámetros.....62
Utilización de Trickle.....62
Archivos del código creados y modificados.....62
6.5Detección de pérdida de comunicación con el sink.....63
Algoritmos para detectar perdida de comunicación en Contiki.....63
Detección de Perdida de Comunicación en ContikiWSN.....64
Archivos del código modificados.....65
6.6Buffereo de datos y envío de datos guardados.....65
Archivos del código modificados y creados:.....66
6.7Sincronización de la hora en la red.....67

Archivos del código modificados....68
6.8Utilización de la aplicación Shell....68
Archivos del código creados y modificados 69
6.9Interfaz collect-view....70
Archivos del código modificados....71
6.10Espacio libre luego de la aplicación....71
6.11Software utilizado para el desarrollo....72
Capítulo 7:Pruebas y Evaluación....73
7.1Introducción....73
7.2Herramientas Utilizadas....73
7.2.1Introducción....73
7.2.2Log Listener....74
7.2.3Radio Logger....75
7.2.4Timeline....76
7.3Simulation Visualyzer....77
7.3.1Msp Code Watcher....78
7.4Estimación de energía....79
7.5Energest....79
7.5.1Deducción de las ecuaciones de cálculo de consumo de collect-view....80
7.5.2Análisis del consumo en base a medidas de laboratorio....81
7.6Metodología de análisis del consumo....81
7.7Deducción de los tiempos y de potencias involucrados....83
7.7.1Transmisión....83
7.7.2Recepción....84
7.7.3Análisis del consumo de la CPU....85
7.7.4Análisis del Broadcast....86
7.7.5Análisis de las CCA con una aproximación geométrica....86
7.8Ecuaciones de estimación de Consumo....87
7.9Ecuaciones expresadas en relación de los bytes para casos de tráfico RPL....88
7.10Proyección del consumo para otros escenarios....90
7.10.1Caso: CCAs....90
7.10.2Caso: Broadcast....91
7.10.3Suma de los efectos....92
7.11Pruebas de Aplicación....93
7.11.1Prueba de Trickle. Simulación....93
7.11.2Resultados....94
7.11.3Pruebas con los motes....96
7.12Prueba con motes durante dos semanas....98
7.13Prueba con mayor cantidad de nodos....100
7.14Comparación de resultados....101
Capítulo 8:Conclusiones....103
Referencias....104
Anexo I: Implementación de ContikiOS....107
Anexo II: Estructura de los mensajes de Control de RPL....114
Anexo III: Ejemplo de Perdida de comunicación....118
Anexo IV: Simulador COOJA....123
Anexo V: Ejemplo Mensajes de Control de RPL....129
Anexo VI: Protocolo Deluge....133
Anexo VII: Rime....134
Anexo VIII: Gestión....136

Índice de Figuras

Figura 1: Red de sensores inalámbricas....11
Figura 2: Tmote Sky....13
Figura 3: Topología....20
Figura 4: Escucha de bajo consumo con ContikiMAC....24
Figura 5: Transmisión unicast....25
Figura 6: Transmisión broadcast....25
Figura 7: Comparación entre ContikiMAC y X-MAC....26
Figura 8: Diagrama de flujo de transmisión y recepción de ContikiMAC....27
Figura 9: CCA medido con osciloscopio....28
Figura 10: Detección de transmisión medida con osciloscopio....28
Figura 11: Transmisión entre 2 nodos medida con el osciloscopio....30
Figura 12: Bucle de control principal....33
Figura 13: Definición de una conexión en uIP en el código....34
Figura 14: Ejemplo de camino al Sink....39
Figura 15: Ejemplo de generación de rutas....43
Figura 16: Ejemplo de ruteo....44
Figura 17: Topología de la simulación....45
Figura 18: Tabla de ruteo del nodo 3....46
Figura 19: Topología luego de aislar al nodo 3....46
Figura 20: Tabla de ruteo del nodo 2....47
Figura 21: Tabla de ruteo del nodo 2....49
Figura 22: RDC promedio original de los nodos en la simulación....50
Figura 23: RDC promedio de los nodos en la simulación, con el bug arreglado....50
Figura 24: Esquema de comunicación one-to-many....51
Figura 25: Ejemplo de Camino al Sink....63
Figura 26: Diagrama de tiempo que indica cuando se intenta enviar mensajes al sink....66
Figura 27: Interfaz collect-view adaptada con ContikiWSN....70
Figura 28: Control Panel....73
Figura 29: Log Listener....74
Figura 30: Ejemplo de filtrado en Log Listener....74
Figura 31: Opciones del Log Listener....74
Figura 32: Diferenciación de nodos por colores....75
Figura 33: Radio Logger....75
Figura 34: Opciones del Radio Logger....75
Figura 35: Timeline....76
Figura 36: Ejemplo de Timeline....76
Figura 37: Opciones de Timeline....76
Figura 38: Propiedades de Timeline....77
Figura 39: Simulation Visualyzer....77
Figura 40: Opciones adicionales de Simulation Visualyzer....77
Figura 41: MSP Code Watcher....78
Figura 42: Watchpoints en la Timeline....79
Figura 43: Circuito de medida....82
Figura 44: Datos procesados numéricamente....82
Figura 45: Detalle de los paquetes....83
Figura 46: Medida de transmisión con el osciloscopio....84
Figura 47: Medida de recepción con el osciloscopio 85
Figura 48: Caracterización del consumo de la CPU....85
Figura 49: Análisis de broadcast....86
Figura 50: Log Listener, Radio Logger y Timeline de COOJA....88

Figura 51: Broadcast medido con osciloscopio......89
Figura 52: Broadcast en la Timeline del simulador......89
Figura 53: Consumo distintos ciclos de escucha en una hora......91
Figura 54: Consumo debido a Broadcast en una hora por ciclos......91
Figura 55: Efectos de escuchas de CCA y de Broadcast por ciclo......92
Figura 56: Suma de Consumo de CCA y Broadcast......92
Figura 57: Red utilizada en la simulación......93
Figura 58: Registro histórico de consumo de la simulación......94
Figura 59: Discriminación del consumo promedio de la simulación......95
Figura 60: RDC promedio de la simulación......95
Figura 61: Registro histórico de consumo de la prueba......96
Figura 62: Discriminación del consumo promedio de la prueba......97
Figura 63: RDC promedio de la prueba......97
Figura 64: Topología de la prueba continuada con motes......98
Figura 65: Discriminación del consumo promedio de la prueba......99
Figura 66: RDC promedio de la prueba......99
Figura 67: Distribución de la red en el IIE......100
Figura 68: Topología de las simulaciones......102
Figura 69: Estructura de carpetas de Contiki......112

Índice de Tablas

Tabla 1: Consumos típicos de operación del Tmote Sky....................14
Tabla 2: Condiciones típicas de operación de la radio CC2420 (Tmote Sky Datasheet)....................15
Tabla 3: Parámetros de modulación....................16
Tabla 4: Resultados del análisis del ejemplo rpl-collect....................89
Tabla 5: Datos relevantes del collect-view de la simulación 96
Tabla 6: Datos relevantes del collect-view de la prueba....................98
Tabla 7: Datos relevantes del collect-view de la prueba continuada 100
Tabla 8: Resultados de collect-view de la prueba en el IIE....................101
Tabla 9: Comparación de resultados simulados....................102

Capítulo 1: Introducción

1.1 Resumen

Las redes de sensores inalámbricas están formadas por varios nodos sensores los cuales típicamente recolectan datos y se lo envían a un nodo recolector que procesa los datos recibidos.

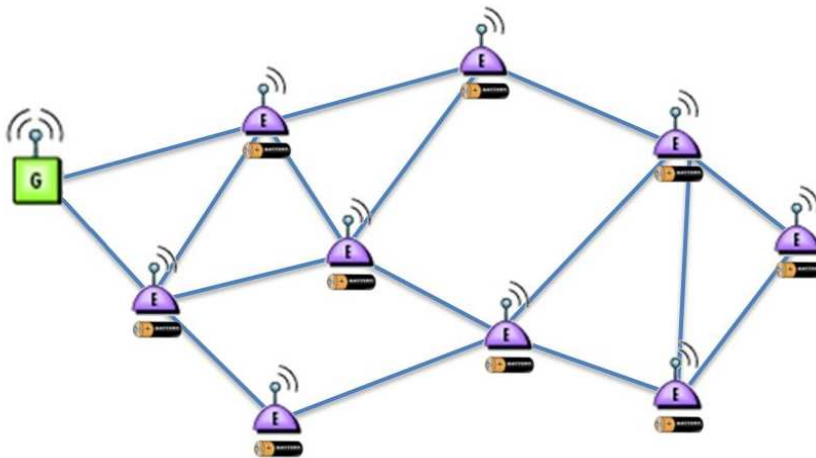


Figura 1: Red de sensores inalámbricas

Cada nodo sensor o sender debe enviar sus datos hacia el nodo recolector o sink. Para ello, puede que un nodo sensor deba enviar sus datos a otro nodo intermedio para que éste los reenvíe y lleguen al sink.

En este tipo de redes es muy importante minimizar el consumo de los nodos de forma de aumentar la autonomía energética de modo que no sea necesario el cambio de la batería de cada nodo, ya que es una tarea tediosa en redes de gran porte.

En este proyecto se utilizó Contiki, un sistema operativo que está diseñado para sistemas con recursos limitados de procesamiento, energía y memoria como lo son las redes de sensores inalámbricas. Se analizaron las distintas capas del sistema, de manera elegir las opciones más convenientes a los efectos de nuestro proyecto. En particular se estudiaron los dos stacks de comunicaciones que ofrece Contiki. Una versión de TCP/IP adaptada para las redes de sensores inalámbricas y Rime que es un stack desarrollado por el equipo de Contiki. Finalmente se decidió utilizar IPv6 ya que es un protocolo estándar.

Contiki presenta una aplicación de recolección de datos llamada rpl-collect. Se mejoraron las funcionalidades de esta aplicación. En particular para que se pudiera configurar la red y para que se pudieran recuperar datos en caso de que un nodo pierda comunicación con el nodo sink. Además se analizó el consumo de los nodos al utilizar el sistema operativo con el fin de poder realizar proyecciones del consumo de los nodos al variar algunos parámetros de los mecanismos que el sistema utiliza.

En la primer parte del documento se presenta las características del sistema operativo Contiki. Luego se expone el diseño e implementación de las modificaciones realizadas al sistema. Por último se describe las pruebas de laboratorio y de campo que se realizaron y los resultados que se obtuvieron.

1.2 Objetivos

En este proyecto se propuso estudiar el sistema operativo Contiki, y a su vez los algoritmos y protocolos que este sistema operativo implementa. Se propuso crear una referencia que introduzca los conceptos fundamentales para trabajar con Contiki. También se propuso implementar una aplicación de una red de sensores inalámbricas mejorando las prestaciones de las aplicaciones ya existentes y analizar el consumo de los nodos al utilizar el sistema.

1.3 Metodología

Antes del comienzo del proyecto, uno de los integrantes del grupo en el curso de Sistemas Embebidos en Tiempo Real utilizó el core del sistema operativo Contiki para crear un Datalogger. El proyecto consistía en una plataforma la cual sensaba la temperatura y almacenaba las medidas registrando el tiempo en que se habían tomado. Luego se podían recuperar los datos almacenados a través de comandos. De Contiki se utilizó solo la estructura de protothreads (ver capítulo 2.2.3), la cual se portó al compilador IAR.

Luego de culminado el curso, se dio la posibilidad de seguir trabajando con el sistema operativo en el proyecto de grado. En una primera instancia se estudiaron todos los papers publicados hasta la fecha por los creadores de Contiki. Se tomó conocimiento que Contiki es mucho más que un sistema operativo multitarea para plataformas de recursos limitados, sino que además propone una solución completa para comunicación por radio. Los papers daban una idea de lo que el Sistema Operativo podía brindar pero de forma muy abstracta y poco práctica. Durante el estudio de los papers se realizaron varios seminarios con el tutor y docentes del IIE (Instituto de Ingeniería Eléctrica de la Universidad de la República). En estos se expusieron los siguientes temas: Contiki, ContikiMAC, Rime, uIP, RPL y COOJA.

El siguiente paso fue la lectura de código. Una primer pregunta que surgió fue si portar el código al compilador IAR para poder hacer debugging y correr el código línea por línea. Esta opción fue descartada debido a que se conoció el simulador COOJA el cual tiene entre otras esta funcionalidad. La lectura de código fue una tarea compleja, debido a la falta de documentación y los pocos comentarios en éste. De todas maneras logramos entender los módulos que nos propusimos. Al poco tiempo de esto se creó la wiki de Contiki la cual fue de gran ayuda, ya que uno de los principales problemas que tuvimos fue la falta de tutoriales.

Luego nos dividimos en dos grupos, de los cuales uno se encargo de la implementación de código y el otro del estudio del consumo del Sistema Operativo.

Para la implementación de código el primer problema que surgió fue la falta de espacio tanto de memoria RAM como de flash en el microcontrolador. Una vez resuelto este problema se implementó una aplicación que detecta cuando un nodo se queda sin comunicación y de manera inteligente guarda en un buffer los datos sensados en vez de intentarlos enviar hasta recuperar la comunicación, momento en el cual se empiezan a enviar los datos buffereados. El nuevo código funcionó sin mayores problemas.

Debido a que las nuevas implementaciones de código funcionaron correctamente se propuso implementar código para poder diseminar datos del nodo sink a los nodos senders. Además se adaptó la interfaz de recolección de datos (collect-view) para que funcionara interactivamente con la aplicación desarrollada.

Luego de que casi estuviese finalizada la implementación se publicó la nueva versión 2.5 de Contiki.

Se migró el código hecho a la nueva versión, tarea que no fue trivial pero tampoco requirió más de un día de trabajo. Como se tocó el código del core del Sistema Operativo para la migración hubo que chequear que el código agregado siguiese siendo coherente.

El aprendizaje del mecanismo ContikiMAC se realizó a partir del análisis del código y de algunos artículos. Para el análisis del código se utilizó el simulador para redes de sensores inalámbricas COOJA (ver Anexo II) para comprender mejor el funcionamiento. Posteriormente se realizaron mediciones en el laboratorio para corroborar el correcto funcionamiento de la aplicación desarrollada y para poder caracterizar el consumo de Contiki durante el funcionamiento de una red de sensores inalámbricas. A partir de estos datos se formularon ecuaciones descriptivas del consumo para tráfico asociado al protocolo de ruteo RPL. Finalmente se realizaron proyecciones de consumo modificando el tiempo de verificación del canal y considerando distintos tipos de tráfico y tiempo de trabajo.

1.4 Hardware

La plataforma utilizada durante el proyecto fue Tmote Sky. Esta plataforma fue desarrollada originalmente en la Universidad de Berkeley y se utiliza en aplicaciones de redes de sensores de bajo consumo. Lleva integrados tanto los sensores como la radio, antena y microcontrolador. Los datos que se presentan a continuación son extraídos de la hoja de datos del fabricante [1].

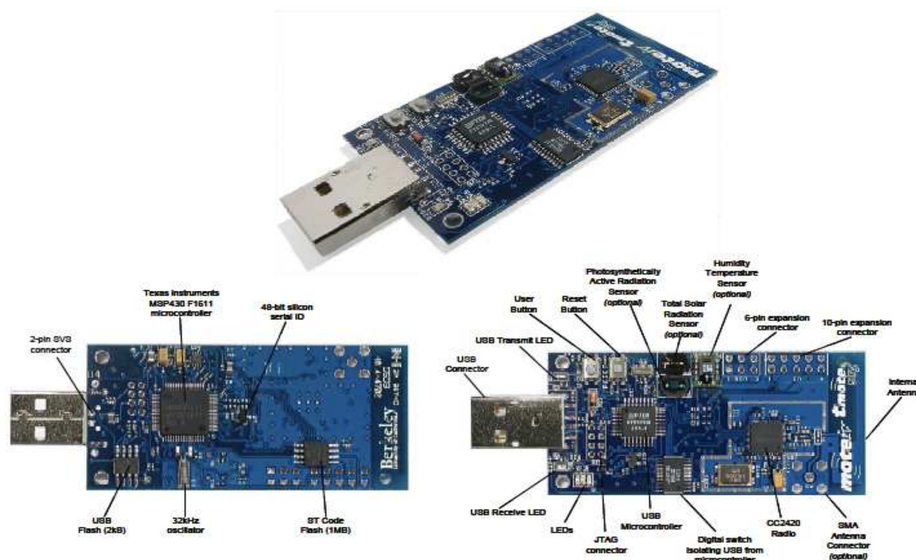


Figura 2: Tmote Sky

(Tmote Sky Datasheet)

El Tmote Sky posee un microcontrolador MSP430 F1611. Este procesador RISC de 16 bits consume muy poca batería, tanto en el estado activo como durante el modo sleep. Para reducir al máximo este consumo, permanece en modo sleep durante la mayoría del tiempo, se despierta tan rápido como puede para procesar, enviar, y entonces vuelve a dormirse.

Utiliza un controlador USB de FTDI para comunicarse con el procesador y maneja una radio Chipcon CC2420, la cual implementa el estándar IEEE (Institute of Electrical and Electronics Engineers) 802.15.4. La antena interna "Invertid –F micro strip" es una antena seudo-omnidireccional que tiene un

margen de potencia de transmisión de 40 dBm y una sensibilidad de recepción de -94 dBm, la cual puede alcanzar los 50 metros dentro de un edificio y los 125 metros en el exterior.

Las características esenciales de Tmote Sky son:

- Transmisor Chipcon inalámbrico de 250Kbps 2.4GHz IEEE 802.15.4
- Interactúa con otros dispositivos IEEE 802.15.4
- Microcontrolador MSP430 Texas Instruments de 8MHz (10Kb de RAM y 48 Kb de Flash)
- ADC, DAC, supervisor de voltaje y controlador DMA (Acceso Directo a Memoria) integrado
- Antena, sensores de humedad, temperatura y luz
- Muy bajo consumo
- Rápido en despertar del modo sleep (<6 µs)
- Hardware para encriptación y autenticación de la capa de enlace
- Programación y recolección de datos por USB
- 16 pines para soportar una expansión y conector de antena opcional SMA (SubMiniature type A)

Los sensores de humedad/temperatura están construidos por Sensiron AG, producidos con procesadores CMOS y unido con un dispositivo ADC de 14 bits.

Energía

El Tmote Sky es alimentado por dos pilas AA. El rango de operación del Tmote Sky es de 2.1 a 3.6 V DC, sin embargo la tensión debe ser de al menos 2,7 V en la programación del microcontrolador flash o flash externo.

En la siguiente tabla se muestran los consumos típicos de operación del Tmote Sky (Tmote Sky Datasheet):

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

Tabla 1: Consumos típicos de operación del Tmote Sky

Radio

Tmote Sky tiene la radio CC2420 de Chipcon para las comunicaciones inalámbricas. El CC2420 es un radio compatible con IEEE 802.15.4 y proporciona la capa física (PHY) y algunas funciones de la capa MAC.

El CC2420 es controlado por el microcontrolador MSP430 a través del puerto SPI y una serie de E / S digitales y líneas de interrupciones. La radio puede ser apagada por el microcontrolador con el fin de obtener un ciclo de trabajo bajo.

	MIN	NOM	MAX	UNIT
Supply voltage during radio operation (Vreg on)	2.1		3.6	V
Operating free air temperature	-40		85	°C
RF frequency range	2400		2483.5	MHz
Transmit bit rate	250		250	kbps
Nominal output power	-3	0		dBm
Programmable output power range		40		dBm
Receiver sensitivity	-90	-94		dBm
Current consumption: Radio transmitting at 0 dBm		17.4		mA
Current consumption: Radio receiving		19.7		mA
Current consumption: Radio on, Oscillator on		365		µA
Current consumption: Idle mode, Oscillator off		20		µA
Current consumption: Power Down mode, Vreg off			1	µA
Voltage regulator current draw	13	20	29	µA
Radio oscillator startup time		580	860	µs

Tabla 2: Condiciones típicas de operación de la radio CC2420 (Tmote Sky Datasheet)

1.5 Descripción de Estructura de Capas

Contiki [2] [3]es un sistema operativo open source, multi-tarea desarrollado para plataformas con recursos limitados de procesamiento, energía y memoria. Este fue implementado por un grupo de académicos del “Swedish Institute of Computer Science” liderado por Adam Dunkels [4]. Sus creadores lo desarrollaron pensando en “Internet of things”, que tiene como idea que en un futuro los objetos que nos rodean tengan la capacidad de comunicarse de forma inalámbrica y de esta manera se podrán ubicar y controlar remotamente.

El sistema operativo ofrece dos stacks de comunicaciones. Por un lado una versión de TCP/IP adaptada a las redes de sensores inalámbricas y por otro lado el stack Rime [5]. Este último es un stack modular, estructurado en capas muy simples de manera de que los encabezados sean lo más pequeños posibles. Las distintas capas de Rime implementan funcionalidades asociadas a los servicios de comunicación que se quieren brindar (ver Anexo V).

La pila de protocolos de Contiki comienza con el driver de la radio, el cual se encarga de enviar y recibir datos crudos por aire. Un nivel más arriba se encuentra la capa de RDC (Radio Duty Cycling) la cual se encarga de prender y apagar la radio para ahorrar energía. Sobre esta capa se encuentra la de acceso al medio (MAC) la cual intenta evitar las colisiones entre las transmisiones de radio de los nodos. Luego se encuentra la capa de red, que se encarga del enrutamiento de los paquetes. Finalmente se encuentra la capa de aplicación en donde se interpretan y procesan los datos recibidos y donde se generan datos para enviar.

El driver de la radio es el código que se encarga de manejarla. Se utiliza la radio CC2420. El driver se encarga de prender, apagar, transmitir y recibir datos.

El mecanismo que se utiliza para la capa RDC es ContikiMAC. Éste se encarga de apagar la radio la mayor cantidad de tiempo. La radio consume mucha energía y este es un recurso limitado debido que los nodos son alimentados por baterías.

El mecanismo que se utiliza para la capa MAC es CSMA (Carrier Sense Multiple Access). Antes de transmitir se escucha el canal para saber si ya hay alguien transmitiendo y de esta manera evitar colisiones.

El mecanismo que se utiliza para la capa de red es SICSlowpan [6]. Es una implementación de los creadores de Contiki de la capa de adaptación 6lowPAN (IPv6 sobre IEEE 802.15.4) [7] [8] para transmitir

tramas IPv6 sobre la capa física de IEEE 802.15.4 en redes de sensores inalámbricas.

1.5.1 Capa física y Capa MAC (IEEE 802.15.4)

En redes de sensores inalámbricas con tasas bajas de transmisión de datos (Low-rate Wireless Personal Area Network, LR-WPAN), el estándar IEEE 802.15.4 define el nivel físico (PHY) y el control de acceso al medio (MAC) [9]. El estándar ha sido desarrollado dentro de la red de área 802.15 personal (PAN) y es la base sobre la que se define la especificación de ZigBee [10], cuyo propósito es ofrecer una solución completa para este tipo de redes construyendo los niveles superiores de la pila de protocolos que el estándar no cubre.

Capa Física

La capa física del estándar 802.15.4 se divide en las subcapas PHY data service y PHY management que son las encargadas de recibir y transmitir mensajes a través del medio de radio.

Opera en una de tres posibles bandas de frecuencia de uso no regulado:

- 868-868,8 MHz: Europa, permite un canal de comunicación (versión de 2003), extendido a tres en la revisión de 2006.

- 902-928 MHz: Norte América, hasta diez canales (2003) extendidos a treinta (2006).

- 2400-2483,5 MHz: uso en todo el mundo, hasta dieciséis canales (2003, 2006).

La versión original del estándar, del año 2003, especifica dos niveles físicos basados en DSSS (Direct Sequence Spread Spectrum): uno en las bandas de 868/915 MHz con tasas de 20 y 40 kbps; y otra en la banda de 2450 MHz con hasta 250 kbps.

Como se puede ver en la Tabla 3, se dispone de diferentes velocidades de transmisión dependiendo de la frecuencia que se utilice.

Los parámetros de modulación para ambas PHY se resumen en la siguiente tabla:

PHY	Banda	Parámetros de los datos			Parámetros del chip	
		Velocidad de bits (Kb/s)	Velocidad de símbolos (Kbaud)	Modulación	Velocidad de chip (Kchip/s)	Modulación
868/915 MHz PHY	868.0-868.6 MHz	20	20	BPSK	300	BPSK
	902.0-928 MHz	40	40	BPSK	600	BPSK
2.4 GHz PHY	2.4-4.4835 GHz	250	62.5	16-ary ortogonal	2000	O-QPSK

Tabla 3: Parámetros de modulación

Capa MAC

El control de acceso al medio (MAC) transmite tramas MAC usando para ello el canal físico. Además del servicio de datos, ofrece una interfaz de control y regula el acceso al canal físico y a la señalización de la red. También controla la validación de las tramas y las asociaciones entre nodos, y garantiza slots de tiempo. Por último, ofrece puntos de enganche para servicios seguros.

El tamaño máximo de paquete en 802.15.4 es de 127 bytes. Los paquetes son pequeños porque IEEE 802.15.4 está diseñado para dispositivos con velocidades de datos bajas. Debido a que la capa MAC agrega un encabezado a cada paquete, la cantidad de datos disponible de un protocolo de capa superior o de una aplicación es entre 86 y 116 bytes. Por lo tanto, los protocolos de capa superior a menudo añaden mecanismos para fragmentar grandes paquetes de datos en múltiples tramas 802.15.4.

Contiki contiene 2 drivers para la capa MAC, CSMA y NullMAC. El mecanismo por defecto es el CSMA, que es el único que tiene retransmisión de paquetes en caso de haber colisiones. La capa MAC recibe los paquetes entrantes de la capa RDC y usa esta capa para transmitir los paquetes.

1.5.2 Capa de Red

Cuando se inició el proyecto Contiki tenía publicada la versión 2.4 que permitía optar entre IPv4 o IPv6. En ese momento ya estaba en curso el desarrollo de la versión 2.5. En el transcurso del proyecto los desarrolladores anunciaron que Contiki se orientaba a tener IPv6 por defecto y a abandonar IPv4.

De allí en más nos centramos en el estudio de IPv6 en el contexto de Contiki, particularmente en el protocolo de ruteo RPL y en las definiciones de 6lowPAN. También fue necesario estudiar los algoritmos Trickle y Deluge, mecanismos que se encargan de propagar y mantener actualizado la información de ruteo y el código respectivamente en redes de sensores inalámbricas. En caso de RPL Trickle está embebido como parte del mecanismo de actualización de rutas. Adicionalmente fue portado a IPv6 para poder utilizarlo en la aplicación desarrollada.

1.5.3 Capa de Aplicación

La capa de aplicación utiliza las capas inferiores para enviar mensajes a otros nodos. Esta capa se encarga de generar los datos de aplicación que serán ser enviados a otros nodos. Utiliza la capa de red para que los datos lleguen a destino. También se encarga de interpretar y procesar los datos de aplicación recibidos.

Por ejemplo en una red de sensores inalámbricas, la capa de aplicación se encargará de sensar datos y luego utilizará las capas inferiores para enviar estos datos al sink. Por otra parte la capa de aplicación del nodo sink interpretará comandos y enviará mensajes al resto de los nodos para configurar el periodo de muestreo y setear el tiempo entre otras cosas.

1.6 Requerimientos ContikiWSN

1.6.1 Introducción

Se implementará una red de sensores inalámbricas la cual medirá las condiciones climáticas. Esta red estará constituida de varios nodos "senders" los cuales se encargarán de realizar las medidas de los fenómenos climáticos y transmitir los datos sensados a un nodo central llamado "sink" el cual tiene alimentación eléctrica permanente. Este estará conectado a un PC al cual le enviará por serial todos los datos recolectados. Por medio del PC también se podrá configurar la red.

Se utilizará el Sistema Operativo ContikiWSN a pedido del tutor. A la hora de escribir los requerimientos se tuvo en cuenta lo que este SO ofrecía.

1.6.2 Requerimientos a nivel de aplicación.

1. Mediante un Software en la PC se podrán visualizar diferentes datos en tiempo real y también su histórico. Los datos a los cuales se podrá tener acceso serán los siguientes:
    a. Datos Sensados de cada nodo
        i. Temperatura
        ii. Batería
        iii. Luminosidad
        iv. Humedad relativa
    b. Datos sobre la performance de la red
        i. ETX (Expected Transmission Count)
        ii. Paquetes perdidos
        iii. Paquetes recibidos
    c. Datos de Consumo de cada nodo
        i. RDC (Radio Duty Cycle)
        ii. Consumo promedio e instantáneo por nodo diferenciando
            1. LPM (Low Power Mode)
            2. CPU
            3. Radio Rx
            4. Radio Tx
        iii. Historial de Consumo
    d. Topología de la Red
        i. Saltos necesario de cada nodos para llegar al sink
        ii. ETX de cada salto
        iii. Cantidad de vecinos que tiene cada nodo

2. A través del Software anteriormente mencionado también se podrá ejecutar los siguientes comandos.

    a. Empezar la colección de datos: Se le comunica a los nodos senders que empiecen a sensar datos y enviarlos al sink.
    b. Parar la colección de datos: Se le comunica a los nodos senders que dejen de sensar datos.
    c. Setear Tiempo: Se configura la hora de todos los nodos de la red según la hora del sistema operativo.
    d. Configurar periodo de muestreo: Se le comunica a los nodos senders cada cuanto tienen que sensar los datos.

3. Se tendrá la opción de ingresar comandos a los nodos directamente por serial, sin necesidad de usar

el Software mencionado. Para esto los nodos tendrán una aplicación Shell que recibirá e interpretará los comandos introducidos.

1.6.3 Requerimientos a nivel de Red

1. Se utilizará los protocolos UDP/IP para la transmisión de datos entre los nodos. Se eligió IP por el fuerte desarrollo de "Internet of Things". Se quiso ser coherente con la tendencia actual. Además utilizando este protocolo se puede asegurar interoperabilidad con otras redes. También es de valor generar un antecedente con el uso de este protocolo en WSN ya que es altamente probable que cada vez se usen más este tipo de redes. Se eligió UDP porque consume menos energía y memoria que el protocolo TCP.

2. El protocolo de ruteo será RPL (Routing Protocol for Low power and Lossy Networks). Se eligió este protocolo porque es el que trae incorporado Contiki para utilizar con el stack IP. Este protocolo fue desarrollado por la IEEE ROLL "Routing Over Low power and Lossy Networks"especialmente para redes con perdidas y recursos limitados de memoria. Se encargará de alimentar las tablas de ruteo de los nodos y designar los nodos padres (preferred parents).

3. La propagación de datos del sink a los senders para configurar la red se hará a través de Trickle, que es un algoritmo de diseminación confiable basado en broadcasts para enviar los datos a todos los nodos vecinos. Se eligió este algoritmo porque disemina de manera confiable los datos utilizando pocos recursos de energía.

4. Se mantendrá sincronizada la hora de todos los nodos. De esta manera los nodos podrán etiquetar en los datos sensados la hora en que se tomó la muestra.

5. En caso que un nodo quede aislado este detectará la perdida de comunicación con el sink, guardará los datos que se muestreen a partir de ese momento en un buffer y los enviará al sink cuando se restablezca la comunicación.

1.6.4 Requerimientos a nivel de Enlace

1. Se utilizará el mecanismo ContikiMAC (ver Capítulo 3.2) ya que es el recomendado por los creadores de Contiki. De este mecanismo se pueden destacar las siguientes características:

    a. Cada nodo prenderá la radio periódicamente para saber si algún otro nodo necesita comunicarse con él.

    b. Cuando un nodo necesita enviar un unicast a un vecino intentará enviar los datos justo cuando éste prende la radio para optimizar su consumo.

    c. A nivel de capa de enlace cuando se haga un unicast se tendrá confirmación de si el vecino recibió o no los datos. Cuando los datos no logren ser recibidos por el vecino se retransmitirán hasta 5 veces.

    d. Cuando un nodo necesite enviar un broadcast prenderá la radio para transmitir durante todo un período para asegurarse que todos los nodos destinatarios prendan la radio mientras él transmite.

1.6.5 Requerimiento a nivel de topología

1. Ningún nodo podrá tener más de 10 vecinos. Este requerimiento surge por limitaciones de memoria en los nodos.

2. Se utilizará una topología de tipo malla de manera que cada nodo tenga más de un camino posible para llegar al sink. De esta manera se logrará una buena redundancia. Ya que los cambios en las condiciones de propagación, o la rotura de nodos pueden provocar la pérdida de alternativas para transmitir.

3. Se podrán agregar o quitar físicamente nodos "senders" sin necesidad de configurar o re-programar el nodo "sink". En caso de agregar un nuevo nodo, éste será configurado automáticamente por sus vecinos mediante Trickle.

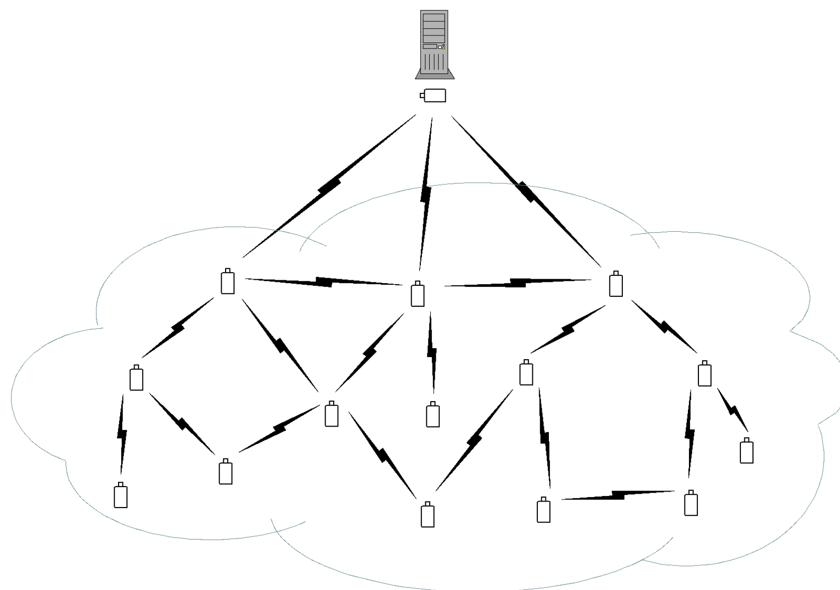


Figura 3: Topología

Capítulo 2: Sistema Operativo Contiki

2.1 Introducción

Contiki es un Sistema Operativo (SO) en tiempo real multitarea que utiliza el lenguaje de programación C. Se caracteriza por usar muy poca memoria del stack del microprocesador en comparación con la arquitectura multithreading.

En este Capítulo se presentará dicho Sistema Operativo. Primero se hará una introducción de su arquitectura y luego se explicaran los diferentes tipos de timers.

La idea de este SO es tener varios hilos de ejecución en paralelo los cuales esperan ciertas condiciones para activarse. Los hilos al activarse realizan sus tareas pertinentes y luego de terminarlas quedan nuevamente en espera hasta que otro evento los active.

Los eventos son los que se encargan de activar a los hilos. Estos se guardan en una cola circular y se procesan por orden cronológico, del más viejo al más nuevo. Cada evento generado tiene un hilo asociado para despertar, al que le puede enviar información cuando lo despierta.

Este capítulo está basado en el documento "Datalogger 2010" [11] escrito por uno de los integrantes del grupo de proyecto.

2.2 Arquitectura

2.2.1 Procesos

Los procesos pueden ser creados por el usuario para desarrollar tareas específicas. Estos se guardan en una lista encadenada, esto quiere decir que solo se necesita conocer la dirección del primero, luego el primer proceso conoce la dirección del segundo, el segundo del tercero, y así sucesivamente, de esta manera se pueden recorrer todos los procesos. Cada proceso tiene un protothread asociado, el cual se ejecuta cuando el proceso es llamado. Un protothread no es más que una función que tiene características especiales. Los estados de los procesos pueden ser desactivado, activado y llamado. Un proceso está en estado llamado cuando se está corriendo su protothread. Está activado cuando puede ser llamado (mediante eventos) y esta desactivado cuando ningún evento puede despertar al protothread.

2.2.2 Eventos

Se almacenan en una cola circular y son procesados por orden cronológico. Cada evento tiene configurado el proceso que va a despertar, al cual puede enviarle datos aparte de comunicarle cual fue el evento que lo despertó. Los eventos pueden ser generados tanto en los protothreads como en cualquier otro lado del programa como por ejemplo en las interrupciones.

2.2.3 Protothreads

Son funciones que tienen la propiedad de ser capaces de esperar por eventos. Cada proceso tiene asociado un único protothread.

Existe un evento que inicializa los procesos. Cuando este evento llama a un proceso, se corre su protothread asociado desde el comienzo. El proceso pasa del estado desactivado al llamado. Cuando el protothread necesite esperar por un evento, retorna, y su proceso asociado pasa del estado llamado al activo. Luego cuando otro evento llame al proceso, se correrá el protothread desde donde se había retornado anteriormente. En donde se chequeará si el evento recibido es el esperado, en caso negativo vuelve a retornar en el mismo punto, en caso positivo se seguirá corriendo el protothread.

2.2.4 Poll y Post Sincrónico

Los procesos también pueden requerir de atención inmediata. En este caso no se querrá esperar a que se procesen todos los eventos pendientes en la cola circular más antiguos cronológicamente. Para este caso pueden tener dos caminos distintos. El primero es hacer un post sincrónico de un evento, lo que hará que el evento se procese sin encolarlo. La otra opción es hacer un poll. Cada proceso tiene una bandera que indica si necesita ser atendido (needspoll). Antes de procesar la cola de eventos, siempre se verifica que ningún proceso necesite poll. A los procesos que necesiten poll se les enviará un evento que despertará a su protothread indicando que se lo despertó por un poll.

La diferencia entre el poll y el post sincrónico es simplemente que el primero espera que se termine el proceso que se está ejecutando para luego darle la atención al proceso que la necesite, mientras que el post sincrónico interrumpe el proceso actual, le da la atención al proceso que la requiere, y luego que éste termine, se le devuelve la atención al proceso que la cedió inicialmente.

2.3 Timers

2.3.1 Etimers

Es un módulo ya implementado por Contiki para el manejo de timers. Se basa en una lista encadenada de estructuras etimers. Estas estructuras guardan una marca de tiempo inicial, un intervalo de tiempo después del cual el timer expirará, un puntero al proceso que se va a despertar cuando el timer expire, y un puntero al próximo etimer de la lista. También se tiene un proceso etimer, cuyo protothread se encarga de procesar la lista de estructuras etimers. Este protothread lo que hace es recorrer la lista de etimers y fijarse si alguno expiró, en cuyo caso se quita la estructura de la lista y se envía un evento "PROCESS_EVENT_TIMER" al proceso que corresponde indicándole que el etimer expiró, y se le envía como dato extra la estructura de este etimer. Para despertar al proceso etimer se usan polls ya que se necesita que se despierte el protothread de inmediato. Esto se hace en la interrupción del timer, se fija según la cuenta actual (unidad de tiempo que equivale a una cierta cantidad de ticks) si algún timer expiró. En dicho caso se prende la bandera needspoll del proceso etimer para que éste se fije cuál fue el etimer que expiró y genere el evento adecuado.

2.3.2 Ctimers

Los callback timers sirven para llamar a una función con un parámetro dado cuando el timer expira. Cuando se setea el ctimer se elige el intervalo de tiempo, la función y el parámetro. El ctimer usará un etimer para que le avise cuando expira el intervalo de tiempo. En ese momento se llama a la función con el parámetro. Solo se podrán usar funciones de un solo parámetro con el ctimer.

2.3.3 Rtimers

Esta estructura representa una tarea en tiempo real y llama a una función en un momento preciso. Su funcionamiento se basa en la cuenta de ticks producidos por el oscilador de 32 kHz que tiene el Mote. Su utilización típica es la de medir intervalos de tiempo con precisión.

Capítulo 3: Capa MAC

3.1 Introducción

La capa MAC es la responsable de la gestión de acceso al medio de las capas superiores. Su función es muy importante para mantener la comunicación entre los nodos con un consumo bajo. En este capítulo primero se presentará ContikiMAC, el mecanismo para enviar paquetes entre nodos vecinos. Luego se explicará su implementación y finalmente se presentará un ejemplo de una transmisión entre dos nodos.

En Contiki esta capa se encuentra sobre la capa RDC (Radio Duty Cycling) y se encarga de evitar colisiones en el medio y de la retransmisión de los paquetes en caso de existir colisiones.

En redes de baja potencia se busca que la radio esté apagada la mayor cantidad de tiempo posible para ahorrar energía. Contiki implementa este ahorro en la capa RDC, implementando mecanismos de gestión del uso de la radio. Actualmente el mecanismo por defecto es ContikiMAC.

3.2 ContikiMAC

ContikiMAC [13] [14] [[15] [[16]]es un mecanismo de la capa RDC de Contiki. A partir de Contiki 2.5 es el mecanismo por defecto del sistema. Se basa en protocolos de ciclos de trabajo ya existentes pero añade un mecanismo de muestreo de canal de muy bajo consumo de energía. Del protocolo B-MAC [17] toma la idea básica de la escucha de baja potencia. Del protocolo X-MAC [18] utiliza la idea de un preámbulo de paquetes. Del protocolo WiseMAC [19] utiliza el mecanismo de phase-lock. Del protocolo BoX-MAC [20] toma la idea de utilizar un paquete de datos para despertar a los nodos vecinos.

3.2.1 Descripción General

La siguiente figura muestra el funcionamiento básico de ContikiMAC:

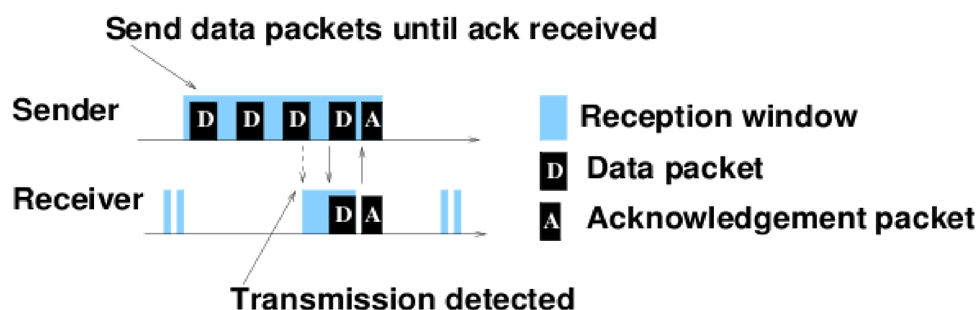


Figura 4: Escucha de bajo consumo con ContikiMAC

El nodo se despierta periódicamente para escuchar el canal de radio para saber si hay transmisiones dirigidas a él. Esto se realiza de manera de lograr un consumo eficiente de energía:

1. Un nodo enciende el radio durante 192µs para medir la señal recibida.
2. Si encuentra una transmisión de un vecino, el nodo mantiene la radio encendida.
3. Para evitar perder transmisiones, el nodo muestrea el medio dos veces cada 0,5ms.
4. Un remitente desencadena una transmisión mediante el envío de un tren de paquetes de datos, hasta que un paquete encuentra al receptor con la radio encendida.
5. Al recibir un paquete, el receptor responde con el reconocimiento y el remitente deja de transmitir la cadena de paquetes.

Las transmisiones unicast ahorran energía en ContikiMAC debido al mecanismo de sincronización phase-lock que permite que los transmisores se sincronicen con sus vecinos. El transmisor se sincroniza con la fase del receptor luego de una transmisión exitosa, como se muestra en la siguiente figura:

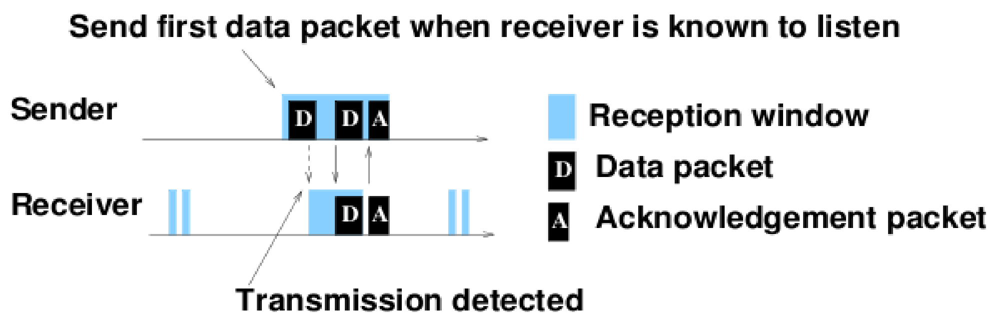


Figura 5: Transmisión unicast

En el caso de transmisiones broadcast, el emisor necesita enviar su tren de paquetes durante un período de muestreo completo para asegurar que todos los vecinos hayan escuchado la transmisión, como se muestra en la siguiente figura (las figuras 5, 6 y 7 fueron sacadas de: Adam Dunkels, Luca Mottola, Nicolas Tsiftes, Fredrik Österlind, Joakim Eriksson, and Nicolas Finne. The announcement layer: Beacon coordination for the sensornet stack. In Proceedings of EWSN 2011, Bonn, Germany, February 2011):

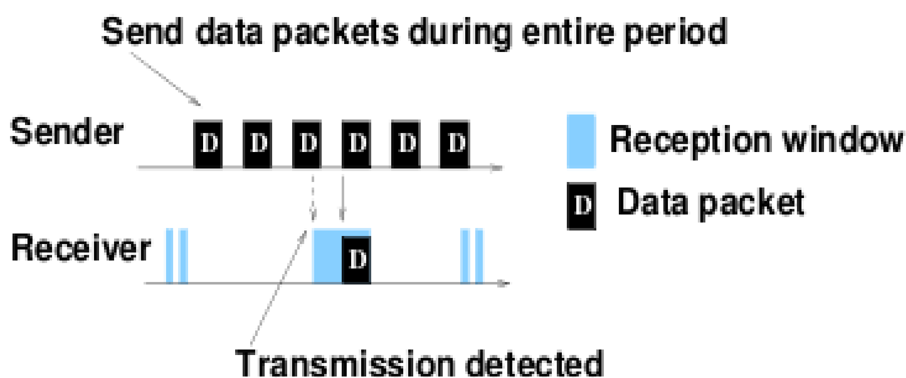


Figura 6: Transmisión broadcast

ContikiMAC tiene un consumo de energía muy bajo en comparación con otros mecanismos. A modo de ejemplo se muestra un resultado desplegado en la Wiki de Contiki. En la figura 8 se muestra un gráfico comparativo entre ContikiMAC y X-MAC, en el cual se realizó una prueba con collect de Contiki en una red de 20 nodos reportando un valor de sensor por minuto. El eje Y muestra el ciclo de trabajo de la radio

medido en porcentaje. Mayor ciclo de trabajo significa mayor consumo de energía. El eje X muestra la configuración channel check rate de los protocolos. Vemos que ContikiMAC supera significativamente a X-MAC en cada channel check rate.

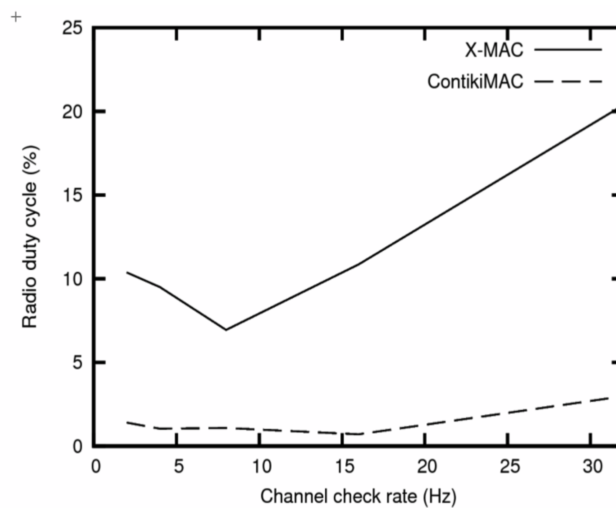


Figura 7: Comparación entre ContikiMAC y X-MAC

3.2.2 Implementación Base de ContikiMAC

A modo de resumen se presentan los diagramas de flujo de la lógica de la recepción y de la transmisión con ContikiMAC.

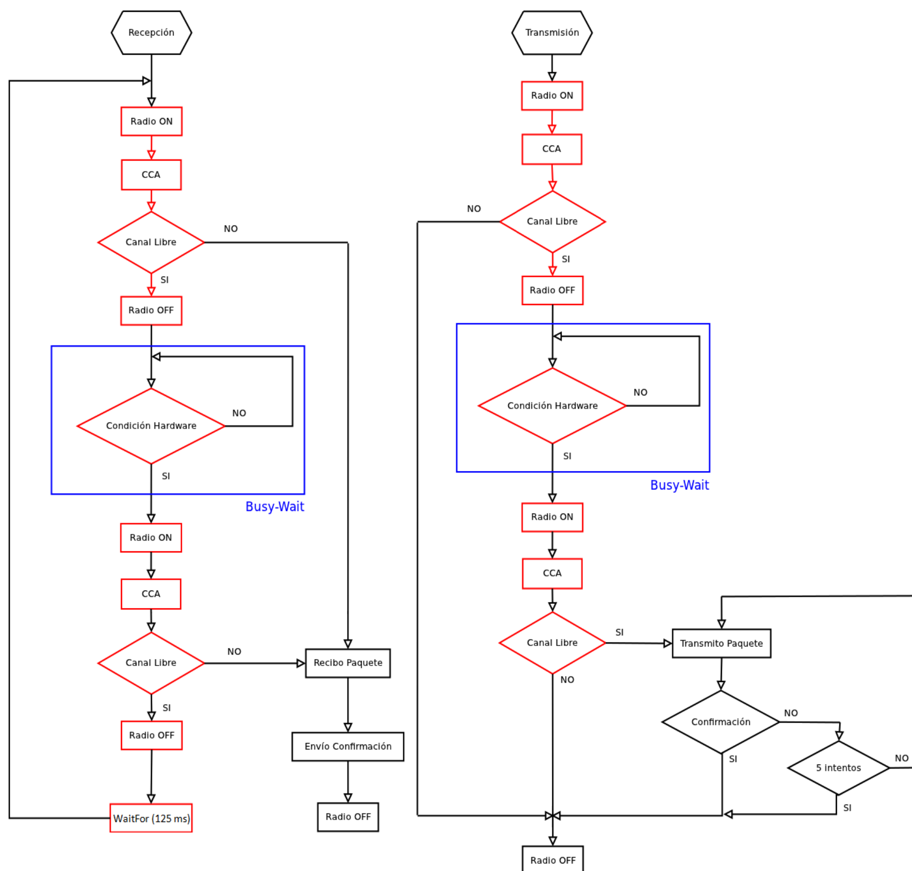


Figura 8: Diagrama de flujo de transmisión y recepción de ContikiMAC

El código de ContikiMAC está implementado en un solo archivo con su correspondiente cabecera. ContikiMAC prende la radio 2 veces periódicamente para escuchar el canal de manera de saber si existe una transmisión. Al encendido de radio para el chequeo del medio se le llama CCA (Clear Channel Assessment). La condición hardware mencionada se explicará en el párrafo 3.2.3. Las macros definidas en el código de ContikiMAC para las CCA son las siguientes:

#define CCA_COUNT_MAX	2	*ContikiMAC performs periodic channel checks. Each channel check consists of two or more CCA checks. CCA_COUNT_MAX is the number of CCAs to be done for each periodic channel check. The default is two.*/*
#define CCA_CHECK_TIME	RTIMER_ARCH_SECOND / 8192	*CCA_CHECK_TIME is the time it takes to perform a CCA check:* $1/8192 = 122us$
#define CCA_SLEEP_TIME	RTIMER_ARCH_SECOND / 2000	*CCA_SLEEP_TIME is the time between two successive CCA checks:* $1/2000 = 0{,}0005s$

La macro RTIMER_ARCH_SECOND representa 1segundo en la arquitectura en la que se esté trabajando y es la referencia temporal para definir todas las demás constantes involucradas. El número de CCA's (CCA_COUNT_MAX) definido por defecto es 2. Esto quiere decir que se harán dos escuchas consecutivas por cada ciclo. El valor CCA_CHECK_TIME (122 µs) es el tiempo de duración de una CCA y el valor CCA_SLEEP_TIME es el tiempo entre los dos CCA's. El valor CCA_SLEEP_TIME vemos que está definido de manera que la separación entre CCA sea de 0,5 ms. Este par de CCA's se realizan cada 125 ms en la configuración por defecto y están definidas con la macro CYCLE_TIME:

```
#define CYCLE_TIME (RTIMER_ARCH_SECOND /NETSTACK_RDC_CHANNEL_CHECK_RATE
```

A continuación se muestra una medida de laboratorio de la curva de consumo de corriente de 2 CCA ´s ,

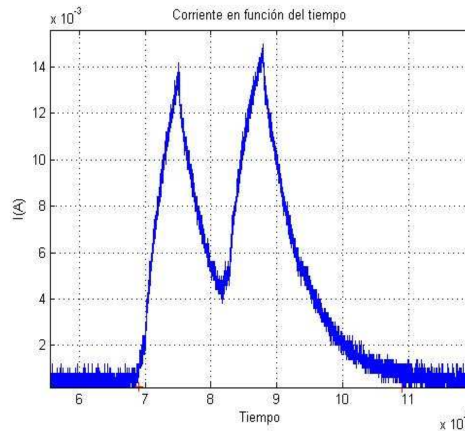


Figura 9: CCA medido con osciloscopio

A pesar de que se da la orden de apagar la radio el consumo de la misma no va a cero debido a las capacidades asociadas a la radio. Cuando se da la orden de encendido de la radio para implementar el segundo elemento de la CCA, aún no terminó de descargarse el condensador. Por este motivo no se observa en la gráfica de consumo que la radio esté apagada. Cuando detecta una transmisión, se deja encendida la radio de manera de completar la recepción y luego se realiza el procesamiento y se manda la confirmación.

También se realizan CCA's previo a la realización de una transmisión como mecanismo para evitar colisiones con las potenciales transmisiones de otros nodos.

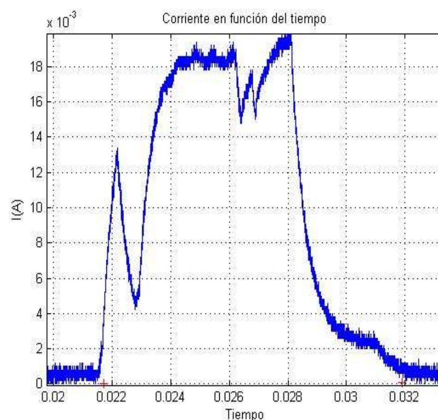


Figura 10: Detección de transmisión medida con osciloscopio

En la figura se observa que el nodo detectó actividad en la segunda CCA y en consecuencia dejó la radio encendida para poder completar la recepción y luego enviar la confirmación.

3.2.3 Implementación de las CCA's

En la implementación de las funciones intervienen los archivos contikimac.c y cc2420.c que implementa el driver de la radio del Tmote. La parte del código para dejar pasar el tiempo entre que se apaga la radio y se vuelve a encender es hecho mediante la función:

```
BUSYWAIT_UNTIL(status() & (BV(CC2420_XOSC16M_STABLE)), RTIMER_SECOND / 100);
```

Donde BV() es otra macro definida de la siguiente manera "*#define BV(b) (1<<(b))*". Mientras que BUSYWAIT_UNTIL está definida de la siguiente forma,

```
#define BUSYWAIT_UNTIL(cond, max_time)                                      \
  do {                                                                      \
    rtimer_clock_t t0;                                                      \
    t0 = RTIMER_NOW();                                                      \
    while(!(cond) && RTIMER_CLOCK_LT(RTIMER_NOW(), t0 + (max_time)));       \
  } while(0)
```

La espera del momento para dar la orden de apagar la radio se implementa con la condición lógica "status() && (BV(CC2420_XOSC16M_STABLE)" que testea el estado de la radio preguntando por un bit de un registro de la radio que indica el estado del chip de la radio.

Según la documentación del chip CC2420:

$$\text{XOSC16M_STABLE} = \begin{cases} 0: \textit{El crital oscilador de } 16 \textit{ MHz está operativo} \\ 1: \textit{El crital oscilador de } 16 \textit{ MHz no está operativo} \end{cases}$$

Por otro lado la función *status* devuelve un byte de estado que en cc2420_const.h se define como CC2420_SNOP = 0x00, que es un registro de tipo S de la radio y no tiene otro efecto que confirmar la lectura de bits de bytes estado de la radio.

La función RTIMER_CLOCK_LT(a,b) detecta el cambio de signo de una diferencia de dos números. En nuestro caso se usa en la función BUSYWAIT_UNTIL() para medir un intervalo de duración max_time, que en este caso particular vale *RTIMER_SECOND / 100,* es decir 10 ms.

La diferencia de tiempos entre la ocurrencia de la condición de estabilidad del oscilador de 16 MHz y el lapso max_time determina que en los hechos la condición sea verdadera con la estabilización de hardware. Esta es la condición hardware mencionada en la figura 9. Los valores de las macros no se utilizan para la generación de las CCA.

3.2.4 Ejemplo relevado en el laboratorio

En la Figura 12 se observa una comunicación completa entre nodos relevada en el laboratorio. En ella se observan dos tipos de CCA. Indicadas con el número 1 las CCA que corresponden al ciclo de 125ms, mientras que las indicadas con el número 2 corresponden las escuchas del canal previas a la realización de una transmisión.

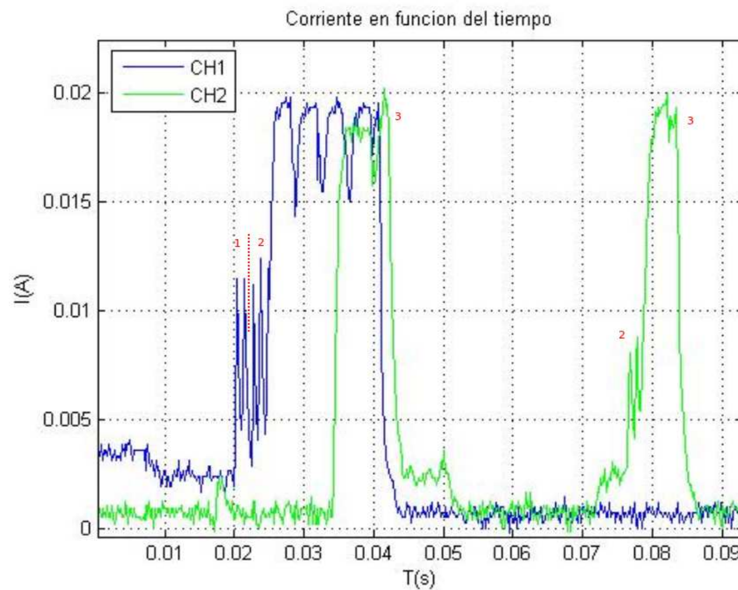


Figura 11: Transmisión entre 2 nodos medida con el osciloscopio

Inicialmente se observa que el nodo 1 (CH1) primero prende la radio para escuchar el canal como parte del funcionamiento base. Luego realiza un par de CCA como paso previo de la transmisión. Un lapso después, el nodo 2 (CH2) enciende su radio para escuchar pero como en la primer CCA encuentra una transmisión deja la radio encendida para realizar la recepción completa y luego envía la confirmación. Con el número 3 se muestra el envío de la confirmación del nodo 2 al 1.

Luego de unos mili-segundos enciende de nuevo la radio para realizar la transmisión hacia el nodo sink. Nuevamente se observa el mecanismo de escucha de canal previo a una transmisión. En este caso alcanzó con un intento para completar la transmisión exitosa debido a que era dirigida al nodo sink que está siempre escuchando el canal. Con el número 3 se ve la recepción que el nodo 2 hace de la confirmación que el nodo sink envió.

Capítulo 4: Capa de Red

4.1 Introducción

La capa de red en Contiki se implementó con el módulo uIP (micro IP) [21]. Para su implementación debió tener en cuenta las restricciones impuestas por el hardware. Para cumplir con las necesidades de conectividad de la red, teniendo en cuenta las restricciones se tuvieron que restringir las funcionalidades.

Los desarrolladores decidieron implementar las funcionalidades esenciales para garantizar la comunicación punta a punta en la red. Por otra parte, las características particulares del tráfico en las redes de sensores inalámbricas también permitieron recortar funcionalidades para poder reducir las necesidades de hardware del módulo.

En este capítulo se describirá las principales características del módulo uIP. Luego se presenta el protocolo de ruteo RPL [22] [23], pensado para operar en redes de sensores inalámbricas funcionando con IPv6. También se da a conocer un Bug que se descubrió en el código de Contiki. Finalmente se expone el algoritmo Trickle de difusión confiable de datos.

4.2 uIP – API (Application Program Interface)

uIP es el componente principal de la comunicación IP en Contiki. Si bien el módulo se desarrolla en Contiki se puede utilizar en forma independiente. Está pensado para ejecutarse en nodos con restricciones de memoria. Inicialmente se desarrolló compatible con IPv4. En 2008 se inicia el proceso de extensión a IPv6.

La API define la interfaz con la que la aplicación interactúa con el stack TCP/IP. Presenta dos opciones; una utiliza una interfaz orientada a eventos y la otra basada en protothreads.

La opción orientada a eventos consume menos memoria que la multitarea y no necesita utilizar buffers adicionales para la comunicación entre el stack y la aplicación. Además el enfoque de orientación a eventos tiene una ejecución más eficiente respecto a enfoque multitarea.

Por otro lado está la API de uIP basada en protothreads, para la cual se desarrollaron los protosockets, que son unos sockets escritos con protothreads. Esta API también provee un mecanismo para la retransmisión de datos liberando al programador de esa tarea, pero a un costo de mayor consumo de memoria.

4.2.1 API orientada a eventos

La idea central del funcionamiento es que la API llama a la aplicación correspondiente cuando ocurre un evento que la concierne. Los eventos pueden ser diversos, recepción de un dato, llegada de la confirmación de entrega de un dato, aviso de necesidad de retransmisión. También consulta periódicamente a la aplicación por si tiene un dato nuevo.

La aplicación debe proveer una función de llamada a uIP que utiliza cada vez que ocurre un evento para la aplicación correspondiente. Para reducir el consumo de memoria uIP necesita que la aplicación (el programador) participe en el proceso de retransmisión de datos. Esto debido a que para que el proceso de

retransmisión de datos sea trasparente para la aplicación sería necesario un mayor consumo de memoria para almacenar el dato hasta tanto llegue la confirmación de su recepción exitosa.

Las aplicaciones serán avisadas de la ocurrencia de un evento que las concierne llamando a la función correspondiente.

Por otra parte existen también las llamada test_function que brindan la posibilidad de preguntarle a la API por la ocurrencia de algún evento. Por ejemplo, si llegó un nuevo dato la API brinda la función uip-newdata() que es distinta de cero cuando el nodo remoto envió un dato. Los datos de la aplicación se colocan en un buffer. Luego de la invocación, los datos no son guardados por uIP, sino que se sobre escribe luego que la función retorne. Es responsabilidad de la aplicación el copiado del dato para su procesamiento posterior.

El envío de datos se realiza durante la invocación de la aplicación copiando el dato en el buffer antes del retorno a uIP. uIP ajusta la longitud del dato enviado por la aplicación de acuerdo al espacio disponible en el buffer y el espacio de la ventana TCP notificado por el receptor.

Las aplicaciones solo pueden enviar un paquete simultáneamente y deben esperar la confirmación del receptor para enviar el siguiente paquete.

Cuando una conexión está libre y como parte de su funcionamiento periódico uIP invoca la aplicación con la bandera de poll en 1. La aplicación verifica la bandera para saber si el llamado se debió al funcionamiento periódico y si ocurrió un evento en particular.

Las consultas periódicas cumplen dos funciones. La primera es informar periódicamente que la conexión está libre. Esto le permite a la aplicación cerrar las conexiones que están libres por mucho tiempo. En segundo lugar informa a la aplicación que puede enviar un nuevo dato. Una aplicación solo puede enviar un dato cuando es invocada por uIP.

4.2.2 Loop de Control Principal

El stack uIP puede funcionar tanto como una tarea en un sistema multitarea, o como el programa principal en un sistema single-tasking. En ambos casos, el bucle de control principal realiza dos tareas; verificar si existen paquetes y verificar si venció algún timer.

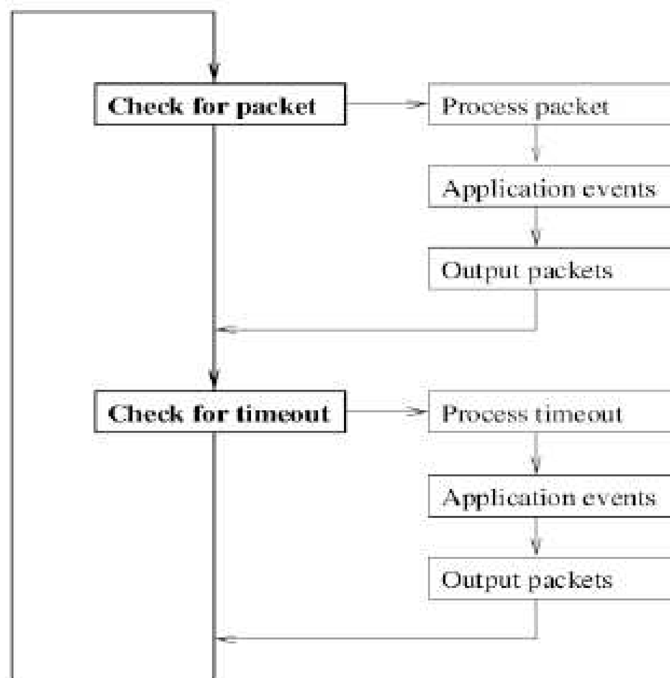


Figura 12: Bucle de control principal

4.2.3 Retransmisión de datos

Las retransmisiones son disparadas por un temporizador periódico de TCP. Cada vez que este temporizador periódico se invoca, el temporizador de retransmisión para cada conexión se decrementa. Si el contador llega a cero, se debe realizar una retransmisión. Como uIP no hace un seguimiento del contenido de los paquetes después de haber sido enviado por el controlador de dispositivo, requiere que la aplicación tome parte activa en la realización de la retransmisión. Cuando uIP decide que un segmento debe ser retransmitido, la función de aplicación se llama con el uip_rexmit() lo que indica que es necesaria una retransmisión.

La aplicación debe comprobar la bandera uip_rexmit() y producir los mismos datos que se enviaron previamente. Desde el punto de vista de la aplicación, la realización de una retransmisión no es diferente de cómo los datos se enviaron originalmente.

Por consiguiente, la aplicación puede ser escrita de tal manera que el mismo código se utilice tanto para el envío de datos como para la retransmisión de datos. Además, es importante señalar que a pesar de que la operación de retransmisión real se lleva a cabo mediante la aplicación, es responsabilidad del stack saber cuando debe hacerse. Así, la complejidad de la aplicación no aumenta necesariamente, por participar activamente en las retransmisiones.

4.2.4 Cierre de Conexiones

La aplicación cierra la conexión actual llamando a uip_close() cuando es llamada desde uIP. Esto hará que la conexión se cierre limpiamente. Para indicar un error fatal, la aplicación que desee abortar la conexión lo hace mediante una llamada a la función uip_abort(). Si la conexión ha sido cerrada por el

extremo remoto, la función de prueba uip_closed() es verdadera. La aplicación puede entonces realizar cualquier limpieza necesaria.

4.2.5 Reporte de errores

Hay dos errores fatales que le puede ocurrir a una conexión. Que la conexión fue abortada por el host remoto, o que la conexión no haya recibido una confirmación y ocurra un time-out. uIP informa de esto llamando a la función de aplicación. La aplicación puede utilizar las dos funciones de prueba uip_aborted() y uip_timedout() para disernir las condiciones de error.

4.2.6 Escucha de puertos

uIP mantiene una lista de los puertos TCP que está escuchando. Para abrir un nuevo puerto de escucha se utiliza la función *uip_listen(), d*e modo que cuando llega una petición de conexión dirigida a un puerto en particular uIP crea una nueva conexión y llama a la función de la aplicación correspondiente. La función de prueba *uip_connected()* devuelve verdadero si la aplicación fue invocada por una nueva conexión creada. En ese momento la aplicación puede verificar el campo *lport* en la estructura *uip_conn* para comprobar a qué puerto se realizó la nueva conexión. En la figura 14 se ve la estructura completa de *uip_conn* que implementa la conexión en uIP.

```
1274  /**
1284  struct uip_conn {
1285    uip_ipaddr_t ripaddr;   /**< The IP address of the remote host. */
1286
1287    u16_t lport;        /**< The local TCP port, in network byte order. */
1288    u16_t rport;        /**< The local remote TCP port, in network byte
1289              order. */
1290
1291    u8_t rcv_nxt[4];    /**< The sequence number that we expect to
1292              receive next. */
1293    u8_t snd_nxt[4];    /**< The sequence number that was last sent by
1294                    us. */
1295    u16_t len;          /**< Length of the data that was previously sent. */
1296    u16_t mss;          /**< Current maximum segment size for the
1297              connection. */
1298    u16_t initialmss;   /**< Initial maximum segment size for the
1299              connection. */
1300    u8_t sa;            /**< Retransmission time-out calculation state
1301              variable. */
1302    u8_t sv;            /**< Retransmission time-out calculation state
1303              variable. */
1304    u8_t rto;           /**< Retransmission time-out. */
1305    u8_t tcpstateflags; /**< TCP state and flags. */
1306    u8_t timer;         /**< The retransmission timer. */
1307    u8_t nrtx;          /**< The number of retransmissions for the last
1308              segment sent. */
1309
1310    /** The application state. */
1311    uip_tcp_appstate_t appstate;
1312  };
```

Figura 13: Definición de una conexión en uIP en el código

4.2.7 Apertura de conexiones

Las nuevas conexiones se pueden abrir desde uIP con la función uip_connect(). Esta función abre una nueva conexión y establece un indicador del estado de la conexión. El uip_connect() devuelve un puntero a la estructura uip_conn para la nueva conexión. Si no hay ranuras de conexión libre, la función

devuelve NULL.

4.3 Protosockets

La biblioteca protosocket proporciona una interfaz para el stack uIP que es similar a la interfaz de sockets BSD tradicional. A diferencia de los programas escritos para la interfaz uIP orientada a eventos, los programas escritos con la biblioteca protosocket se ejecutan de manera secuencial y no tienen que ser implementados como máquinas de estado de forma explícita. Los protosockets sólo funcionan con conexiones TCP.

La biblioteca protosocket utiliza protothreads para proporcionar un flujo de control secuencial. Esto hace que los protosockets tengan menos requerimientos en términos de consumo de memoria, pero también significa que los protosockets heredan las limitaciones funcionales de los protothreads. Cada protosocket sólo vive dentro de un bloque de función único. Es necesario tener en cuenta que debido a que los protosockets utilizan la biblioteca de los protothreads, las variables locales no siempre se guardan en una llamada a una función protosocket. Por tanto se aconseja que las variables locales se utilicen con extremo cuidado.

La biblioteca protosocket proporciona funciones para el envío de datos sin tener que hacer frente a las retransmisiones y reconocimientos, así como funciones para la lectura de datos sin tener que hacer frente a datos fragmentados en más de un segmento TCP.

Debido a que cada protosocket se ejecuta como un protothread, el protosocket tiene que ser iniciado con una llamada a PSOCK_BEGIN() al comienzo de la función en la que se utilice el protosocket. Del mismo modo, el protosocket puede ser cerrado llamando a PSOCK_EXIT().

Estas macros están definidas en el archivo psock.h

```
#define PSOCK_BEGIN(psock) PT_BEGIN(&((psock)->pt))
#define PSOCK_EXIT(psock) PT_EXIT(&((psock)->pt))
```

4.4 Implementación de los protocolos en uIP

Para lograr optimizar el consumo de memoria y el tamaño que ocupa el código, la implementación de uIP se limita a implementar las funcionalidades indispensables para cumplir con los estándares y asegurar la interoperabilidad. Muchos mecanismos usualmente implementados en los stacks TCP/IP se diseñaron para mejorar la performance del protocolo. En estos caso la mejora se logra a expensas de mayores requerimientos de memoria. Es por esto que no están implementados en uIP.

4.4.1 Re-ensamblado de los fragmentos IP

El re-ensamblado IP se implementó con un buffer de re-ensamblado independiente al de recepción. Cuando todos los fragmentos se re-ensamblaron, el paquete se entrega a la capa de transporte. Si luego de transcurrido un lapso no se recibieron todos los fragmentos, éstos son desechados. Este mecanismo no consume mucha memoria debido a que la implementación del mapa de bit utilizado para administrar los fragmentos está hecha con 8 bytes.

Este buffer está separado del de recepción para evitar que los fragmentos de un paquete sean sobre escritos por otro paquete. Por ser único, uIP solo es capaz se re-ensamblar un paquete a la vez.

4.4.2 Manejo de TCP

Las implementaciones actuales de TCP disponen de mecanismos para mejorar el througput. Muchos de estos mecanismos no son necesarios en redes que no tiene la necesidad de transportar gran cantidad de información. El compromiso entre uso eficiente de memoria y gran capacidad de tráfico se resuelve en uIP tomando la opción de privilegiar el uso eficiente de la memoria.

En uIP, TCP es manejado por los paquetes entrantes y por el procesamiento periódico. Los paquetes entrantes son revisados por TCP y si contienen paquetes para alguna aplicación se la llama invocando la función correspondiente. Si el paquete entrante consiste en la confirmación un paquete enviado, TCP actualiza el estado de la conexión y notifica a la aplicación, habilitándola a enviar un nuevo dato.

TCP permite que una conexión escuche si llega una solicitud de conexión nueva. En uIP una conexión que escucha se identifica con un puerto de 16 bits y las solicitudes de conexiones entrantes se comparan con una lista de potenciales solicitudes. Esta lista es dinámica y puede modificarse a través de una aplicación.

4.4.3 Ventana Deslizante

Este mecanismo no está implementado en uIP debido a que no es requerido por las especificaciones de TCP y consume muchos recursos. Para su implementación necesita realizar muchas cuentas con los números de secuencia usados en TCP que son de 32 bits. Este consumo es muy grande en sistemas de 8 y 16 bits como en los que está pensado usar ese stack. Existe un inconveniente adicional debido a que uIP no guarda los paquetes enviados y el mecanismo de Ventana Deslizante requiere almacenar tantos paquetes como el tamaño de ventana. En lugar de esto uIP permite tener un solo paquete no reconocido a la vez. La falta de este mecanismo no afecta la interoperabilidad ni el cumplimiento de los estándares.

4.4.4 Control de Flujo

La aplicación no puede enviar más datos de los que puede procesar el receptor. Antes de enviar un dato el transmisor debe verificar cuanta información se puede enviar.

4.4.5 Control de Congestión

Como uIP solo envía un paquete por conexión a la vez no es posible limitar más aún, por lo tanto no es necesario disponer de un mecanismo de control de congestión.

4.4.6 Datos Urgentes

El mecanismo de marcar datos como urgentes permite que las aplicaciones intercambien información referida a los datos. Usualmente se implementa con un mecanismo de notificaciones asíncronas que vuelve el

código más complejo. En la API de uIP ya se utiliza un mecanismo asíncrono orientado a eventos por lo que la funcionalidad no vuelve el código más complejo.

4.4.7 Cálculo de sumas de comprobación

En TCP/IP se implementan sumas de comprobación a partir de los datos y los encabezados de los paquetes. Como estos cálculos se realizan con todos los bytes de cada paquete enviado y recibido es importante que se hagan de manera eficiente. El stack uIP provee funciones genéricas para el cálculo de las sumas de comprobación. Estas funciones son uip_ipchksum() y uip_tcpchksum() y es posible sustituirlas por implementaciones particulares que se ajusten a la arquitectura en la que va a ejecutarse uIP. Incluso se podría realizar una implementación en assembler en lugar de C para lograr un resultado óptimo.

4.5 IPv6 en Contiki

En Contiki se realizó una implementación de dos drafts del grupo de trabajo de la IETF “IPv6 over Low power WPAN (6lowpan)” que trabaja en el proceso de estandarización de IPv6 para redes descriptas en el standard IEEE 802.15.4.

En el código de Contiki se aclara cuales son los estándares implementados en el código. Los archivos sicslowpan.{c,h} realizan la implementación del trabajo del grupo 6lowpan. La especificación implementada en la versión 2.5 de Contiki es la detallada en el RFC 4944 “Transmission of IPv6 Packets over IEEE 802.15.4 Networks”. El dinamismo del proceso de estandarización provoca que en la actualidad este RFC ya tenga un sucesor que actualiza los temas y deja obsoleto el RFC 4944. Se trata del RFC 6282 “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks” [24].

Algunas de las funcionalidades de IPv6 no fueron implementadas en base a definiciones específicas para redes de sensores inalámbricas, sino que los desarrolladores se refirieron a información más general. Es el caso del descubrimiento de la red que en Contiki fue implementado tomando como base el RFC 4861 “Neighbor Discovery for IP version 6” en lugar de referirse al borrador draft-ietf-6lowpan-nd “*Neighbor Discovery Optimization for Low Power and Lossy Networks*” cuya última versión liberada es la 18 y cuyo estado es “*Publication Requested*”.

4.6 RPL - Ruteo en WSNs

El protocolo de ruteo RPL es desarrollado por el grupo de la IEEE ROLL “Routing Over Low power and Lossy networks” [25]. El grupo trabaja considerado las redes “Low power and Lossy networks” (LLNs) de las cuales las redes de sensores inalámbricos son un caso particular.

En la actualidad el proyecto tiene concluidos varios RFC. Los cuatro primeros en ser concluidos fueron los dedicados a establecer los requerimientos de ruteo en distintos escenarios de aplicación: urbano (RFC 5548) [26], industrial (RFC 5673) [27], doméstico (RFC 5826) [28] y automatización de edificios (RFC 5867) [29]. El siguiente RFC en aparecer fue el referido al algoritmo de difusión confiable de datos Trickle (RFC 6206) [30], que se describirá más adelante.

El documento referido al protocolo propiamente dicho es el borrador draft-ietf-roll-rpl, la presente introducción está basada en el documento RFC 6206, The Trickle Algorithm [30].

4.6.1 Introducción al protocolo RPL

RPL y teoría de grafos

Un grafo es una representación abstracta de un conjunto de objetos donde algunos pares de objetos están conectados por vínculos. Los objetos interconectados son representados por las abstracciones matemáticas llamadas vértices, y los vínculos que los conectan se llaman bordes. Típicamente, un grafo se representa de forma esquemática como sistema de puntos para los vértices, unidos por las líneas o las curvas para los bordes.

Algunas definiciones básicas

- **DAG - Directed Acyclic Graph**

  Es un grafo orientado que tiene la propiedad que todos los caminos están orientados de tal modo que no existen ciclos. Todos los vértices están contenidos en caminos orientados hacia y terminando en uno o varios nodos de raíz.

- **DAG root**

  Es un nodo del DAG que no tiene bordes salientes. Los DAG deben tener al menos un nodo root.

- **Destination Oriented DAG (DODAG)**

  Es un DAG con un único nodo root.

- **Sub-DODAG**

  El sub-DODAG de un nodo es el conjunto de nodos cuyo camino a la raíz pasa por él. Los nodos que pertenecen al sub-DODAG de un nodo tienen un rank superior al del nodo.

- **Mensajes de Control RPL**

  - DIO (DODAG Information Object): son enviados por los nodos RPL para publicar información sobre el DODAG. Sirve para que un nodo conozca a sus vecinos y sus respectivos ranks.

  - DAO (Destination Advertisement Object): son usados para propagar información de direccionamiento aguas arriba, alimentando las tablas de ruteo de los nodos padres.

  - DIS (DODAG Information Solicitation Message): se usan para descubrir DODAGs en el vecindario y solicitar mensajes DIOs de los nodos RPL.

4.6.2 Rank

El rank de un nodo representa que tan bueno es el camino del nodo al sink. Este parámetro depende de todos los links de capa 2 que se usan para llegar al sink. Cuanto más bajo el valor del Rank, significa que mejor es el camino al Sink. El nodo con el Rank más bajo es el propio Sink cuyo Rank vale 256 (unidad mínima indivisible para medir ranks llamada MinHopRankIncrease). .

El best parent (o preferred parent) de un nodo es el nodo vecino con el cual se consigue el mejor camino para llegar al Sink. De esta manera se minimiza el rank del nodo.

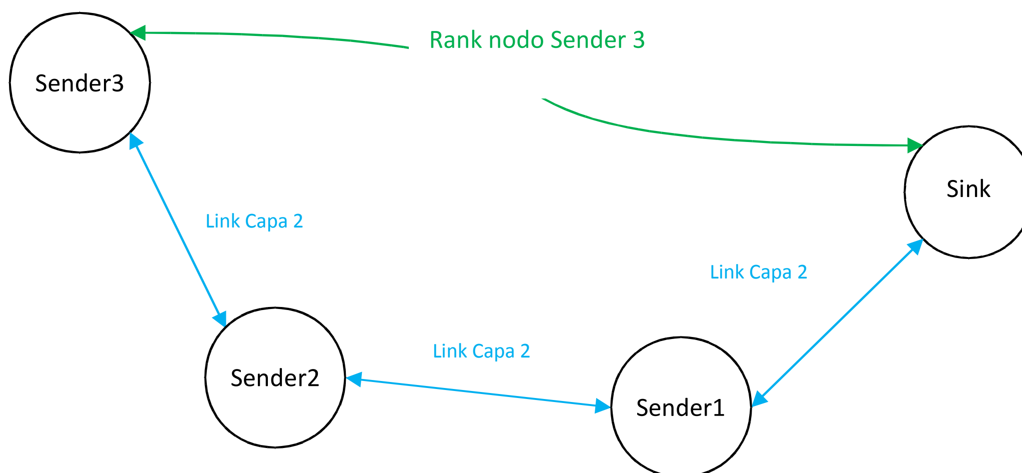


Figura 14: Ejemplo de camino al Sink

El rank del nodo Sender3 depende de la calidad de los 3 links de capa 2 intermedios.

Para medir la calidad del link con un nodo vecino se utiliza la métrica ETX. Está expresa cuantas transmisiones se estiman necesarias para que un paquete llegue sin errores al destino.

Cuando un mensaje es enviado a un vecino, se espera un reconocimiento. Si este no es recibido entonces se reenvía el mensaje. Se pueden generar hasta 5 reintentos, luego se da el mensaje por perdido.

El valor de ETX se calcula cada vez que se le envía un mensaje al vecino utilizando la siguiente fórmula:

ETX = 0,9 * ETX_Almacenado + 0,1 * ETX_Ultimo_Paquete

Donde ETX_Almacenado refiere al valor de ETX antes de utilizar la fórmula. O sea, el valor de ETX válido hasta el momento, el cual expresa cuantas transmisiones se estiman necesarias para que el vecino reciba exitosamente un mensaje.

ETX_Ultimo_Paquete refiere a cuantas transmisiones fueron necesarias para que el último paquete que se envió fuera reconocido por el vecino. Tener en cuenta que si el paquete no fue reconocido este valor

es definido como 15. Por lo tanto los valores posibles para la variable ETX_Ultimo_Paquete son 1, 2, 3, 4, 5 o 15. Teniendo esto en cuenta y observando la formula se puede concluir que el valor máximo que puede tomar la variable ETX es 15 y el valor mínimo es 1.

Cuando se descubre un vecino con el primer paquete que se envíe se inicializará ETX= ETX_Ultimo_Paquete ya que no hay ningún valor para ETX_Almacenado. A partir de este momento se utilizará la formula general.

El rank de un nodo se calcula tomando como base el rank del best parent y sumándole la calidad del link de capa 2 con este.

Rank de un nodo = Rank del preferred parent + [ETX con su preferred parent] * 256

Donde [ ] significa parte entera del valor que encierra.
Si el link de capa 2 con el preferred parent es perfecto, o sea siempre se reciben los reconocimientos en el primer intento, entonces el ETX será 1. En este caso el rank del nodo se incrementará en 256 comparado con el del preferred parent.

Como ya se mencionó, el best parent de un nodo es el nodo vecino con el cual se consigue el mejor rank para el nodo. Esto no coincide necesariamente con el nodo vecino de mejor Rank.
Si en el ejemplo de la figura 20 el nodo Sender2 es capaz de comunicarse directamente con los nodos Sender1 y Sink, obviamente el nodo con mejor Rank va a ser el Sink, pero imaginemos que:

- El ETX del nodo Sender 2 con el nodo Sink es de 4,3
- El ETX del nodo Sender 2 con el nodo Sender 1 es 1.
- El ETX del nodo Sender 1 con el nodo Sink es 1.

Entonces:

Rank del sink = 256

Rank Nodo Sender 1 = Rank del Sink + [ETX con el Sink] * 256
= 256 + [1] * 256
=512

Rank Nodo Sender 2 tomando como preferred parent al nodo Sender 1

Rank Nodo Sender 2 = Rank del nodo Sender 1 + [ETX con el nodo Sender 1] * 256
= 512 + [1] *256
= 768

Rank Nodo Sender 2 tomando como preferred parent al Sink

Rank Nodo Sender 2 = Rank del Sink + [ETX con el Sink] * 256
= 256 + [4,3] *256
= 1280

De los cálculos se puede observar que el Rank se optimiza tomando como preferred parent al nodo Sender1 y llegando al Sink con 2 saltos sin perder mensajes.

Si un nodo debe enviar un mensaje al Sink, éste será encaminado a través de su preferred parent. Por lo tanto para llegar al sink, un mensaje se encaminará por una sucesión de preferred parents hasta llegar a éste.

4.6.3 Procesamiento de Mensajes RPL

Mensajes DIS

Cada nodo chequea periódicamente si conoce un DODAG, en caso negativo el nodo enviará un broadcast de mensaje DIS. Lo que hace este mensaje es solicitar información del DODAG a los nodos que lo reciban.

Todo nodo que recibe un mensaje DIS, envían a la brevedad un broadcast de DIO. Notar que antes de enviar el mensaje DIO se agrega una componente de tiempo aleatoria para que no haya colisiones.

Mensajes DIO

Los broadcast de mensajes DIO se envían periódicamente. Publican la información sobre el DODAG. Se utiliza una lógica muy parecida a la de Trickle para minimizar la cantidad de mensajes enviados en régimen.

La idea es que cada vez que se dispare un evento que indique que la red debe actualizarse se setea el timer que genera los mensajes DIO con el intervalo mínimo I=Imin. Cada vez que el timer expira se envía un broadcast de mensaje DIO y se duplica el intervalo I. El intervalo I no puede superar a un valor máximo predefinido Imax.

Para introducir una componente aleatoria y evitar colisiones lo que se hace es, si el intervalo es I, setear el timer con un valor aleatorio entre [I/2, I].

El intervalo mínimo Imin= 4 segundos y el intervalo máximo Imax= 17 minutos.

Los siguientes eventos generarán que el intervalo se restablezca I=Imin.

- El nodo descubre un DODAG.
- El nodo es seteado como root.
- El nodo cambia de preferrent_parent.
- Cambia el rank del nodo.
- Se recibe un mensaje DIS.
- Se recibe un mensaje de un nodo con rank infinito.
- Se recibe un mensaje DIO con una versión distinta de DODAG
- Se ejecuta global rpl_repair
- Se ejecuta local rpl_repair

Cuando un nodo recibe un mensaje DIO lo procesa de la siguiente manera:

1. Agrega a la tabla de vecinos al remitente (si ya no estaba incluido)
2. Si no conocía al DODAG:
   a. Agrega como preferred_parent al remitente
   b. Si el nodo no tiene dirección IP se la asigna juntando el prefijo con la dirección mac.
   c. Se configura el timer para enviar a la brevedad un broadcast DIO con la información del nuevo DODAG.
   d. Se envía un mensaje DAO al remitente del mensaje DIO avisándole que es el

preferred_parent del nodo.

3. En caso que la condición 2 sea falsa (ya conocía el DODAG) se hace lo siguiente
    a. Se agrega al remitente como parent (salvo que ya estuviese incluido)
    b. Se compara el remitente con el resto de los parents y se chequea si es el preferred_parent.
    c. En caso afirmativo el nodo le envía un mensaje DAO al remitente indicándole que es su preferred_parent.

Mensajes DAO

Los mensajes DAO son disparados en las siguientes situaciones y siempre se envían al preferred_parent:

1. El nodo cambia de preferred parent.
    • Se envía un mensaje al preferred parent viejo con la indicación de eliminar al nodo de la tabla de ruteo.
    • Se envía un mensaje al preferred parent nuevo con la indicación de agregar al nodo a la tabla de ruteo.
2. El preferred parent tiene un rank demasiado alto.
    • Se envía un mensaje al preferred parent con la indicación de eliminar al nodo de la tabla de ruteo.
3. Cuando se recibe un mensaje DIO del preferred parent (Notar que los mensajes DIO son periódicos).
    • Se envía un mensaje al preferred parent. Como este ya tiene agregado al nodo a la tabla de ruteo lo que se hace es actualizar el lifetime de la entrada.
4. Cuando un nodo se une a una red nueva (DODAG).
    • Se envía un mensaje al preferred parent nuevo con la indicación de agregar al nodo a la tabla de ruteo.
5. Cuando se ejecuta rpl global_repair.
    • Se envía un mensaje al preferred parent que envío la versión de DAG nueva con la indicación de agregar al nodo a la tabla de ruteo.
6. Cuando se recibe un mensaje DAO de un nodo hijo.
    • Se re-envía el mensaje al preferred parent.

Cuando se recibe un mensaje DAO se procesa de la siguiente manera:

1. Si el mensaje indica que el nodo dejo de ser el preferred_parent del remitente. Quita a este de la tabla de ruteo y a todas las entradas que lo tengan como next-hop.
2. Si el mensaje indica que el nodo es el prefered parent del remitente.
    a. Agrega al nodo a la tabla de ruteo. Si la entrada ya se encontraba renueva el tiempo de expiración de la ruta.
    b. Se reenvía este mensaje DAO al preferred_parent.

Los mensajes DAO son enviados de un nodo a su preferred parents y éste lo reenvía a su preferred parent y así sucesivamente. El mensaje pasa de preferred parent en preferred parent hasta llega al sink.

De esta manera es que el sink recibe todos los mensajes DAO y por lo tanto sabe llegar a todos los nodos de la red.

Ejemplo de Generación de Rutas

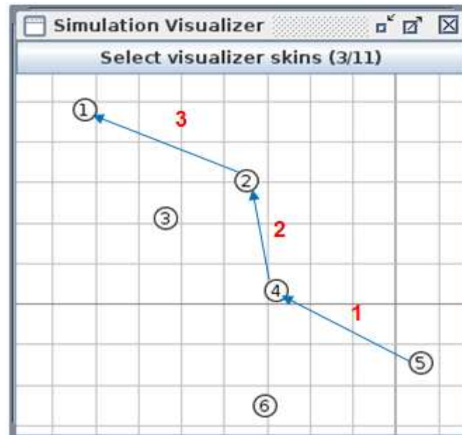


Figura 15: Ejemplo de generación de rutas

En la red de la figura 21 el nodo 1 (el sink) es el preferred parent del nodo 2, el nodo 2 es el preferred parent del nodo 4 y el nodo 4 es el preferred parent del nodo 5.

1. El nodo 5 envía el mensaje DAO a su preferred parent que es el nodo 4. Éste aprende a llegar al nodo 5
2. El nodo 4 reenvía el mensaje DAO recibido a su preferred parent que es el nodo 2. De esta manera el nodo 2 aprende que para llegar al nodo 5 lo tiene que hacer por medio del nodo 4.
3. El nodo 2 reenvía el mensaje DAO al sink. De esta manera el sink sabe que si tiene que enviar un mensaje al nodo 5 el next-hop será el nodo 2.

4.6.4 Envío de datos al sink

Para enviar mensajes al sink no se precisan entradas en la tabla de ruteo ya que la ruta por defecto es siempre el preferred parent. Por lo tanto si el mensaje se va pasando de preferred parent a preferred parent va a llegar al sink por el camino óptimo.

4.6.5 Ruteo

En la tabla de ruteo un nodo tendrá las rutas hacia los nodos de su Sub-DODAG. Por lo tanto el nodo raíz tendrá en su tabla de ruteo todos los nodos del DODAG.

Cuando un nodo tiene que enviar un mensaje a otro nodo el cual no aparece en su tabla de ruteo, el mensaje es enviado por la ruta por defecto cuyo next-hop es el preferred parent. Si este otro nodo tampoco lo tiene en su tabla de ruteo, lo reenviará a su preferred parent. Este proceso se puede repetir hasta llegar al sink, que conoce como llegar a todos los nodos del DODAG.

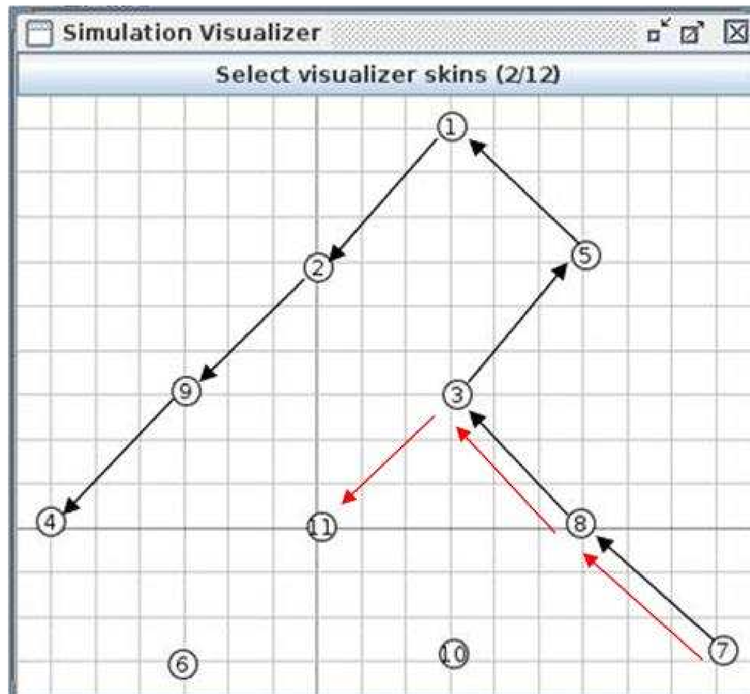


Figura 16: Ejemplo de ruteo

En la imagen se ve la ruta para dos mensajes distintos que son enviados por el nodo 7.

El primer mensaje es enviado al nodo 4. Los nodos 8,3,5 no tienen en su tabla de ruteo al nodo 4 por lo tanto envían al preferred parent el mensaje. Los nodos 9, 2 y el root si tienen como llegar al nodo 4 en su tabla de ruteo y es reenviado al next-hop hasta llegar al destino.

El segundo mensaje es enviado al nodo 11. El nodo 8 no tiene en su tabla de ruteo al nodo 11 por lo tanto lo reenvía al nodo 3, este nodo si tiene en su tabla de ruteo al nodo 11. Por lo tanto dirige el mensaje directamente al destino.

4.6.6 Rpl global repair

El sink puede intentar reparar la red cuando hay problemas. Para esto lo que hace es reinicializar el protocolo de ruteo desde cero. Cuando se presiona el botón de usuario del nodo sink este incrementa el número de versión del DODAG. Los mensajes DIO contienen el número de versión del DODAG. Cuando un nodo recibe un mensaje DIO con un número mayor de version de DODAG empieza a ejecutar la rutina global_repair en donde remueve a todos sus parents rpl, setea al nodo remitente como preferred parent (es el único parent que conoce hasta el momento) y se agenda enviar un mensaje DIO a la brevedad.

4.6.7 Rpl local repair

Cuando un nodo pierde a todos sus parents o no tienen ningún parent con un rango aceptable ejecuta la rutina rpl_local_repair. Esta rutina le setea rank infinito al nodo, le borra todos los parents y agenda un mensaje DIO para que se envíe a la brevedad.

4.7 Bug RPL

4.7.1 Introducción

Se descubrió un Bug en el sistema operativo Contiki en el modulo de RPL. Es importante destacar que el bug no afectaría a nuestra aplicación ContikiWSN. El bug puede producir problemas con el ruteo aguas abajo y nosotros no usamos esta funcionalidad para nuestra aplicación.

De todas maneras nos pareció interesante estudiar el bug en profundidad y descubrir su alcance y posible solución. Además también nos parece importante contribuir al desarrollo del sistema operativo Contiki reportando la falla.

4.7.2 Alcance del Bug

El bug afecta la generación de nuevas entradas en las tablas de ruteo de los nodos. Por lo tanto pueden ocurrir problema en el ruteo aguas abajo. Los nodo en ciertas ocasiones no saben llegar a todo el conjunto del Sub-DODAG.

Los nodos no envían mensajes DAO periódicos. Cuando se cambia la estructura del DODAG, solo los nodos que cambian de preferred parent enviaran mensajes DAO. Por lo tanto no se podrá llegar correctamente a los nodos que no enviaron mensajes DAO.

4.7.3 Reproducción del Bug

Se logró reproducir el bug en el simulador COOJA. Partimos de la siguiente topología:

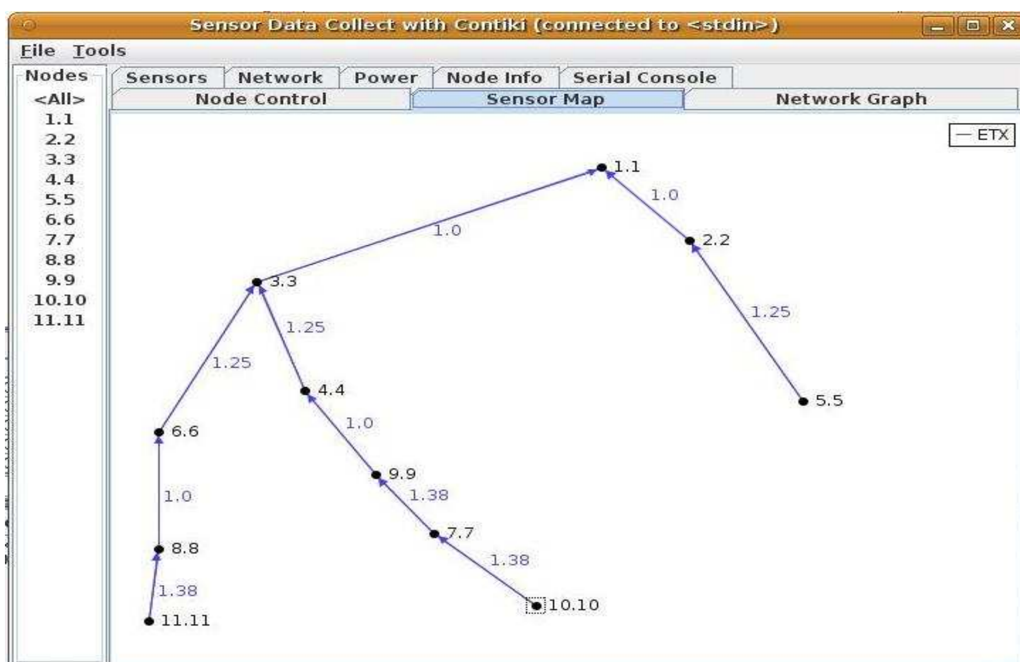


Figura 17: Topología de la simulación

En la figura 23 se ve como cada nodo apunta con una flecha a su preferred parent.

El nodo 3 debe tener en su tabla de ruteo como llegar a todo su Sub-DODAG. Este es integrado por los nodos {11, 8, 6, 10, 7, 9, 4}. Se corre el comando netprint en el nodo 3 y se confirma lo dicho.

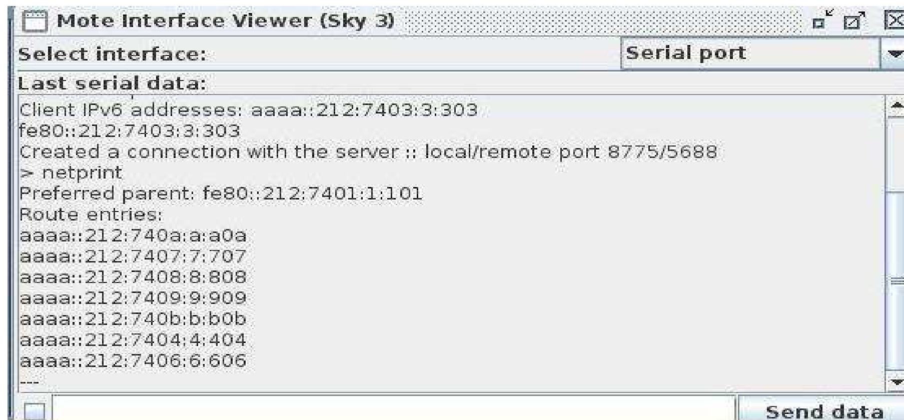


Figura 18: Tabla de ruteo del nodo 3

En la figura se muestran las entradas de la tabla de ruteo del nodo 3.

Se separa el nodo 3 de la red de manera que quede aislado. La topología queda de la siguiente manera.

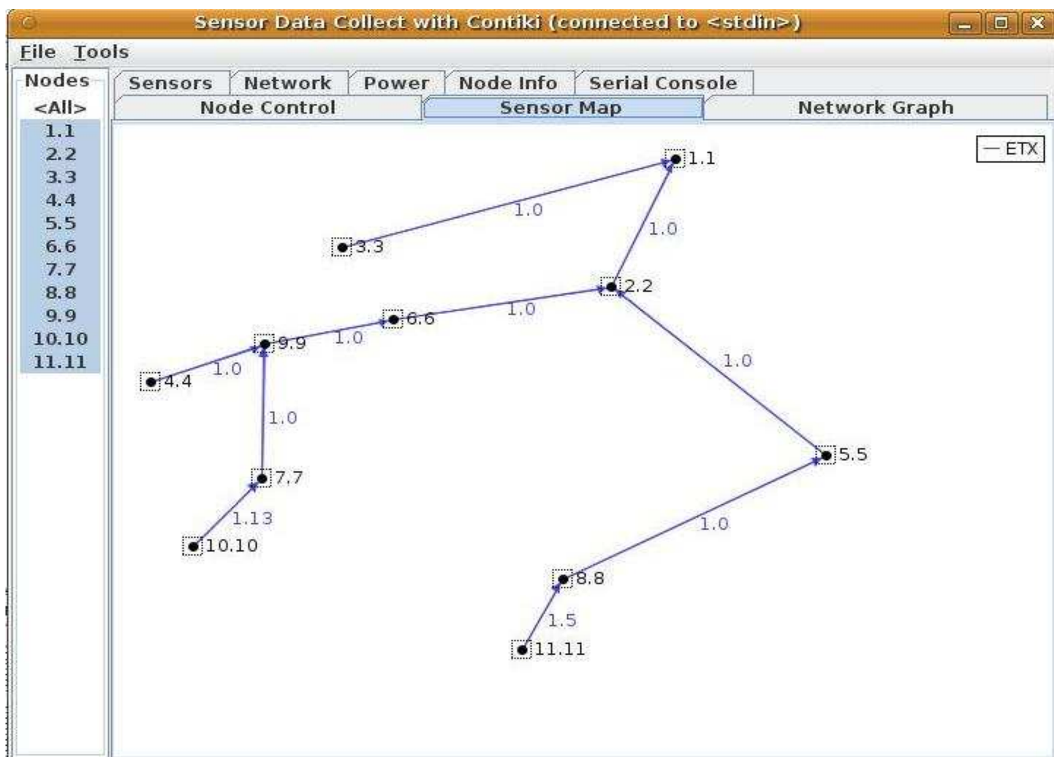


Figura 19: Topología luego de aislar al nodo 3

Ahora el nodo 2 debería tener en su tabla de ruteo a los nodos {4, 10, 7, 9, 6, 11, 8, 5}. Ejecutamos el

comando netprint en el nodo 2 y obtenemos el siguiente resultado.

Figura 20: Tabla de ruteo del nodo 2

Como se puede observar, el nodo 2 sabe llegar solo a los nodos {4, 6, 8, 5}. El 5 siempre estuvo en su tabla de ruteo. Luego con el cambio de topología los nodos {4, 6, 8} cambiaron de preferred parent y por lo tanto enviaron mensajes DAO. Por otro lado los nodos {10, 7, 11} mantuvieron su preferred parent y por lo tanto no enviaron mensajes DAO. Como tampoco se envían mensajes DAO periódicos estos nodos nunca enviarán mensajes para que el nodo 2 actualice la tabla de ruteo.

4.7.4 El Bug en el código

Según el draft-ietf-roll-rpl:m

```
Triggering DAO Message
Nodes can trigger their sub-DODAG to send DAO messages.
node maintains a DAO Trigger Sequence Number (DTSN),
which it communicates through DIO messages.
If a node hears one of its DAO parents increment its DTSN,
the node MUST schedule a DAO message transmission using rules
in Section 9.3 and Section 9.5.
```

En el código de Contiki siempre que un nodo envía un mensaje DIO, el numero DTSN se incrementa.

Por lo tanto siempre que un nodo recibe un mensaje DIO de su preferred parent debería enviar un mensaje DAO. Esto no es así ya que en la rutina “rpl_process_dio” siempre se retorna antes de chequear si es necesario enviar un mensaje DAO. A continuación un fragmento de código:

```
//La siguiente condición siempre es verdadera.
if(rpl_process_parent_event(dag, p) == 0) {
  //Siempre se retorna aquí.
  return;
}
//Nunca se llega a esta parte del código, lo que hace es chequear si el DTSN
// se incremento y en dicho caso se agenda para enviar un mensaje DAO.
if(should_send_dao(dag, dio, p)) {
   rpl_schedule_dao(dag);
}
```

Ahora vemos porque la condición es siempre verdadera:

En la rutina rpl_process_parent_event se encuentra el siguiente código:

```
if(parent_rank == INFINITE_RANK ||

        !acceptable_rank(dag, dag->of->calculate_rank(NULL, parent_rank))) {


 return 0;

}
```

En donde siempre se retorna 0.

Donde la siguiente expresión pasada como parámetro vale:

```
dag->of->calculate_rank(NULL, parent_rank) = parent_rank + 5 *256
```

En la rutina acceptable_rank se encuentra el siguiente código:

```
static int
acceptable_rank(rpl_dag_t *dag, rpl_rank_t rank)
{
 return rank != INFINITE_RANK && (dag->max_rankinc == 0 ||
   DAG_RANK(rank, dag) <= DAG_RANK(dag->min_rank + dag->max_rankinc, dag));
}
```

La condición de acceptable_rank puede ser siempre falsa para un nodo:

rank <= min_rank + max_rankinc

Donde:

rank = valor ingresado como parámetro = parent_rank + 5 *256

min_rank = es el mínimo rank que alguna vez tuvo el nodo

max_rankinc = 3*256

Entonces:

parent_rank + 5 *256 <= min_rank + 3*256

parent_rank+2*256 <=min_rank

Si el min_rank es muy chico porque en algún momento los nodos tuvieron muy buena comunicación por ejemplo porque estuvieron muy cerca y luego se alejan, entonces nunca será verdadera la condición de arriba y por lo tanto siempre se retornará antes de chequear si es necesario enviar un mensaje DAO.

Si los nodos tienen muy buena comunicación y nunca se alejan, también siempre será falsa a condición de arriba ya que en este caso min_rank = min_parent_rank + min_enlace. Donde min_enlace=256. Entonces como ejemplo estos 2 casos siempre tendrán problema.

4.7.5 Reparación del Bug

Se envío un mensaje al foro de los creadores de Contiki reportando el Bug. También se probó eliminar la siguiente condición de la rutina rpl_process_parent_event ya que no se le encontró sentido:

```
if(parent_rank == INFINITE_RANK ||

        !acceptable_rank(dag, dag->of->calculate_rank(NULL, parent_rank))) {


 return 0;

}
```

Al eliminar está condición siempre se chequeará el DTSN y en caso que se incremente se enviará un mensaje DAO. Tal como menciona el IETF.

El problema es que el Contiki siempre incrementa el DTSN al enviar mensajes DIO. Entonces siempre que se reciba un mensaje DIO del preferred parent se va a generar un mensaje DAO que se propagara de parent en parent. A priori esto parece que puede producir un gran incremento del consumo.

Se probó eliminar la condición. Esto reparo el Bug y no incrementó el consumo tanto como se esperaba a continuación los resultados.

Se hizo la misma prueba que falló anteriormente. En este caso el nodo 2 aprendió las rutas para llegar a todos los nodos de su Sub-DODAG {10, 9, 4, 6, 7, 8,11, 5}.

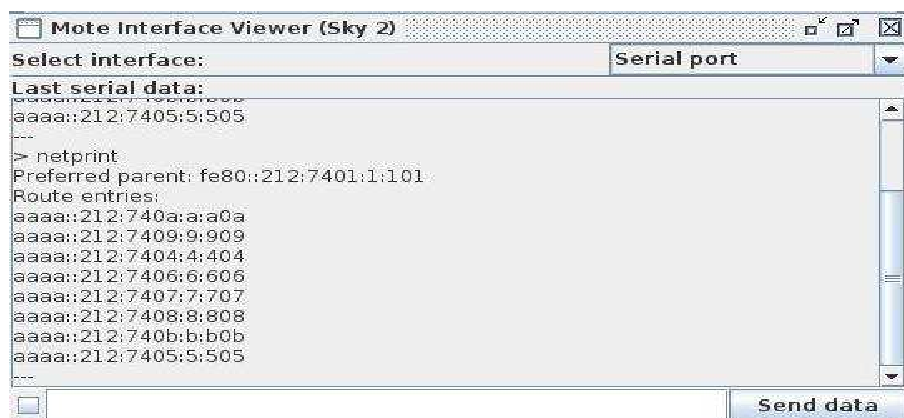


Figura 21: Tabla de ruteo del nodo 2

A continuación se analiza el consumo con la topología descripta anteriormente. Se corre la simulación durante 1 hora. La recolección de datos de los nodos es cada 60 segundos.

RDC con Contiki Original:

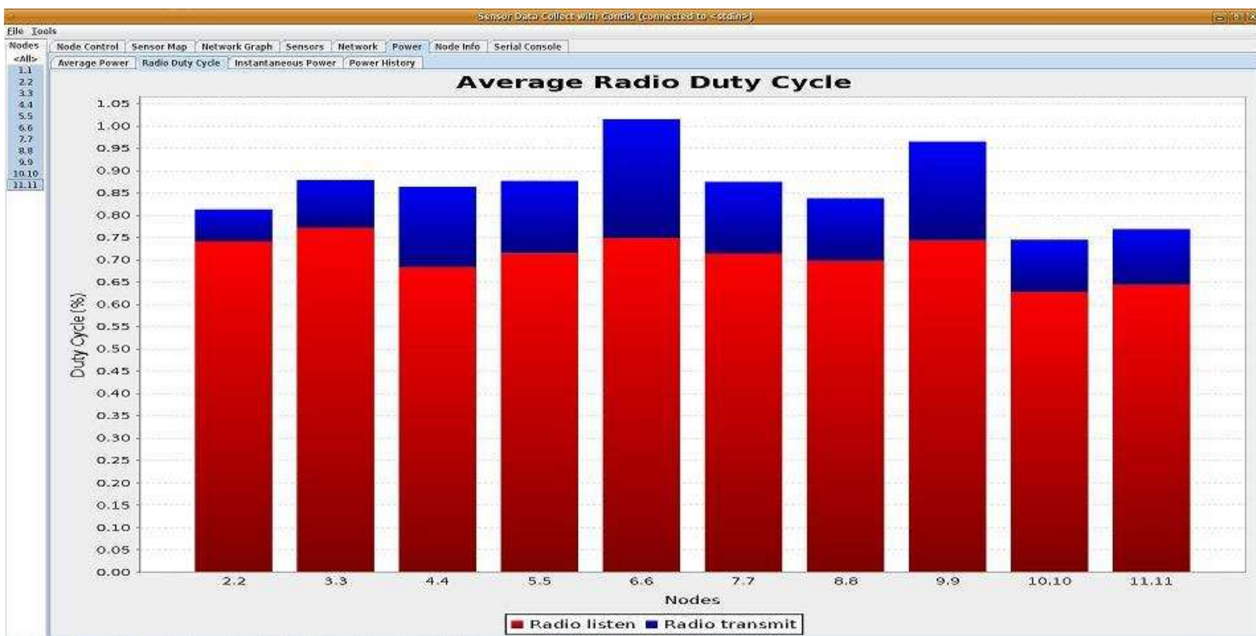


Figura 22: RDC promedio original de los nodos en la simulación

RDC con el Bug Arreglado:

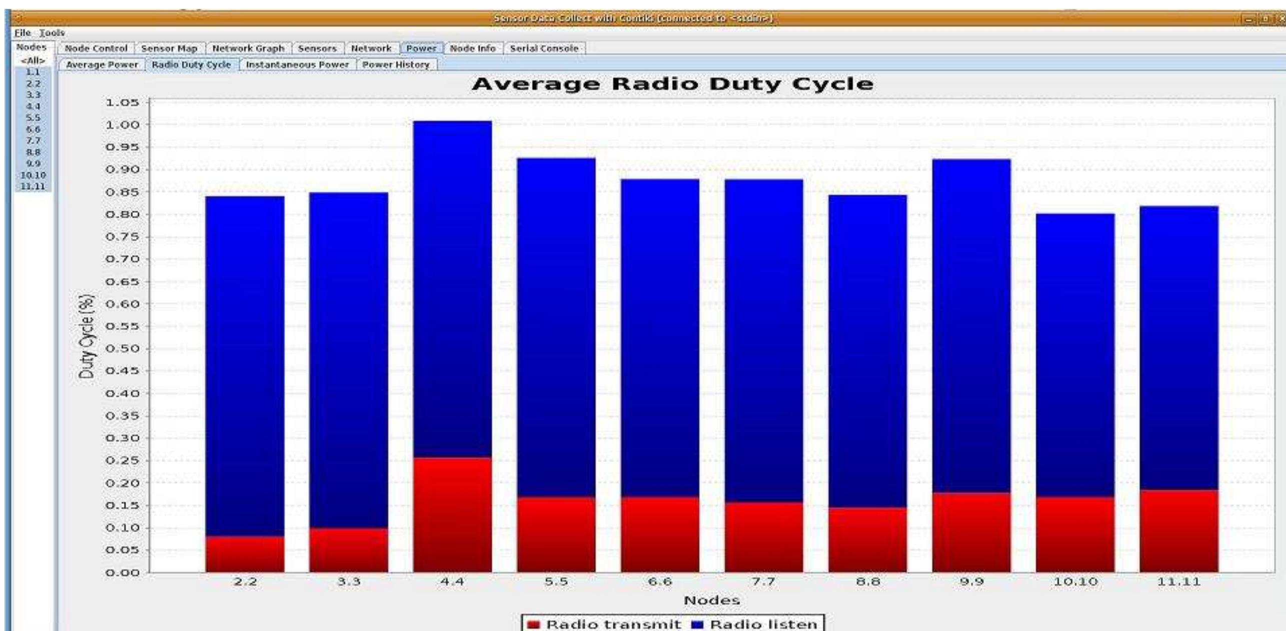


Figura 23: RDC promedio de los nodos en la simulación, con el bug arreglado

Como vemos el RDC es bastante similar y esta solución sería aceptable. De todas maneras sería conveniente incrementar el DTSN cada cierta cantidad de minutos y no cada vez que se envía un mensaje DIO.

4.8 Algoritmo Trickle

Trickle [30]es un algoritmo para propagar datos de manera confiable. Este algoritmo asegura que

todos los nodos tengan la última versión de los datos enviados. Trickle detecta cambios de versión de manera que los nodos puedan mantenerse actualizados. El algoritmo permite que los nodos que comparten un medio con pérdidas intercambien información de manera robusta, con eficiencia energética, de manera sencilla y escalable. El ajuste dinámico de las ventanas de transmisión le permite a Trickle difundir información nueva bajando la cantidad de transmisiones en la capa de red mandando pocos mensajes de red.

A partir del algoritmo Trickle se pueden implementar mecanismos de comunicación confiable uno-a-muchos diseñado para redes de baja potencia. Utiliza retransmisiones periódicas para asegurar que los mensajes perdidos sean retransmitidos. Para evitar sobrecargar la radio con demasiadas retransmisiones, el algoritmo proporciona un mecanismo para reducir el número de mensajes que se envían. Se asignan números de secuencias para poder chequear si todos los nodos recibieron la última versión del mensaje Trickle. Si algún nodo se da cuenta que otro tiene un número de secuencia viejo, le envía un mensaje para actualizarlo. De este modo se busca asegurar que la última versión del mensaje llegó a todos los nodos.

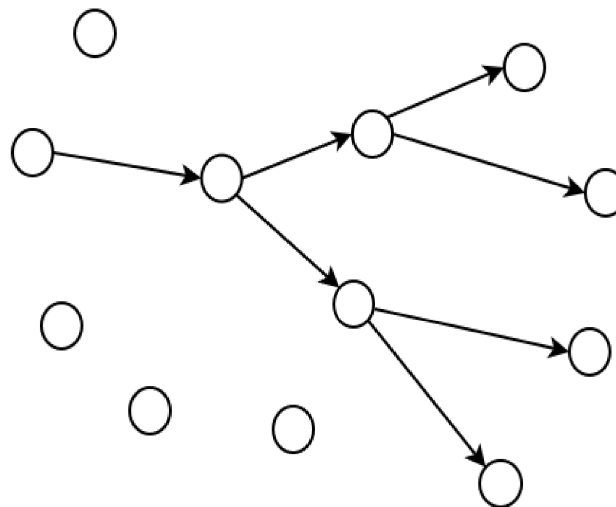


Figura 24: Esquema de comunicación one-to-many

Se trata de un protocolo desarrollado por Philip Levis y luego propuesto por el grupo ietf-roll que es un grupo de la IETF que se ocupa del desarrollo del protocolo de ruteo RPL, pensado especialmente para las redes desde baja potencia y con pérdidas.

El algoritmo tiene subyacente un modelo de consistencia que rige la comunicación entre los nodos. Cuando un nodo encuentra una inconsistencia con el vecino, este nodo responde rápidamente (mili segundos) de manera de resolver esta inconsistencia. En la medida de que la información intercambiada sea consistente, los nodos enlentecen su comunicación de manera de reducir lo más posible el envío de paquetes (algunos paquetes por hora).

En lugar de inundar la red con paquetes, el algoritmo controla la tasa de envío de paquetes de modo que cada nodo escucha muy pocos paquetes.

4.8.1 Principales características del Algoritmo

Un nodo transmite hasta que escuche una transmisión que le indique que sus datos son redundantes. El estado de información de ruteo y la actualización de versión de software son ejemplos de datos que se analizan.

Este principio de funcionamiento permite propagar rápidamente información de manera robusta frente a desconexiones esporádicas permitiendo también ampliar la cantidad de nodos de la red con un bajo overhead de mantenimiento.

La forma en que Trickle realiza los envíos depende del protocolo IP subyacente y del modo en que los protocolos de capas superiores usan el algoritmo. En IPv6 se puede usar una dirección de multicast local mientras que en IPv4 se puede utilizar la dirección de broadcast 255.255.255.255.

Trickle produce en esencia dos resultados. Inconsistencia de datos cuando los datos propios y los ajenos difieren, o consistencia cuando los datos coinciden.

El hecho de que la comunicación de Trickle sea tanto en transmisión como en recepción le permite operar en redes densas o de escasa cantidad de nodos. En redes escasas se requiere una mayor cantidad de transmisiones por nodo pero esta característica permite que la utilización del medio compartido no crezca.

Parámetros de configuración

Trickle utiliza un temporizador que tiene tres parámetros de configuración, y una constante k.

- ***Tamaño Mínimo de intervalo, Imin***, tiene unidades de tiempo (milisegundos, segundos). Un protocolo puede definir *Imin=100ms*.

- ***Tamaño Máximo del intervalo, Imax***, se describe como un número natural *n* tal que, $Imáx = Imin * 2^n$, es decir $n = \log_2(Imáx / Imín)$. Por ejemplo, un protocolo puede definir $Imáx = 16$ entonces si $Imín = 100\text{ms} \rightarrow Imáx = 100\text{ms} * 65536$.

- La ***Constante de Redundancia k***, es un entero positivo.

Variables manejadas por Trickle

- I, tamaño del intervalo actual,
- t, un tiempo en el intervalo actual
- c, un contador.

4.8.2 Descripción del Algoritmo

Trickle tiene 6 reglas:

1) Cuando inicia la ejecución configura dentro *I* dentro del intervalo [Imin, Imax] .
2) *c=0*, y *t* es aleatorio dentro de [I/2, I)
3) Si transmisión es *consistente* => c=c+1
4) en el instante *t*, Trickle trasmite si y solo si *c<k*
5) cuando el intervalo I expira, Trickle duplica la longitud del intervalo. Si el intervalo resultante es mayor que *Imax*, se configura *I=Imax*.
6) Cuando Trickle escucha una transmisión inconsistente, y *I>Imin* re-inicializa el temporizador de Trickle. Para esto configura *I=Imin* e inicializa un nuevo intervalo como en el paso 2. Si cuando escucha un inconsistencia *I=Imin* Trickle no hace nada. Trickle puede reiniciar su temporizador en respuesta a eventos externos.

Estos eventos externos deben definirse en cada caso que se quiera utilizar el algoritmo. En cualquier aplicación del algoritmo Trickle es responsabilidad del desarrollador definir convenientemente los conceptos de consistencia/inconsistencia y programar las funciones para detectarlas.

4.8.3 Usos típicos de Trickle

a) Inundación/diseminación confiable

Un protocolo puede usar Trickle para anunciar periódicamente el último dato recibido, típicamente un número de versión. Una inconsistencia ocurre cuando un nodo recibe un número de versión nuevo o recibe un nuevo dato. La consistencia ocurre cuando recibe un número de versión más viejo o igual al que tiene. Cuando recibe un número de versión anterior el nodo envía una actualización en lugar de reiniciar el temporizador. Los nodos con versiones anteriores que reciben la actualización reiniciarán sus propios *temporizadores* intentando acelerar la propagación de nuevo dato.

Se ha utilizado para realizar *multicast*, configuración de red e instalación de nuevas versiones de programas.

b) Control de tráfico de ruteo

Un protocolo usa Trickle cuando envía datos que contienen información del estado de las rutas. Una inconsistencia ocurre cuando la topología cambia de modo que se puedan producir loops o un cambio en el costo de una determinada ruta. La consistencia se define como la situación en la que la topología opera de modo de que logra entregar los paquetes exitosamente.

Un ejemplo es su utilización en RPL, CTP (Collection Tree Protocol) y algunas implementaciones comerciales de IPv6.

4.8.4 Algunos comentarios

Ya existe una aplicación Trickle implementada en el stack Rime. Nosotros tomaremos esta aplicación y la adaptaremos para que funcione con IPv6. Luego se utilizará para que el sink configure los nodos de la red. En el Capítulo 6: Diseño e Implementación se explicará como se adaptó.

Capítulo 5: Capa Aplicación

5.1 Introducción

En este capítulo se describirá las aplicaciones que ya están implementadas en Contiki. Primero se explicará la la aplicación "rpl-collect" que es la que usaremos como base para el desarrollo de nuestra aplicación ContikiWSN. Luego se describirá la aplicación auxiliar "shell" la cual usaremos como interfaz con el sistema operativo.

5.2 Aplicación rpl-collect

En el sistema operativo Contiki ya se encuentra implementada una aplicación llamada "rpl-collect" la cual consiste en una aplicación para recolectar datos. Se encuentra disponible tanto el código para el nodo sink como para los nodos senders. El nodo sink es el que recibe los datos sensados de los nodos senders utilizando direccionamiento IPv6 y el protocolo de ruteo RPL.

Nuestra objetivo será mejorar la aplicación existente y agregarle nuevas funcionalidades, lo cual se describe en el capítulo 6.

5.2.1 Nodo sink

En el nodo sink se corren los procesos *udp_server_process* y *collect_common_process.*

El proceso *udp_server_process* cuando se inicializa ejecuta las siguientes tareas:

1. Le asigna una IP fija al nodo, esta IP es conocida por todos los nodos senders

2. Setea al nodo como nodo raíz de RPL, por lo tanto cuando envíe mensajes de control RPL, estos contendrán información indicando que él es el nodo raíz.

3. Crea una conexión UDP llamada *server_conn* y especifica su puerto de origen y destino. Esta conexión se usará para recibir los datos sensados por los nodos senders. El nodo sink solo aceptará paquetes UDP cuyos puertos coincidan con el de alguna conexión. En caso contrario los descartará. Cuando un paquete es identificado con una conexión se envía el evento *tcpip_event* al proceso que creó dicha conexión. De esta manera se le avisa al proceso la llegada del paquete y este deberá encargarse de procesar a nivel de aplicación el paquete recibido.

   Por lo tanto cuando un paquete llegue con:

   - Puerto destino del paquete UDP recibido = Puerto origen de conexión "server_conn"

   - Puerto origen del paquete UDP recibido = Puerto destino de conexión "server_conn"

   Se llamará al proceso *udp_server_process* el cual se encargará de procesar el paquete UDP recibido.

4. Configura el duty cycle de la radio de manera que este siempre prendida. Como el sink no es alimentado por baterías, no se necesita optimizar su consumo.

Luego de terminar las inicializaciones el proceso entra en un loop infinito en donde lo único que se hace es esperar el evento *tcpip_event.* Este evento es enviado al proceso cuando se recibe un paquete UDP perteneciente a la conexión *server_conn*. Cuando llega este evento, se llama a una función que se encarga de enviar por serial todos los datos recibidos del sender y la hora en que estos se recibieron según el reloj del sink. De esta manera el Software collect-view recibe los datos por serial y luego los procesa y los despliega en la interfaz gráfica.

El proceso *collect_common_process* se usa para el nodo sink y también en los senders.

Este proceso cuando se inicializa se encarga de:

1. Configurar el módulo UART (Universal Asynchronous Receiver-Transmitter) para recibir y enviar datos por serial.

2. Crear y setear los etimers que se encargarán de despertar los nodos senders para sensar y enviar datos. Estos etimers también despiertan al nodo sink, pero no se utilizan para ningún propósito.

Luego de terminar las inicializaciones queda en un loop infinito esperando por los eventos *serial_line_event_message* y process*_event_timer.*

- *serial_line_event_message* : Cuando se recibe este evento quiere decir que se recibieron datos por serial. Se compara la línea recibida con todos los comandos existentes, si coincide con alguno, el comando es ejecutado. Los comandos disponibles son:
    - `net:` Imprime por serial el preferred parent y la tabla de ruteo.
    - `time[n]:` Se actualiza el tiempo del sink. Donde n son los segundos a partir de 1970.
    - `mac[n]:` Prende/Apaga el driver RDC que se encarga de manejar la cadencia con la que se prende la radio. Con 1 prende el driver, con 0 apaga el driver y deja la radio prendida. Se utiliza para dejar la radio prendida en el sink.

- process*_event_timer:* Indica que expiró un etimer, se puede consultar el parámetro *data* para saber cuál fue el etimer que expiró. Se utilizan 2 etimers el *period_timer* y el *wait_timer* . El *period_timer* se encarga de llevar la cuenta del periodo para saber cuándo se debe enviar mensajes. El *wait_timer* se encarga de introducir una delay aleatorio para evitar colisiones. La lógica es la siguiente, por ejemplo si configuramos el *period_timer* con 60 segundos y el *wait_timer* con 30 segundos. Cada 60 segundos va a llegar un evento process*_event_timer* enviado por el *period_timer*, se procesa este evento reseteando el etimer para que vuelva a enviar un evento dentro de 60 segundos y a su vez se setea el *wait_timer* para que envíe un evento en un tiempo aleatorio entre 0 y 30 segundos. Cuando llega el process*_event_timer* enviado por el *wait_timer* si el proceso se está corriendo en un nodo sender se sensan los datos, se construye el mensaje y se envía por la conexión *client_conn.* Si el proceso se está corriendo en un nodo sink, no se hace nada.

5.2.2 Nodo sender

En el nodo sender se ejecutan los procesos *udp_client_process* y *collect_common_process.* El proceso *udp_client_process* cuando se inicializa ejecuta las siguientes tareas:

1. Se genera una dirección IP única a partir de la dirección MAC.

2. Se crea una conexión UDP llamada *client_conn* y define sus puertos de origen y destino de manera

que sean coherentes con la conexión *server_conn* del sink. La conexión *client_conn* es usada por los nodos senders para enviar los datos sensados al sink.

El proceso *collect_common_process* es utilizado por el nodo sender para llamar según el período programado a la función *collect_common_send* que se encargará de sensar y enviar los datos al sink. Esta función lleva la cuenta del número de secuencia para los mensajes que se van a enviar. Este número va del 0 al 256, cada vez que se envía un mensaje se incrementa este número, con la única particularidad de que cuando se produce overflow se pasa al número 128 para identificar si un nodo se resetea. De esta manera en la recepción de mensajes chequeando este número se puede saber la cantidad de mensajes que se perdieron de un nodo determinado. Luego esta función guarda en la estructura *collect_view_data_msg* todos los datos sensados y de status del nodo sender que serán enviados al nodo sink, además se agrega el número de secuencia por fuera de esta estructura. Finalmente se envía un mensaje con todos estos datos por la conexión *client_conn* utilizando como puerto destino el puerto de origen de la conexión "server_conn" del sink y como IP destino la IP del sink.

5.3 Aplicación Shell

Existe una aplicación auxiliar para la arquitectura Rime que implementa un Shell. Es la interfaz entre el usuario y el sistema operativo. Actúa a través de las líneas de comando que introduce el usuario. Su función es la de leer la línea de comandos, interpretar su significado, llevar a cabo el comando y después desplegar el resultado.

Las funcionalidades del shell son:

Prompt: Cada vez que el Shell esté listo para recibir una nueva línea despliega "a.b : Contiki>" donde a y b son el penúltimo y ultimo byte de la dirección IP expresados en formato decimal del nodo en donde se está ejecutando el Shell.

Pipe: Permite asignar la salida estándar de un comando a la entrada estándar de otro. "comando1 | comando2".

Back-ground: El comando se procesa invisiblemente para el usuario y deja libre el Shell para que se pueda seguir ingresando más comandos. "comando &".

Terminar un comando activo: Se puede terminar cualquier comando que este activo. Con "kill <command>" se termina un comando específico y con "killall" se terminan todos los comandos activos.

Terminar un comando front-ground: Con "~K" se termina cualquier comando que este en el front-ground y de este modo se libera el Shell para que pueda ser usado nuevamente.

Help: El comando "help" indica todos los comandos disponibles en el Shell

Utilizaremos la aplicación shell en ContikiWSN. En el Capítulo 6: Diseño e Implementación se explicará como se adaptó.

Capítulo 6: Diseño e Implementación

6.1 Introducción

En este capítulo se describirán los cambios que se hicieron al código y las nuevas funcionalidades que se agregaron a la aplicación y al sistema operativo Contiki. A la nueva aplicación "rpl-collect" modificada por nosotros la llamaremos ContikiWSN.

6.2 Compilación y MakeFiles

Los archivos makefile especifican las dependencias de los diferentes archivos del código para que cuando se compilen se genere el archivo binario correctamente. Cuando se implementa un programa que utiliza el Sistema Operativo Contiki se necesita crear un makefile que incluya a todos los archivos que se usarán.

Contiki tiene un makefile implementado en la carpeta raíz llamado Makefile.include. Cuando se quiera implementar un programa nuevo se debe crear un archivo makefile el cual incluya al archivo Makefile.include para que se incluyan todos los archivos necesarios del Sistema Operativo.

A continuación se muestra el makefile del programa ContikiWSN:

```
#Primero en la variable CONTIKI se debe indicar donde se encuentra
#la carpeta raíz de Contiki en donde se encuentra el Makefile.include
CONTIKI = ../../..
#Luego en APPS se debe indicar que aplicaciones se usarán.
APPS = powertrace collect-view trickle-ipv6 serial-shell
#En CONTIKI_PROJECT se debe indicar cuál es el programa a compilar.
#En este caso se quiere compilar a la vez el programa del sender y del sink.
CONTIKI_PROJECT = udp-sender udp-sink
#En la variable PROJECT_SOURCEFILES se debe agregar si se precisa incluir
#algún archivo .c extra para que sea posible la compilación.
PROJECT_SOURCEFILES += collect-common.c

#Prendiendo estas flags se le indica al Makefile.include que se
#utilizará IPv6 y este compila los archivos necesarios.
WITH_UIP6=1
UIP_CONF_IPV6=1

#Cuando se ejecute make TARGET=sky all se compilará los
#programas incluidos en CONTIKI_PROJECT
all: $(CONTIKI_PROJECT)
```

```
#Se incluye el archive Makefile.include
include $(CONTIKI)/Makefile.include
```

Si se crean aplicaciones nuevas en la carpeta apps, también se le deben crear archivos makefiles.

A continuación se muestra el makefile de la aplicación trickle-ipv6 que se encuentra en apps\trickle-ipv6. Es mandatorio que el nombre de la carpeta sea el mismo que el de la aplicación.

```
#El makefile de la aplicación se debe llamar
#"Makefile." + <nombre de la aplicación>
# Por lo tanto para este caso el nombre será Makefile.trickle-ipv6
#En la variable trickle-ipv6_src se debe incluir los archivos que
#se tienen que agregar a la compilación para que la aplicación funcione.
#Notar que el nombre de la variable es  <nombre de la aplicación>_src
trickle-ipv6_src = trickle-ipv6.c
```

6.3 Optimización de memoria RAM y ROM

El primer problema que se nos presento fue la falta de espacio en la memoria flash y RAM para implementar código nuevo. La capacidad de memoria del microcontrolador MSP430 F1611 es de 48KB de Flash y 10KB de RAM.

Compilando la aplicación rpl-collect y dejando todos los parámetros por defecto de Contiki OS se obtienen los siguientes resultados:

- En el nodo sink se usan 44.4KB de memoria Flash y 9.2KB de memoria RAM.
- En el nodo sender se usan 46.5KB de memoria Flash y 9.3KB de memoria RAM.

Como vemos hay muy poco espacio disponible en Flash y en RAM directamente no queda espacio utilizable ya que el poco espacio que queda libre se utiliza para el stack.

Para optimizar la memoria se hizo lo siguiente:

1. Eliminación del stack TCP de la compilación.

   El stack TCP no se utiliza ya que todas las comunicaciones se harán a través de UDP. Por lo tanto se excluye de la compilación. Esto se hace definiendo:

   ```
   #define UIP_CONF_TCP      0
   ```

   En el archivo core\net\uipopt.h

   Luego de esto:

   En el nodo sink se usan 40.5KB de memoria Flash y 9.0KB de memoria RAM.

   En el nodo sender se usan 42.6KB de memoria Flash y 9.0KB de memoria RAM.

Se agregan flags de compilación para optimizar la compilación.

Se agregaron en el Makefile.include ubicado en la carpeta raíz del Contiki las siguientes flags de compilación para optimizar la compilación:

```
CFLAGS += -ffunction-sections
LDFLAGS += -Wl,--gc-sections,--undefined=_reset_vector__,--undefined=InterruptVectors,--
undefined=_copy_data_init__,--undefined=_clear_bss_init__,--undefined=_end_of_init__
```

Luego de esto:

En el nodo sink se usan 38.1KB de memoria Flash y 9.0KB de memoria RAM.

En el nodo sender se usan 39.7KB de memoria Flash y 9.0KB de memoria RAM.

Se disminuye la cantidad de vecinos del cache y la de elementos de la tabla de ruteo.

Disminuyendo la cantidad de vecinos conocidos que cada nodo puede guardar y también disminuyendo la cantidad de elementos de la tabla de ruteo se logra liberar una gran cantidad de espacio en la memoria RAM.

Los valores por defecto de Contiki OS que están en platform\sky\contiki-conf.h son:

UIP_CONF_DS6_NBR_NBU = 30 (Cuantos vecinos se guardan en el cache)
UIP_CONF_DS6_ROUTE_NBU = 30 (Cantidad máxima de elementos de tabla de ruteo)

Los valores nuevos:

UIP_CONF_DS6_NBR_NBU = 10
UIP_CONF_DS6_ROUTE_NBU = 1 (Además de la ruta por defecto)

Como nueva restricción se tiene que los nodos no deberán tener contacto con más de 10 vecinos. Debido a que cuando a un nodo le llega un dato de un vecino que no tiene en el cache y además el cache está lleno, borra el vecino que se guardó hace más tiempo. Esto puede producir que se borre el preferred parent y por lo tanto produzca problemas para comunicarse con el sink. Por lo tanto se tiene que asegurar en la topología que cada nodo no tenga contacto con más de 10 vecinos. Esto no es un inconveniente mayor ya que no sería normal implementar una topología de este estilo.

Por otro lado se seteará la cantidad de rutas a 1 ya que la tabla de ruteo en la implementación ContikiWSN no se usará debido a que no se necesitará enviar mensajes de un nodo sender a otro nodo sender. Si se usará el ruteo de los nodos sender hacia el sink, y eso se hace con la lógica de los preferred parents y las rutas por defecto las cuales no se cuentan como elementos de las tablas de ruteo.

Finalmente:

En el nodo sink se usan 38KB de memoria Flash y 6.6KB de memoria RAM.

En el nodo sender se usan 39.7KB de memoria Flash y 6.7KB de memoria RAM.

Ahora si hay espacio suficiente para la aplicación ContikiWSN.

Archivos del código modificados

platform\sky\contiki-conf.h	modificado	Se cambia la capacidad de la tabla de ruteo y la de vecinos.
\Makefile.include	modificado	Se agregan instrucciones para que la compilación ocupe menos memoria flash

6.4 Modificación y Uso de Trickle

Contiki OS contiene una aplicación Trickle para el stack Rime. Nosotros nos encargaremos de portarla para IPv6.

6.4.1 Trickle Original

La aplicación existente está ubicada en core\net\rime\trickle.c y consiste en lo siguiente:

Se utiliza un protothread para el algoritmo de Trickle llamado `run_trickle` .En este se define el intervalo de Trickle con la siguiente sentencia:

```
interval = initial_interval * 2^(interval_scaling)
```

El valor de `initial_interval` es definido cuando se inicializa Trickle y `interval_scaling` se inicializa con 0 y puede incrementarse hasta el valor `INTERVAL_MAX`. Como vemos cada vez que se incrementa en uno `interval_scaling` el intervalo de Trickle se incrementa al doble.

Por lo tanto el intervalo mínimo se produce cuando `interval_scaling = 0` y vale

```
Imin= initial_interval
```

El intervalo máximo se produce cuando `interval_scaling = INTERVAL_MAX` y vale

```
Imax= interval * 2^(INTERVAL_MAX)
```

Cuando se inicializa Trickle se setea el valor de `initial_interval` y también a `interval_scaling` se le da el valor de cero. Luego se llama al protothread `run_trickle`.

La primera vez que se corre `run_trickle` lo primero que se hace es darle a `interval` el valor del intervalo mínimo ya que `interval_scaling` vale cero. Luego se setean 2 timers que llamarán al protothread cuando expiren. El primero se setea con el valor de `interval` el segundo se setea con un valor randómico entre `interval/2` y `interval.` Por lo tanto este segundo timer será el que expire primero. Luego el protothread entrega el procesador y queda esperando a que expire este timer. Cuando expira, se chequea la cantidad de mensajes duplicados que se recibieron. Cada vez que se recibe un mensaje Trickle con la misma versión que tiene el nodo se aumenta el contador de mensajes duplicados. Si la cantidad de mensajes duplicados supera el umbral `DUPLICATE_THRESHOLD` el nodo evita enviar un mensaje Trickle para actualizar a los vecinos ya que lo más probable que tengan la misma versión que él. En caso contrario el nodo envía un broadcast con su versión de mensaje Trickle. Luego se vuelve a entregar el procesador y se espera a que venza el timer que indica el fin del intervalo. Cuando expira este timer, se incrementara en uno `interval_scaling` por lo tanto se incrementara al doble `interval`.

Envío de mensajes Trickle

Un nodo para enviar un mensaje Trickle usa una conexión Rime de broadcast la que llegará a todos los vecinos.

Recepción de mensajes Trickle

Cuando se recibe un mensaje Trickle se llama a una función que compara la versión del mensaje recibido con la del nodo.

Si las versiones son iguales, se incrementa el valor de los mensajes duplicados.

Si la versión recibida es más antigua, entonces el nodo envía un broadcast con su versión de mensaje Trickle, ya que es más actualizada. Además setea `interval_scaling = 0`.

Si la versión recibida es más nueva, entonces se actualiza la versión de Trickle, se procesan los datos nuevos y se reinicia el protothread y el intervalo de Trickle.

6.4.2 Trickle para IPv6

La nueva aplicación de Trickle adaptada para IPv6 está ubicada en la carpeta "apps\trickle-ipv6\". Los principales cambios que se necesitan hacer son para el envío y recepción de mensajes Trickle. Estos mensajes se enviarán a través de paquetes de control ICMP de IPv6. Para esto se necesitará crear un nuevo tipo de mensaje ICMP llamado ICMP6_TRICKLE.

Inicialización de Trickle IPv6

Con la función `trickle_ipv6_init` se inicializa Trickle. Todos los nodos que necesiten usar Trickle deben llamarla. Como parámetro se le debe pasar el intervalo inicial con el cual se establece el valor de `initial_interval` además se setea `interval_scaling=0` y se agrega el nodo en la dirección de multicast de Trickle. Esto quiere decir que a partir de este momento el nodo cuando reciba un paquete con dirección IP destino igual a la de multicast de Trickle, el nodo procesará el paquete y no lo descartará.

Envío de mensajes Trickle IPv6

Los datos a enviar se cargan en el buffer UIP_ICMP_PAYLOAD, el cual está diseñado para albergar el payload de los mensajes ICMP. Luego se envía el mensaje utilizando la función `uip_icmp6_send` y especificando en sus parámetros la dirección IP del multicast, el tipo de mensaje ICMP (ICMP6_TRICKLE) y cuantos bytes se cargaron al buffer UIP_ICMP_PAYLOAD.

Recepción de mensajes Trickle IPv6

Con la inicialización de Trickle se agrega la dirección multicast de Trickle como dirección valida. De esta manera los mensajes que lleguen con esta dirección IP serán aceptados. Además cuando la rutina `uip_process` reconozca que se recibió un mensaje ICMP del tipo ICMP6_TRICKLE, se avisará a la

aplicación Trickle la cual interpretará el mensaje y procesará los datos del payload. Al igual que en el caso de Trickle con rime el nodo compara la versión del mensaje recibido con la suya y lo procesa de acuerdo al resultado.

Configuración de Parámetros

Lo bueno de Trickle es que cuando se precisa actualizar la red, se envían los mensajes con una cadencia rápida y cuando la red esta actualizada los mensajes se mandan más distanciados minimizando el consumo.

Los parámetros de Trickle se configurarán con los siguientes valores:

```
initial_interval = 1 segundo. => Intervalo mínimo = 1 segundo.
INTERVAL_MAX = 10 => Intervalo máximo = 1024 segundos = 17 minutos.
DUPLICATE_THRESHOLD = 1 => Si un nodo recibe un mensaje que tenga su misma versión, se
salteara de enviar el próximo mensaje Trickle.
```

Por lo tanto cuando el algoritmo este en régimen los nodos que no reciban mensajes duplicados enviarán cada 17 minutos un mensaje de Trickle minimizando el consumo.

Por otro lado cuando los nodos necesiten actualizarse se enviaran mensajes Trickle cada 1 segundo minimizando la latencia.

Utilización de Trickle

En la aplicación ContikiWSN los mensajes Trickle se usarán para la diseminación de datos del sink a los senders. De esta manera se puede asegurar que todos los nodos que estén al alcance de la red van utilizar el último mensaje enviado por el nodo sink para configurar sus parámetros.

Tener en cuenta que gracias a Trickle no habría problema con agregar nodos senders nuevos a la red o con nodos que se aíslen por un largo tiempo de la red. Ya que estos nodos cuando entren en contacto con la red se configurarán automáticamente a través de sus vecinos con los últimos datos enviados por el sink.

Datos enviados en el payload de los mensajes Trickle:

```
seqno[16]: Número de versión del mensaje Trickle. Cuanto más grande más nuevo es el mensaje. Al
utilizarse 16 bits la versión no puede ser más grande que 65535. Si se produce overflow dejaría de
funcionar. Pero debido a que nunca serán necesarias tantas versiones de mensajes Trickle esto no es
un problema.

samplePeriod[16]: Periodo de muestreo. Determina cada cuanto tiempo hay que sensar y guardar
mensajes en el buffer para luego enviarlos al sink.

time_seconds_now[32]:Se utiliza para sincronizar la hora. De esta manera todos los nodos Sender
tendrán la hora sincronizada con la del sink.

collect_activated[8]: Indica si los nodos senders deben sensar y enviar datos al sink o no.
```

Archivos del código creados y modificados

apps\trickle-ipv6\trickle-ipv6.c	creado	Se adapta trickle para ipv6 tomando como base el archivo trickle.c de rime
apps\trickle-ipv6\trickle-ipv6.h	creado	
apps\trickle-ipv6\Makefile.trickle-ipv6	creado	Se crea el makefile para la nueva aplicación
core\net\uip6.c	modificado	Se detecta la recepción de mensajes icmp de trickle y se envian a la aplicación trickle-ipv6.
core\net\uip-icmp6.h	modificado	Se crea el nuevo identificador de mensajes icmp del tipo trickle ipv6
examples\ipv6\ContikiWSN-Application\udp-sink.c	modificado	Se inicializa trickle en el sink
examples\ipv6\ContikiWSN-Application\udp-sender.c	modificado	Se inicializa trickle en los nodos senders

6.5 Detección de pérdida de comunicación con el sink

Cada nodo sender debe poder detectar cuando pierde comunicación con el sink. De esta manera los nodos sender podrán en la nueva implementación, primero, dejar de gastar energía para intentar enviarle mensajes con datos sensados al sink si se perdió la comunicación, y segundo, en vez de enviar los mensajes los podrá guardar en un buffer para enviarlos cuando se recupere la comunicación.

Algoritmos para detectar perdida de comunicación en Contiki

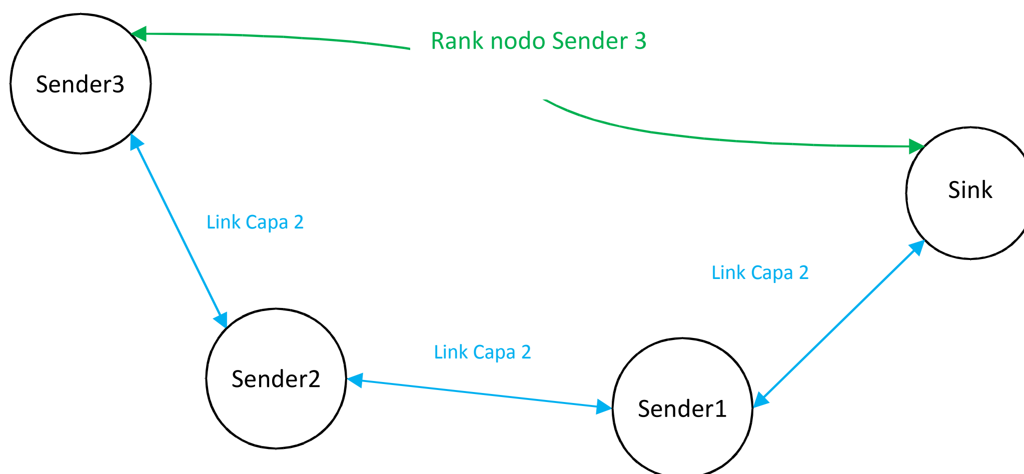


Figura 25: Ejemplo de Camino al Sink

Contiki utiliza dos algoritmos distintos para detectar la perdida de comunicación según dos casos.

CASO 1: Cuando un nodo pierde a su best parent y éste era el único camino para llegar al sink.

En este caso el nodo no va a recibir más mensajes del best parent, por lo tanto el rank del best parent

se mantendrá fijo. Pero igual se incrementará el rank del nodo porque el link con el best parent cada vez será peor por no recibir acks de éste. Llegar a rank infinito por este método demoraría mucho tiempo entonces lo que se hace es que después de cierto tiempo sin recibir mensajes del best parent se manda unos probes al best parent y si este no responde entonces se lo da por perdido, se pone el rank en infinito y se reinicializa la búsqueda de parents.

En Contiki la lógica está implementada de la siguiente manera:

Cuando se descubre un vecino nuevo se le setea un timer llamado REACHABLE el cual está por defecto en 600 segundos (UIP_CONF_ND6_REACHABLE_TIME) y para nuestra aplicación lo calibramos en 30 segundos.

Este timer se empieza a decrementar. Cuando llega a 0 segundos el vecino pasa a estado STALE. El vecino queda en este estado hasta que el mote envía un mensaje cualquiera (no cuentan mensaje de broadcast como los de RPL y Trickle). En este momento se pasa el vecino de STALE a DELAY y se vuelve a setear el timer REACHABLE pero a otro valor más chico el cual indica que se debe enviar un probe. Cuando el timer expira envía el probe al vecino, si este no responde vuelve a intentar. Si no responde 3 veces se da el vecino por perdido.

En cambio, si el vecino responde se lo pasa al estado REACHABLE y se resetea el timer REACHABLE al valor (UIP_CONF_ND6_REACHABLE_TIME/2) y cuando expire comenzara de nuevo el proceso pasando el vecino al estado STALE.

Por lo tanto, el tiempo que demora el mote en darse cuenta que el vecino se perdió depende fuertemente del periodo de muestreo. Y se perderá por lo general 1 mensaje con el cual el nodo se dará cuenta que la comunicación con el best parent se perderá.

CASO 2: El nodo sigue teniendo contacto con el best parent, pero pierde contacto con el sink.

Imaginemos una topología del estilo de la figura 31. Supongamos que el nodo Sender1 se rompe.

Entonces el nodo Sender2 va a incrementar su rank porque el link con el nodo Sender1 se perdió.

El nodo Sender3 aunque sigue manteniendo perfectamente el link con el nodo Sender2, usa el rank de su best parent como base. Recordemos que el rank de un nodo se calcula como el rank del best parent (que en este ejemplo empeora con el tiempo) sumado a la calidad del enlace con este mismo (en este ejemplo el link entre el nodo Sender2 y Sender3 se mantiene con la misma calidad por lo tanto este término es constante).

Por otro lado cada nodo lleva la cuenta del rank mínimo que alguna vez tuvo y en caso de que el rank de su best parent sea mayor al rank mínimo del nodo sumado a un offset de 3 unidades mínimas (3 * 256) entonces el nodo supone que el best parent perdió contacto con el sink. En este momento pone su rank en infinito y empieza con el algoritmo para encontrar parents desde cero.

Desde que el nodo Sender1 se rompe hasta que el nodo Sender3 detecta que se perdió comunicación con el Sink, se pierde al igual que en el CASO1 en general 1 mensaje.

Detección de Perdida de Comunicación en ContikiWSN

Luego de estudiar los algoritmos se determinó que el valor de rank infinito es un buen indicador para detectar perdidas de comunicación con el el sink.

En la aplicación ContikiWSN el nodo chequeará cada vez que tenga que enviar un mensaje al Sink si su rank es infinito. En caso afirmativo, se pospone el envío.

Archivos del código modificados

examples\ipv6\ContikiWSN-Application\collect-common.c	modificado	Se implementa la función para consultar si el rank es infinito

6.6 Buffereo de datos y envío de datos guardados

La aplicación ContikiWSN se encargará de que cuando un nodo pierda la comunicación con el sink, deje de enviarle mensajes y posponga su envío.

Para esto se tendrá un buffer circular, el cual se tomó de wikipedia http://en.wikipedia.org/wiki/Circular_buffer y se lo retocó para que se adapte a nuestro propósito. A la estructura que maneja el buffer se le agregó una variable que cuenta la cantidad de elementos. De esta manera se puede identificar fácilmente si el buffer está lleno o está vacío.

El código se guardó en la librería de Contiki en el directorio core\lib\circularbuf.c

La memoria RAM para el buffer es reservada en el momento de ejecución con la función malloc. Se utiliza 4 bytes para la estructura que maneja el buffer y 50 bytes para cada elemento del buffer.
Para la aplicación ContikiWSN se tendrán 10 elementos por lo tanto se usarán en total 504bytes. Cuando el buffer está lleno, se dejan de guardar elementos.

Cada elemento del buffer se compone de:

```
time_seconds[32] -> Hora a la que se sensaron los datos
seqno[8]-> Número de secuencia del mensaje
for_alignment[8]-> Para alinear de a 16 bits
collect_view_data_msg[352] -> estructura collect_view_data_msg que contiene datos sensados y más.
```

Se creará un proceso en los nodos senders llamado *ipv6_sender_process* que se encargará de guardar los datos y enviarlos con la siguiente lógica.

Se utilizará un etimer *store_timer* el cual se seteará con el periodo de recolección de datos. Cada vez que expire el etimer, se reseteará y además si esta activa la colección de datos, el nodo sender hará las medidas, creará un mensaje con el formato de la estructura *collect_view_data_msg* y lo guardará en el buffer. Este mecanismo tiene la ventaja con respecto al rpl-collect que las medidas se harán periódicamente respetando exactamente el periodo de recolección.

Por otro lado se utilizarán otros 2 etimers *send_timer_ipv6* y *wait_timer*, los cuales se usarán para enviar los datos. Estos dos timers tendrán como periodo la mitad del periodo de recolección. La lógica de los timer se muestra en la siguiente Figura.

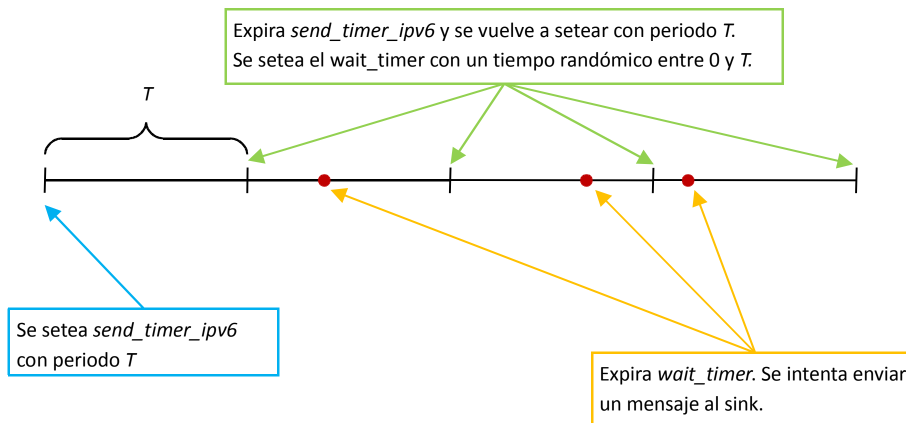


Figura 26: Diagrama de tiempo que indica cuando se intenta enviar mensajes al sink

El período T= (Período de Recolección) / 2

De esta manera logramos que los datos se envíen en base a un tiempo aleatorio, para evitar colisiones y además a la frecuencia deseada.

Cada vez que el wait_timer expira, si se cumple que el Rank del nodo es distinto de infinito y además si el buffer no está vacío, entonces se enviará el elemento más antiguo del buffer al sink.
De esta manera se intentará enviar datos del buffer a una frecuencia del doble de la de recolección. Por lo tanto, si en algún momento el buffer se llena, porque el Rank del nodo es infinito, cuando se retome la comunicación con el sink no habrá problema para vaciarlo.

Archivos del código modificados y creados:

core\lib\circularbuf.h	creado	circular de wikipedia y se modifica
examples\ipv6\ContikiWSN-Application\udp-sender.c	modificado	Se inicializa el buffer circular indicando la cantidad máxima de datos sensados que se podrán almacenar
\Makefile.include	modificado	Se indica que el archivo circularbuf.c de la librería debe ser compilado
examples\ipv6\ContikiWSN-Application\udp-sink.c	modificado	Se crea la lógica de bufferear los datos sensados y enviarlos cuando sea conveniente.
core\sys\etimer.c	modificado	Se agrega una función que cambia el momento en el que expira un etimer pero sin cambiar el proceso al que apunta.
core\sys\etimer.h	modificado	

6.7 Sincronización de la hora en la red

En la aplicación original rpl-collect, cuando el nodo Sink recibe un mensaje de un Sender, le agrega la hora al mensaje y se toma esta hora como la hora en que el nodo Sender recolectó los datos.
Desde que el nodo Sender envía los datos hasta que el Sink los recibe, en el peor de los casos se puede demorar unos pocos segundos, lo que es perfectamente tolerable.

El problema es que con la aplicación nueva que recupera datos viejos esta lógica no sirve porque si se envía al Sink datos que fueron sensados hace 2 horas, cuando este los reciba les va a poner la hora actual. En la aplicación ContikiWSN los nodos Sender guardarán los mensajes con la hora en la que se recolecto. El problema que se genera es cómo hacer para que la hora de todos los nodos Sender este sincronizada con la hora del Sink.

Para esto se usarán los mensajes Trickle. La metodología es que cada nodo antes de enviar un mensaje Trickle obtiene su hora actual y la agrega a dicho mensaje. Por otro lado cuando un nodo recibe un mensaje Trickle con versión más actualizada ajusta su hora con la recibida en el mensaje. Puede haber unos mili segundos de imprecisión debido al tiempo de la transmisión pero esto es perfectamente tolerable.

Por lo tanto cada vez que el Sink envía una nueva versión de mensaje Trickle disemina su hora a todos los nodos. Si algún nodo está aislado no habrá problema, ya que cuando vuelva a tomar contacto con la red, uno de sus nodos vecinos le enviará un mensaje Trickle con su hora actual, la cual esta sincronizada con la hora del Sink.

Para representar la hora se utilizará el tiempo UNIX, o sea la cantidad de segundos transcurridos desde la medianoche del 1 de enero de 1970. Se utilizará la función `clock_seconds` de Contiki la cual devuelve los segundos desde que el microcontrolador se prendió en una variable de 32bits. A este valor se le sumará un offset el cual representa la diferencia entre la hora sincronizada de la red y en la que se prendió el microcontrolador.

Imaginemos que un nodo recibe un mensaje Trickle con una versión superior a la conocida, entonces calcula el offset con la siguiente fórmula:

Offset = Hora recibida por Trickle – Hora del microcontrolador

Observar que la Hora del microcontrolador se calcula con la función `clock_seconds`. Luego de tener el Offset se puede calcular la hora sincronizada por Trickle en cualquier momento con la siguiente fórmula:

Hora sincronizada = Hora del microcontrolador + Offset

Todas las variables son de 32bits. No habría problema con que se produzca overflow ya que con 32 bits se pueden representar hasta 136 años. Al tomarse como punto de partida el año 1970 podemos representar la hora hasta el año 2106.

Se precisa sincronizar la hora de la red periódicamente con la del Sink, por lo tanto en la aplicación ContikiWSN el nodo sink enviará una nueva versión de mensaje Trickle cada 24hs. Como se utilizan 16 bits para el número de versión de mensajes Trickle no hay problema de overflow, ya que con 16bits se pueden representar 65536 valores. Esto alcanzaría para enviar una nueva versión por día durante 179 años antes de hacer overflow.

Archivos del código modificados

examples\ipv6\ContikiWSN-Application\udp-sender.c	modificado	Los nodos senders envían en el mensaje la hora en la que se sensaron los datos.

6.8 Utilización de la aplicación Shell

Se utilizará la aplicación auxiliar Shell en ContikiWSN y se le agregarán comandos nuevos. Se deberá modificar el archivo makefile del ContikiWSN para que incluya la aplicación shell. Esto se hace agregando en la variable APPS "serial-shell". Luego se deberán quitar los comandos que son para Rime y se agregarán los que son para ContikiWSN.

Para agregar comandos se debe crear un archivo .c y .h con el nombre del comando en la carpeta apps\shell. A continuación se muestra un ejemplo de como se deben implementar los comandos.

apps\shell\shell-comando-de-prueba.c
apps\shell\shell-comando-de-prueba.h

```
/*---------------------------------------------------------------------------*/

/* Archivo shell-comando-de-prueba.c */

/*Se declara el proceso que se utilizará para procesar el comando*/
PROCESS(shell_comando_de_prueba_process, "comando de prueba");

/* Se define el comando con SHELL_COMMAND(name, command, description, process)
   name: corresponde al nombre del comando
   command: Indica como debe ser llamado el comando en el shell.
   description: La descripción que se hará del comando cuando se ejecuta el help
   process: Indica cual es el proceso asociado al comando*/
SHELL_COMMAND(shell_comando_de_prueba_command,
              "prueba",
              "prueba: Es el comando de prueba",
              &shell_comando_de_prueba_process);
/*---------------------------------------------------------------------------*/

/* Se define el protothread del proceso shell_comando_de_prueba_process */
PROCESS_THREAD(shell_comando_de_prueba_process, ev, data)
{

    PROCESS_BEGIN();

    /* Aquí se debe procesar el comando.
       En caso de que el comando precise manejar parámetros,
       en la variable data se encuentra el puntero al primer carácter
       ingresado luego de "prueba"*/


    PROCESS_END();
}
/*---------------------------------------------------------------------------*/

/* Con la siguiente función se agrega el comando a la lista de comandos disponibles */
void
shell_ipv6_set_time_init(void)
{
  shell_register_command(&shell_comando_de_prueba_command);
}
```

```
/*---------------------------------------------------------------------------*/
```

El archivo shell-comando-de-prueba.h sólo debe declarar la función `void shell_ipv6_set_time_init(void)` para que pueda ser accedida públicamente.

Los comandos que se implementarán para ContikiWSN serán los siguientes:

time[n]: Se actualiza el tiempo de la red. Donde n son los segundos a partir de 1970. Tener en cuenta que el sink luego de actualizar su hora enviará un mensaje trickle incrementando su número de versión.

confnodes[n]: Se actualiza cada cuanto tiempo deben recolectar datos los nodos senders. Luego que el sink recibe este comando enviará un mensaje trickle incrementando su número de versión. Tener presente que este comando configura el tiempo de recolección pero no da la orden de comenzar a recolectar datos.

collect: En rpl-collect los nodos senders siempre están recolectando y enviando datos al sink. En cambio en la aplicación nueva todos los nodos senders empezarán ociosos y con este comando se le envía la orden de que empiecen a recolectar y enviar datos al sink. Luego de recibir el comando el sink enviará un mensaje trickle incrementando su número de versión.

stopcollect: El sink al recibir este comando envía una orden a los nodos senders para que dejen de recolectar y enviar datos al sink. Luego de recibir el comando el sink enviará un mensaje trickle incrementando su número de versión.

netprint: El comando es idéntico al de rpl-collect.

mac[n]: El comando es idéntico al de rpl-collect.

Archivos del código creados y modificados

Archivo	Estado	Comentario
apps\shell\shell-ipv6-confnodes.c	creado	Se crea el comando confnodes
apps\shell\shell-ipv6-confnodes.h	creado	
apps\shell\shell-ipv6-mac.c	creado	Se crea el comando mac
apps\shell\shell-ipv6-mac.h	creado	
apps\shell\shell-ipv6-net-print.c	creado	Se crea el comando netprint
apps\shell\shell-ipv6-net-print.h	creado	
apps\shell\shell-ipv6-set-time.c	creado	Se crea el comando set-time
apps\shell\shell-ipv6-set-time.h	creado	
apps\shell\shell-ipv6-start-collect.c	creado	Se crea el comando start-collect
apps\shell\shell-ipv6-start-collect.h	creado	
apps\shell\shell-ipv6-stop-collect.c	creado	Se crea el comando stop-collect
apps\shell\shell-ipv6-stop-collect.h	creado	
apps\shell\shell.h	modificado	Se incluyen los headers de los comandos creados
apps\shell\Makefile.shell	modificado	Se incluyen los archivos de los comandos creados
examples\ipv6\ContikiWSN-Application\udp-sink.c	modificado	Se inicializa el shell
examples\ipv6\ContikiWSN-Application\udp-sink.c	modificado	Se crea un proceso para enviar una versión nueva de mensaje trickle cada 24hs.
examples\ipv6\ContikiWSN-Application\udp-sender.c	modificado	Se inicializa el shell
examples\ipv6\ContikiWSN-Application\collect-common.c	modificado	Se inicializan los comandos.

6.9 Interfaz collect-view

Collect-view es un software implementado en java por los creadores de Contiki. Recibe como input los datos por serial del Sink, los procesa y los despliega de manera gráfica. También puede enviar comandos al sink, mediante pulsaciones de botones.

La interfaz original de este software está pensada para el firmware sky-shell. Con rpl-collect los comandos que se envían al presionar los botones no son reconocidos.

Para la aplicación ContikiWSN.

- Se tendrá que cambiar el software que carga el collect-view a los nodos.
- Se tendrá que cambiar los botones y los campos para ingresar valores según las nuevas necesidades
- Se tendrá que cambiar los comandos que se envían al sink cuando se presionan los diferentes botones.
- Se tendrá que cambiar las instrucciones “Quick Startup Instructions” para que sean coherentes a la nueva aplicación.

Figura 27: Interfaz collect-view adaptada con ContikiWSN

Cuando se presione el botón *Program Nodes…* se tendrá que cargar al mote que esté conectado al USB el firmware del nodo sink. Precaución: Para que cargue la última versión antes de compilar el collect-view con ant run se deberá ejecutar ant clean. De manera que si existe un archivo binario viejo del sink se borrará.

Se sustituirá el botón *Send stop to nodes* por *SetTime* y se quitarán campos para agregar datos innecesarios.

Cuando se presionen los botones se ejecutarán los siguientes comandos:

```
Start Collect: "~K" + "killall" + "mac 0" + "collect"

Stop Collect: "~K", "killall", "stopcollect"

Set Time: “time” + <Java Time>

Send command to nodes: "confnodes " + <Report interval>
```

Las instrucciones para la nueva aplicación son que primero se programe el nodo sink y se conecte al serial. Luego se setee el tiempo de la red y se configure el periodo de muestreo. Finalmente se puede comenzar la colección de datos.

Archivos del código modificados

Archivo	Estado	Comentario
tools\collect-view\src\se\sics\contiki\collect\CollectServer.java	modificado	Se utiliza el firmware udp-sink de la aplicación ContikiWSN en vez de shell-sky cuando se presione el Botón “Program”
tools\collect-view\src\se\sics\contiki\collect\gui\NodeControl.java	modificado	Se modifica la interfaz gráfica de CollectView y los comandos que se envían al nodo sink cuando se presionan los botones.
tools\collect-view\build.xml	modificado	Se compila ContikiWSN y genera el archivo udp-sink.ihex que será utilizado para cargar en el sink

6.10 Espacio libre luego de la aplicación

Como se analizó en la sección “Optimización de memoria RAM y ROM” la memoria disponible de los nodos antes de comenzar a implementar la nueva aplicación era la siguiente:

En el nodo sink se usan 38KB de memoria Flash y 6.6KB de memoria RAM.

En el nodo sender se usan 39.7KB de memoria Flash y 6.7KB de memoria RAM.

Mientras que la capacidad de memoria del microcontrolador msp430 F1611 es de 48KB de Flash y 10KB de RAM.

Luego de implementar la aplicación ContikiWSN el msp430-size despliega los siguientes valores:
En el nodo sink se usan 42KB de memoria Flash y 7.1KB de memoria RAM.

En el nodo sender se usan 44KB de memoria Flash y 7.1KB de memoria RAM.

Al nodo sender hay que agregarle la memoria RAM que se reserva con malloc para el buffer de datos sensados. Como en el buffer se pueden guardar hasta 10 elementos, la memoria RAM reservada es de 504bytes.

Finalmente:

En el nodo sink se usan 42KB de memoria Flash y 7.1KB de memoria RAM.

En el nodo sender se usan 44KB de memoria Flash y 7.6KB de memoria RAM.

6.11 Software utilizado para el desarrollo

- Ambiente de desarrollo Code::Blocks,
- Compilador GCC, toolchain para el microcontrolador MSP430 v3.2.3
- Código fuente del Sistema Operativo Contiki v2.5
- Winmerge. Se utilizó para comparar archivos y carpetas.
- SVN. Se llevó el control de las versiones a través de Subversion [31].
- Google Code. Se utilizó como repositorio del código.
- Instant Contiki, es una maquina virtual lista para compilar Contiki y sus aplicación de java. Los creadores de Contiki la crearon porque recibieron muchas quejas de problemas de compatibilidad.
- COOJA. Simulador de Contiki.
- Collect-view. Aplicación para mostrar de forma gráfica los datos recolectados por el sink y también para enviarle comandos a este. Está implementada en java por el equipo de Contiki.

Capítulo 7: Pruebas y Evaluación

7.1 Introducción

Para poder estudiar el consumo del los nodos al utilizar el sistema operativo Contiki se realizaron pruebas de laboratorio y simulaciones. También se estudió Energest que es un mecanismo de estimación de consumo que ofrece Contiki. De esta manera se pudo caracterizar el consumo al utilizar Contiki y establecer proyecciones del mismo a partir de variaciones de parámetros de ContikiMAC. En el simulador se realizó un estudio del consumo de la aplicación desarrollada ContikiWSN y se comparó con la aplicación rpl-collect que tiene funcionalidades similares.

Para realizar la evaluación del consumo al utilizar Contiki se realizaron mediciones en el laboratorio que consistieron en armar una red con dos nodos recolectores de datos y un nodo sink. Se ejecutó la aplicación de recolección de datos que viene con el sistema, rpl-collect. Los dos nodos recolectores de datos se conectaron a los canales de un osciloscopio a través de sus baterías de alimentación de manera de poder ver la corriente consumida. Relevamos curvas de la corriente consumida por cada nodo y posteriormente se proceso esta información.

Para interpretar estos datos se comparó con la simulación realizada en COOJA de el mismo código que se grabó en la prueba de laboratorio. Para poder describir la comunicación entre los nodos se utilizaron varios plugins de COOJA que permiten analizar la simulación desde distintos puntos de vista. De este modo fue posible caracterizar el tráfico relevado en el laboratorio.

7.2 Herramientas Utilizadas

7.2.1 Introducción

Las principales herramientas de análisis que brinda el simulador COOJA para realizar análisis del funcionamiento de una simulación, “Log Listener”, “Radio Listener”, “Timeline” y “Code Watcher”. Se trata de distintas formas de presentar la información de simulación y además herramientas de ejecución paso-a-paso y marcado de eventos de simulación. La simulación se controla desde el panel de control permitiendo pararla, detenerla o avanzar en intervalos de un ms.

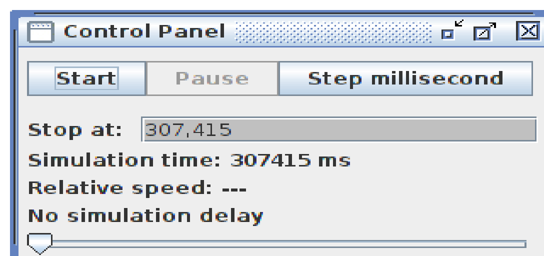


Figura 28: Control Panel

En el panel se muestra el tiempo transcurrido desde el inicio de la transmisión, el instante de la

última parada y la velocidad relativa que solo tiene sentido si se eligió algún retardo de la simulación respecto al tiempo que transcurriría en una ejecución real. Esta opción es útil si se desea observar más lentamente la simulación para detenerla.

7.2.2 Log Listener

El "Log Listener" muestra la información que los nodos imprimen a través del comando printf(). El uso típico es depurar el funcionamiento de un programa. Se utilizó para poder visualizar variables del código que afectan la comunicación entre los nodos y que de otro modo no son accesibles. La información se presenta en 3 columnas: *Time ms*, *Mote*, *Message*. La columna *Messages* contiene el resultado de la líneas que los nodos hayan impreso con printf().

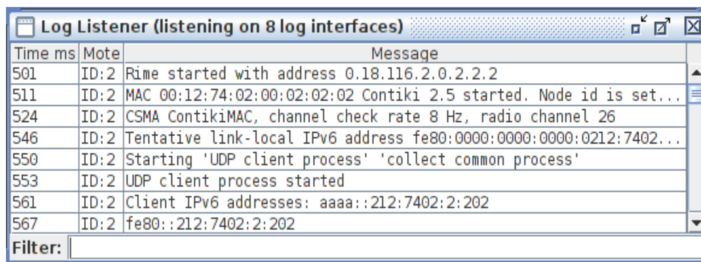


Figura 29: Log Listener

Dada la gran cantidad de información que se muestra es de utilidad poder seleccionar que se quiere visualizar. Con el campo *Filter* podemos escribir una palabra para utilizarla como criterio de filtrado de los mensajes escritos en las columnas *Mote* o *Messages*. Si filtramos por ejemplo con *ID:4* podremos ver exclusivamente los mensajes de nodo 4. En el ejemplo vemos las líneas que contienen el criterio *strobes=*, de este modo se ven solo las líneas que la contienen.

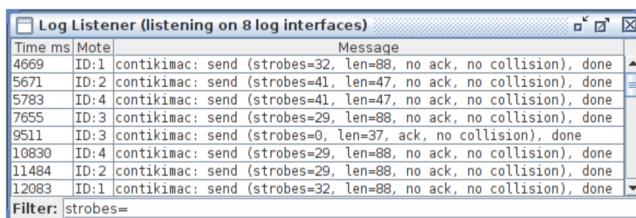


Figura 30: Ejemplo de filtrado en Log Listener

Realizando click-derecho sobre una fila dentro de la ventana permite visualizar las opciones disponibles:

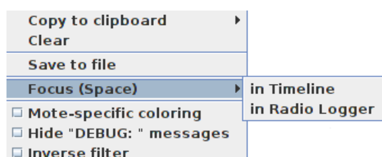


Figura 31: Opciones del Log Listener

Como esta información es parcial es necesario poder relacionarla con la que está disponible en otros pluggins. Para esto disponemos de la opción *Focus (Space)* que aparece en obscuro permite marcar la

ocurrencia de ese evento en la *Timeline* o en *Radio Logger*, o bien el evento producido en el instante más próximo. Si se elige la opción *Mote-specific coloring* COOJA diferenciará los mensajes de los distintos nodos asignándoles un color.

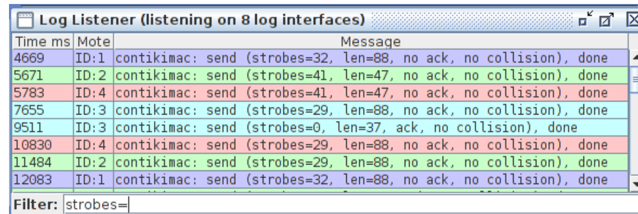


Figura 32: Diferenciación de nodos por colores

En nuestro caso nos fue muy útil para identificar la cantidad de paquetes que componían una ráfaga, si se trata de un mensaje ACK, si se produjo o no una colisión. La opción *Save to file* guarda la información mostrada en la ventana en un archivo de texto para su posterior análisis.

7.2.3 Radio Logger

En el Radio Logger se dispone de información vinculada a la interfaz de aire IEEE 802.15.4. Las columnas contienen el tiempo de simulación en ms, el ID del nodo transmisor y del receptor y el dato que transmite. La figura siguiente ilustra una la ventana de este plugin.

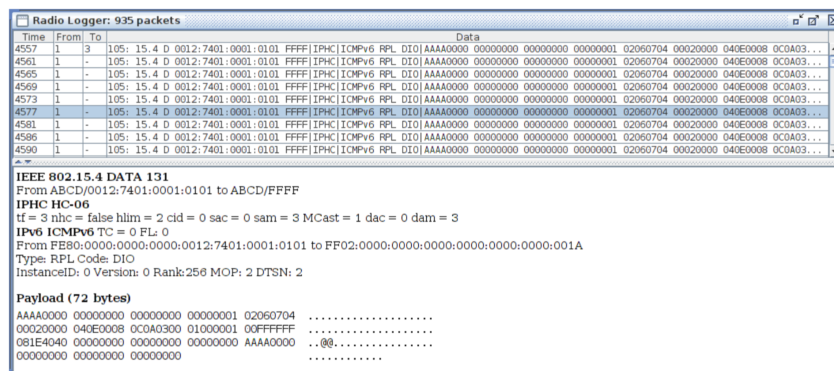


Figura 33: Radio Logger

En primer fila de la imagen se ve una trasmisión unicast de nodo 1 al nodo 3. Mientras que en las demás se ven trasmisiones broadcast. En el cuadro inferior se ven los detalles del paquete marcado en la línea obscura (time=4557). En esta ventana también están disponibles opciones con click-derecho:

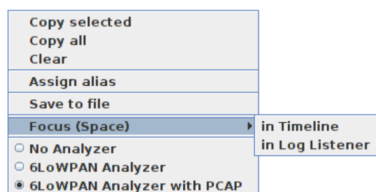


Figura 34: Opciones del Radio Logger

Las opciones *Save to File* y *Focus* cumplen la misma función que en casos anteriores. Las tres

opciones inferiores permiten elegir si se quiere visualizar algo en el cuadro inferior. La opción *No Analyzer* deja el cuadro inferior vacío, la opción *6LowPAN Analyzer* muestra detalles de los paquetes correspondiente al protocolo *6LowPAN*. La tercera opción permite que además de mostrar esta información, que se guarde en un archivo con extensión .pcap que pueda abrirse con el analizador de protocolos Wireshark. La figura 40 fue tomada con esta última opción seleccionada.

7.2.4 Timeline

En esta ventana se presenta de manera simultánea la actividad de todos los nodos en relación a la actividad de la radio, los leds y los watchpoints que pueda agregar el usuario. A la izquierda se puede elegir que información se quiere mostrar en la línea de tiempo de la derecha.

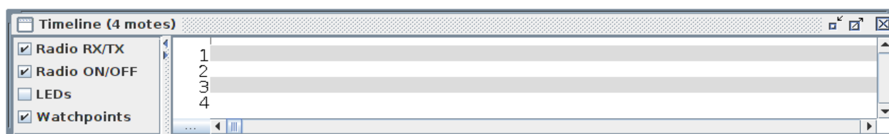


Figura 35: Timeline

La información relativa a cada nodo aparecerá en franjas horizontales diferenciada por colores. En la figura 42 se observa un ejemplo típico. En gris obscuro se indica que la radio está encendida. Si marcamos con el ratón en la franja blanca que está por encima de la franja del nodo 1 tendremos la información del tiempo. Al deslizarlo horizontalmente veremos que el valor cambiará y el valor que está entre paréntesis cambiará indicando el lapso transcurrido desde el instante correspondiente al instante que se marcó inicialmente y el actual.

Figura 36: Ejemplo de Timeline

En el ejemplo se midió el tiempo transcurrido desde el inicio de la trasmisión realizada por el nodo 2 132,5 ms. En la figura 42 se ve en azul una transmisión broadcast realizada por el nodo 2 y la recepción realizada por el nodo 3 en verde. Esta funcionalidad fue clave para medir los tiempos nominales que surgen del código para luego compararlos con los valores surgidos del laboratorio. Como se ilustra en la figura 43, también se dispone de opciones con el botón derecho del ratón,

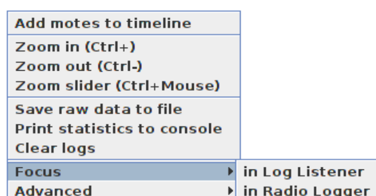


Figura 37: Opciones de Timeline

Al realizar zoom podemos ver con más detalle la simulación de manera de poder tener una visión más detallada y poder extraer mejor información sobre los eventos de simulación. En la imagen se observa que la radio está encendida (línea gris obscura) y que se está realizando una trasmisión. Al presionar el botón

izquierdo del ratón inmediatamente de finalizada la línea azul, encontramos obtenemos el tiempo de simulación en que se termino la transmisión.

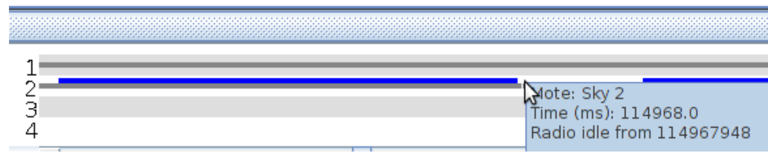


Figura 38: Propiedades de Timeline

Toda la información representada en la Timeline se puede guardar en un archivo de texto utilizando la opción *Save raw data to file* para luego ser procesada independientemente si se quiere.

7.3 Simulation Visualyzer

Es la ventana en la que se ve la disposición de los nodos y la comunicación entre ellos. En ella se puede configurar parámetros del canal entre otros y de visualización.

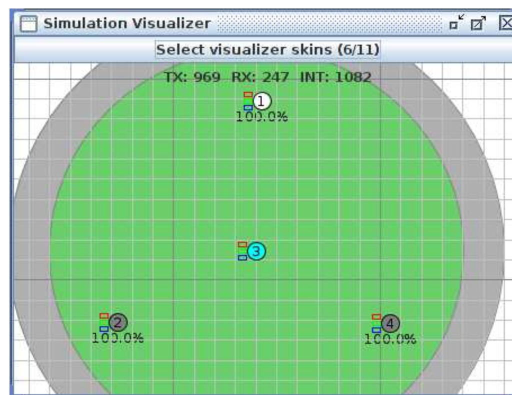


Figura 39: Simulation Visualyzer

Las opciones disponibles se muestran en la figura de la derecha. Presionando el botón derecho del ratón sobre uno de los nodos aparecen opciones adicionales para ver información de cada nodo y herramientas para debuggear.

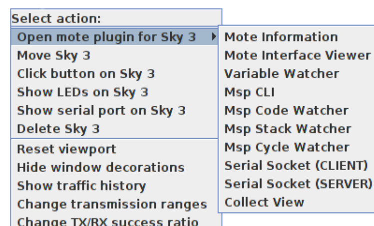


Figura 40: Opciones adicionales de Simulation Visualyzer

En nuestro caso la funcionalidad más utilizada fue Msp Code Watcher. Fue de suma utilidad para

estudiar la ejecución del código tanto de Contiki como de la aplicación.

7.3.1 Msp Code Watcher

Es la interfaz que permite visualizar el código que se ejecuta en un nodo particular y realizar ejecución instrucción por instrucción tanto del código en C como en assembler y además agregar watchpoints. Como se muestra en la figura 47, la ventana se divide en tres partes. La de la derecha que muestra el código en C, la de la izquierda muestra el código en assembler correspondiente a la instrucción C y finalmente en el medio los watchpoints que fueron agregados por el usuario.

Se puede ejecutar una sesión del Msp Code Watcher por cada nodo simulado. Esto permite estudiar por separado nodos distinto ya sea porque ejecutan distintas aplicaciones o porque están situados en distintos lugares de la red.

Para agregar un watchpoints es necesario presiona el botón derecho del ratón en la línea del código C en la que se quiere agregarlo. Se le dará un nombre descriptivo y se le asociará un color para ser identificado. Ese watchpoint aparecerá en el cuadro central. En la columna *info* el nombre asociado con su color. Luego se informará la dirección de memoria y la fila y el archivo que el que se creó.

Cada vez que la simulación pase por esa línea de código se detendrá en caso de que esté marcada la columna *stop*. En caso contrario, la simulación continuará y dejará en la Timeline una marca del color elegido al crear el watchpoint.

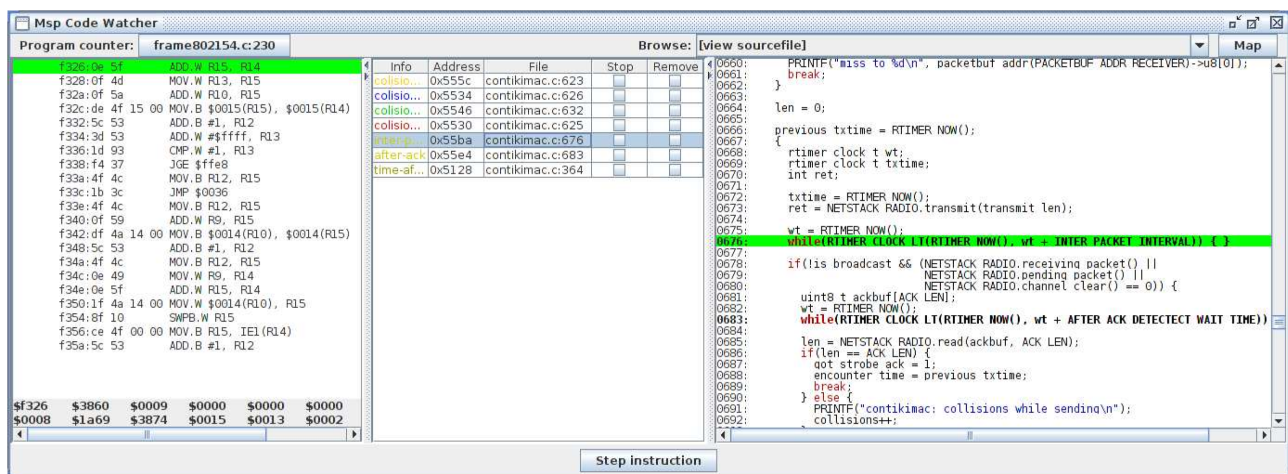


Figura 41: MSP Code Watcher

En la figura se puede observar en verde las línea de código en la que se agregó el watchpoint en los paneles laterales y en el panel central los datos asociados al mismo. En este caso se trata de un ciclo while(). Para visualizarlo en la Timeline es necesario que esté seleccionada la visualización. Contando las marcas es posible en este caso contar cuantas veces se ejecutó el ciclo y el tiempo transcurrido entre dos pasadas consecutivas.

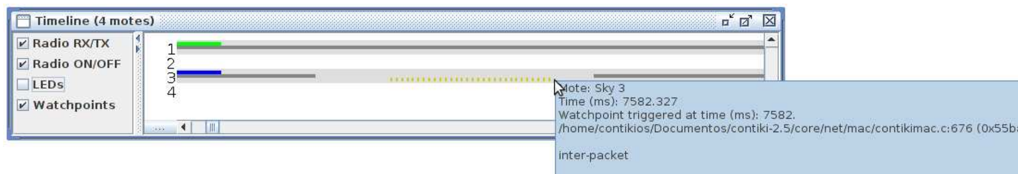


Figura 42: Watchpoints en la Timeline

Pinchando con el ratón al lado de una de esas marcas aparece un recuadro en azul que presenta información que permite confirmar de que watchpoint se trata. También se puede ver la información del archivo y de la línea en la que se lo colocó, el instante en el que ocurrió en términos de tiempo de simulación y por supuesto permite identificar el mote.

7.4 Estimación de energía

Dadas las características de los nodos involucrados en redes de sensores inalámbricas es de suma utilidad tener la posibilidad de poder tener información en tiempo real del desempeño en términos de consumo de nuestra red. Para esto es que el equipo de Contiki desarrolló Energest [32] que es una herramienta software de estimación de consumo. A continuación presentamos sus principales características.

7.5 Energest

La idea central de Energest consiste en llevar la cuenta de cuanto tiempo está la radio y la CPU funcionando y en que estado; encendido, apagado y en el caso de la CPU en estado de bajo consumo. Los autores consideraron los siguientes casos,

CPU	Consumo asociado a la CPU con la radio apagada
LPM	Consumo asociado a la CPU en modo bajo consumo y radio apagada
Tx	Consumo asociado a la actividad de la radio y la CPU mientras transmite
Rx	Consumo asociado a la actividad de la radio y la CPU mientras recibe
Ci	Consumo asociado a otros componentes como los Leds y sensores

Los casos Tx y RX incluyen el consumo correspondiente al microprocesador y a la radio. Cada vez que un componente de hardware es encendido el mecanismo de estimación guarda una marca temporal. Luego, cuando el componente se apaga, se releva el tiempo transcurrido en ese estado y se añade al tiempo total que estuvo encendido el componente. El mecanismo de estimación mantiene una lista de los cuatro tipos de consumo y el tiempo que estuvieron encendidos. Con el acumulado de cada tiempo es posible relevar la energía total en un intervalo de tiempo dado a partir de la ecuación siguiente:

$$\frac{E}{V} = I_m.t_m + I_l.I_l + I_t.I_t + I_r.I_r + \sum I_{c_i}.I_{c_i}$$

El mecanismo produce un pequeño overhead de procesamiento. El código que guarda el tiempo es muy pequeño. Según los autores alcanzan 11 ciclos de reloj para guardar la marca temporal previa al encendido del componente, y 20 ciclos de reloj para actualizar el tiempo total luego de que el componente se apaga.

La evaluación del consumo se realizó equipando cada nodo con condensadores de 1F. El condensador se carga encendiendo el nodo de manera que se conecta la batería al condensador. Cuando se desconecta la batería, el nodo funciona con la energía contenida en el condensador. De este modo el condensador puede alimentar el nodo de prueba durante unos minutos, dependiendo del consumo de energía del software que se ejecute.

La prueba consistió en ejecutar en los nodos tres códigos distintos emulando el uso estándar de una red de sensores, encendiendo la radio y los LED a intervalos regulares y, enviando paquetes por la radio. Los intervalos para los tres programas se configuraron de manera distinta y dejaron funcionar los nodos hasta que se apagaron. Este mecanismo de prueba implica la realización varias tareas manuales que constituyen fuentes de inexactitud, a tal punto que plantea en futuro la necesidad de mejorar el procedimiento de validación.

7.5.1 Deducción de las ecuaciones de cálculo de consumo de collect-view

Collect-View es un programa escrito en java que se ejecuta en el PC que está conectado al nodo sink y cuya función es la mostrar la información recibida por este nodo tanto sea de la información los sensores como información relativa al consumo.

Para calcular el consumo en un intervalo de tiempo T collect-View tiene en cuenta los distintos tipos de consumos que se producen, en base a lo informado por Energest. Estos están asociados a la actividad de la CPU y de la radio. Para la CPU se distinguen los estados de bajo consumo, LPM y la actividad de procesamiento. A su vez cuando la CPU está activa puede hacerlo con la radio apaga o encendida y en este último caso sea trasmitiendo o recibiendo. Los tipos de consumo considerados en Collect-View son los que maneja Energest salvo los relativos al consumo de los sensores y los leds.

En consecuencia los tiempos involucrado el cálculo realizado por Collect-View son:

$$\Rightarrow \begin{cases} T_{CPU} \;-\; \textit{Tiempo asociado al estado de CPU activa} \\ T_{LPM} \;-\; \textit{Tiempo asociado el estado de bajo consumo} \\ T_{LISTEN} \;-\; \textit{Tiempo asociado al estado de trasmición} \\ T_{TRANSMIT} \;-\; \textit{Tiempo asociado al estado de recepción} \end{cases}$$

Se cumplen las siguientes ecuaciones:

$$\Rightarrow \begin{cases} T = T_{CPU} + T_{LPM} \\ T_{CPU} > T_{LISTEN} + T_{TRANSMIT} \end{cases}$$

La desigualdad es estricta debido a que antes de encender la radio existe un procesamiento previo. Al analizar Energest se vio que la idea es multiplicar los tiempos medidos en cada nodo para cada estado por valores instantáneos de referencia. En el código los autores utilizan los siguientes valores:

$$\Rightarrow \begin{cases} \textit{Voltaje de la fuente}: \; V = 3\text{V} \\ CPU: \; P_{0,CPU} = 1{,}800.V \;\; mW \\ LPM: \; P_{0,LPM} = 0{,}0545.V \;\; mW \\ Tx: \; P_{0,TRANSMIT} = 17{,}7.V \;\; mW \\ Rx: \; P_{0,LISTEN} = 20{,}0.V \;\; mW \end{cases}$$

Es interesante observar que para los dos primeros se trata de los valores de las hojas de datos

detallados en la Tabla 1, mientras que los dos segundos son valores que está por debajo del valor nominal de dicha tabla. Para calcular el acumulado de cada estado es necesario tener en cuenta la porción del tiempo total que se estuvo en ese estado. De esa manera se obtiene:

$$\Rightarrow \begin{cases} E_{CPU-total} = T_{CPU} \cdot P_{0,CPU} \ [Joules] \\ E_{LPM-total} = T_{LPM} \cdot P_{0,LPM} \ [Joules] \\ E_{TRANSMIT-total} = T_{TRANSMIT} \cdot P_{0,TRANSMIT} \ [Joules] \\ E_{LISTEN-total} = T_{LISTEN} \cdot P_{0,LISTEN} \ [Joules] \end{cases}$$

De manera que la energía total consumida en el intervalo T considerando todos los estados es:

$$E_{total} = T_{CPU} \cdot P_{0,CPU} + T_{LPM} \cdot P_{0,LPM} + T_{LISTEN} \cdot P_{0,LISTEN} + T_{TRASMIT} \cdot P_{0,TRASMIT} \ [Joules]$$

Los autores calculan el promedio temporal como sigue:

$$P_{promedio} = \frac{T_{CPU} \cdot P_{0,CPU} + T_{LPM} \cdot P_{0,LPM} + T_{LISTEN} \cdot P_{0,LISTEN} + T_{TRASMIT} \cdot P_{0,TRASMIT}}{T_{CPU} + T_{LPM}} \ [mW]$$

Otra forma de analizar los datos es calcular la proporción del consumo de cada estado en el total. Para esto multiplicamos por $T_{est_X} / (T_{CPU} + T_{LPM})$ los valores de referencia de cada estado est_X. En cada caso se puede expresar de la siguiente manera:

$$P_{CPU} = P_{0,CPU} \cdot \frac{T_{CPU}}{T_{CPU} + T_{LPM}} = \frac{T_{CPU} \cdot P_{0,CPU}}{T_{CPU} + T_{LPM}} \ mW$$

$$P_{LPM} = P_{0,LPM} \cdot \frac{T_{LPM}}{T_{CPU} + T_{LPM}} = \frac{T_{LPM} \cdot P_{0,LPM}}{T_{CPU} + T_{LPM}} \ mW$$

$$P_{LISTEN} = P_{0,LISTEN} \cdot \frac{T_{LISTEN}}{T_{CPU} + T_{LPM}} = \frac{T_{LISTEN} \cdot P_{0,LISTEN}}{T_{CPU} + T_{LPM}} \ mW$$

$$P_{TRASMIT} = P_{0,TRASMIT} \cdot \frac{T_{TRASMIT}}{T_{CPU} + T_{LPM}} = \frac{T_{TRASMIT} \cdot P_{0,TRASMIT}}{T_{CPU} + T_{LPM}} \ mW$$

En todos los casos el término $T_{est_X} \cdot P_{0,est_X}$ corresponde a la energía total (en Joules) consumida durante el intervalo T en el estado *est*$_X$. Si sumáramos estos valores recién calculados obtenemos el valor de potencia promedio para el intervalo, $P_{promedio}$ calculado recién.

7.5.2 Análisis del consumo en base a medidas de laboratorio

7.6 Metodología de análisis del consumo

Como se adelantó en la introducción, para analizar el consumo del mote se montó una maqueta en el laboratorio con dos nodos y un nodo base. Conectamos un mote a cada uno de los dos canales del osciloscopio, conectamos una resistencia en serie con una de las dos pilas de alimentación para medir la caída de tensión Vr. Con esa tensión y el valor de la resistencia calculamos la corriente y la potencia consumida por el nodo. En todos los casos se trata de la potencia consumida por la placa en su conjunto.

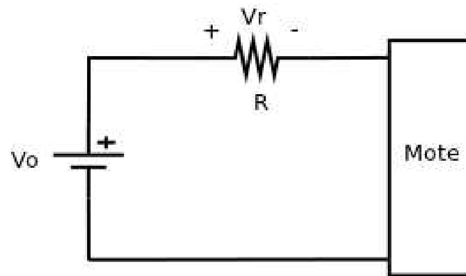


Figura 43: Circuito de medida

Los datos medidos se ilustran en la siguiente figura:

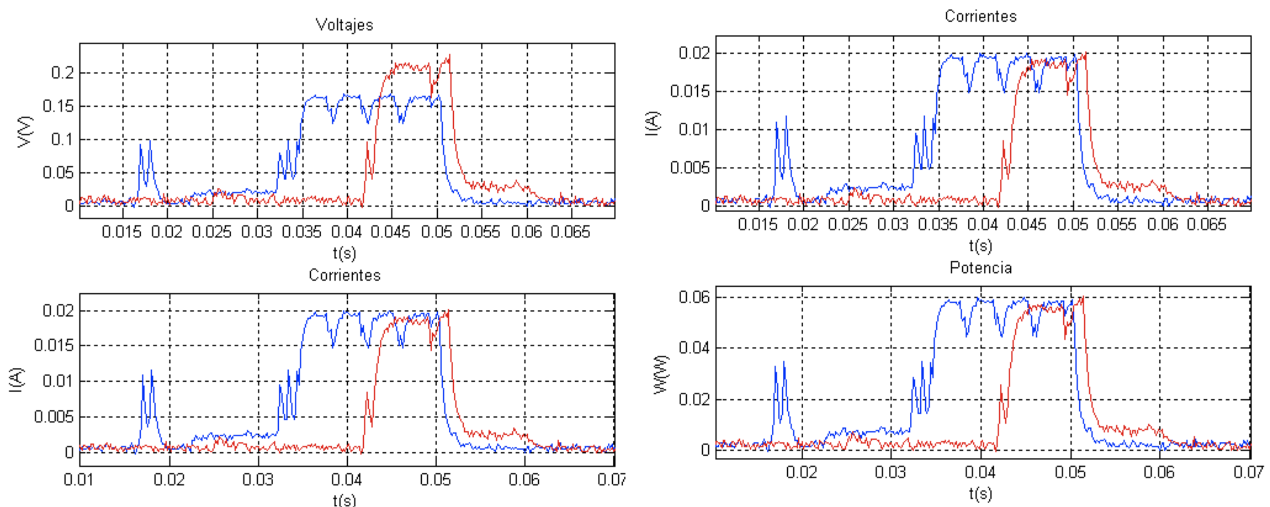


Figura 44: Datos procesados numéricamente

a) b)

En la figura 50 a) se ilustra el voltaje tal cual se midió y la correspondiente corriente. Cabe notar que los voltajes son notoriamente distintos. Esto se debe a que se utilizaron resistencias distintas en cada nodo. En la figura 50 b) se muestra la corriente con la potencial correspondiente.

En este ejemplo se muestra una transmisión unicast en la que se observa claramente el apagado y encendido de la radio para trasmitir cada paquete. El tiempo que está en la parte plana de la curva corresponde a la puesta en el canal de cada uno de los paquetes. Entre cada uno de los envíos se observa la descarga y carga del condensador de la radio. El momento en el que se observa la carga del condensador constituye un mínimo de consumo. Podemos aproximar la sucesión de esos mínimos por una recta horizontal. Mientras que el máximo estará dado por la potencia necesaria para poner en el aire cada paquete.

Para evaluar una transmisión completa es necesario además tener en cuenta las CCA's iniciales de cada transmisión.

En definitiva, para estimar el consumo es necesario considerar la cantidad de paquetes, el tiempo necesario para poner cada uno en el canal, el consumo en esa situación y el consumo que se produce durante la descarga y la carga del condensador.

Las medidas realizadas en los dos canales se realizaron con resistencias distintas. Por este motivo se observa una diferencia en los valores de tensión. Al calcular la corriente se utilizó el valor de la resistencia verificado con un multímetro en lugar del valor nominal. La gráfica de la potencia se obtiene utilizando el valor nominal de la fuente 3V.

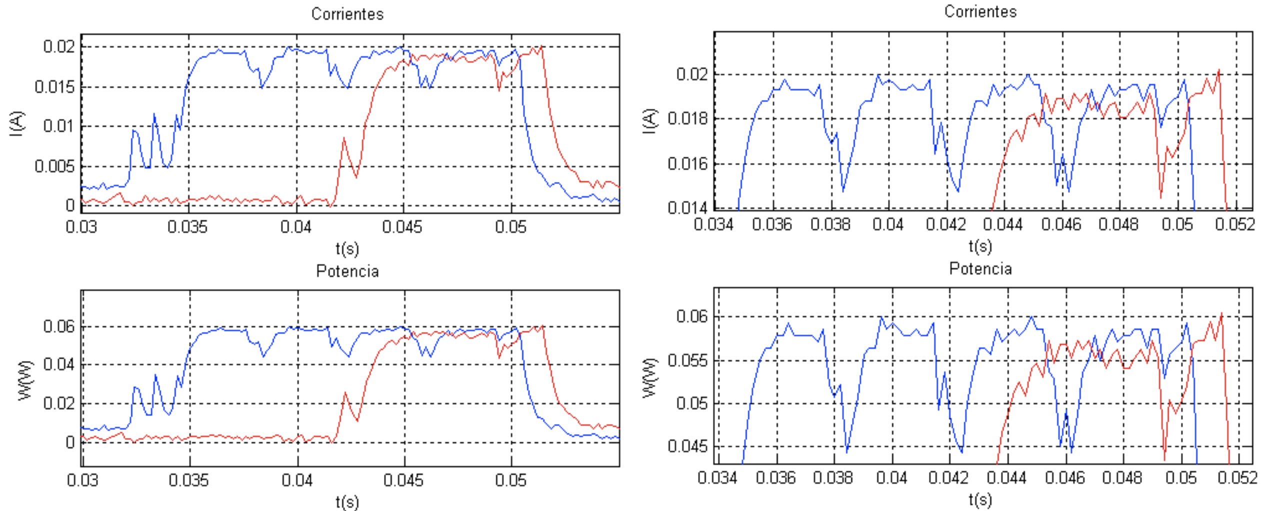


Figura 45: Detalle de los paquetes

El consumo mínimo por el envío de cada paquete es de 0,044W, 44mW. Para la recepción de un paquete el consumo es de 0,055W, 55mW. Para calcular la energía necesaria es necesario estimar el tiempo durante el cual está encendida la radio, que es el tiempo necesario para recibir un paquete completo.

7.7 Deducción de los tiempos y de potencias involucrados

A los efectos de la caracterización del consumo decidimos estudiar el comportamiento durante la realización de las tareas básicas, transmisión, recepción, CPU, CCA y broadcast. Se realizó a través del tratamiento numérico de los datos relevados en el laboratorio.

Apoyados en este análisis se realizó la caracterización del consumo a través de ecuaciones en función de la duración de cada tipo de actividad. En particular se estudió el caso del tráfico RPL, analizando los mensajes DIS, DAO y DIO. Para esto se contrastó la información relevada en el laboratorio con los nodos ejecutando el código rpl-collect con el análisis de la simulación del mismo código.

Finalmente, realizamos una proyección del consumo asumiendo cambios en el valor de algunos parámetros de ContikiMAC, las condiciones de uso y distintos intervalos de trabajo.

7.7.1 Transmisión

En la figura 52 se observa que con claridad el efecto que atribuimos de la carga y descarga del condensador de la radio. El tiempo en que está apagada la radio no es lo suficientemente largo como para que el consumo se anule. Se midieron intervalos entre los mínimos que se producen cuando comienza nuevamente la carga del condensador luego del encendido de la radio.

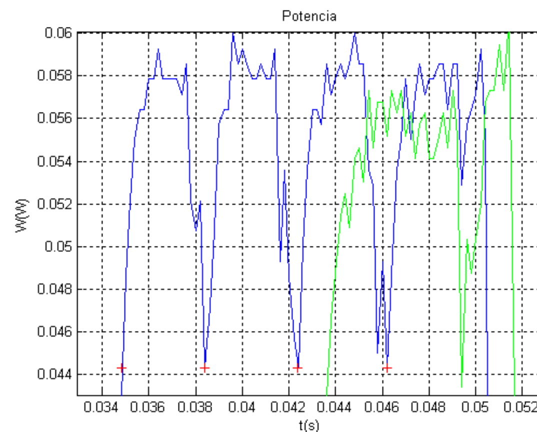


Figura 46: Medida de transmisión con el osciloscopio

Los tres intervalos de tiempo marcados con los puntos rojos tienen una duración de, Int1=0.0036s, Int2=0.0040s, Int3=0.0038s, el promedio resultante es de 3,8ms por intento. Es decir que por cada intento de transmisión de ese paquete estoy consumiendo como mínimo 44mWx3,8ms=167,1uJ. Eso variará con la duración de tiempo necesario de transmisión de cada tipo de paquete.

En el momento en que se trasmite realmente el paquete el consumo es de 58mW. Adicionalmente es necesario tener en cuenta el consumo que se produce durante la carga y descarga del condensador.

7.7.2 Recepción

En las gráficas anteriores se observa que el consumo durante una recepción es de 55mW. Nuevamente el tiempo necesario para completar una recepción también dependerá del tamaño del paquete y del instante en el que el potencial receptor encienda la radio. En la mayoría de los ejemplos relevados se observó que la radio del receptor se encendió aproximadamente en la mitad de la transmisión de un paquete y tuvo que esperar al siguiente paquete para realizar la recepción completa y luego enviar la confirmación. Apuntando a evaluar el consumo es necesario tener en cuenta si la detección de la transmisión por parte del receptor se realiza en la primera o segunda CCA.

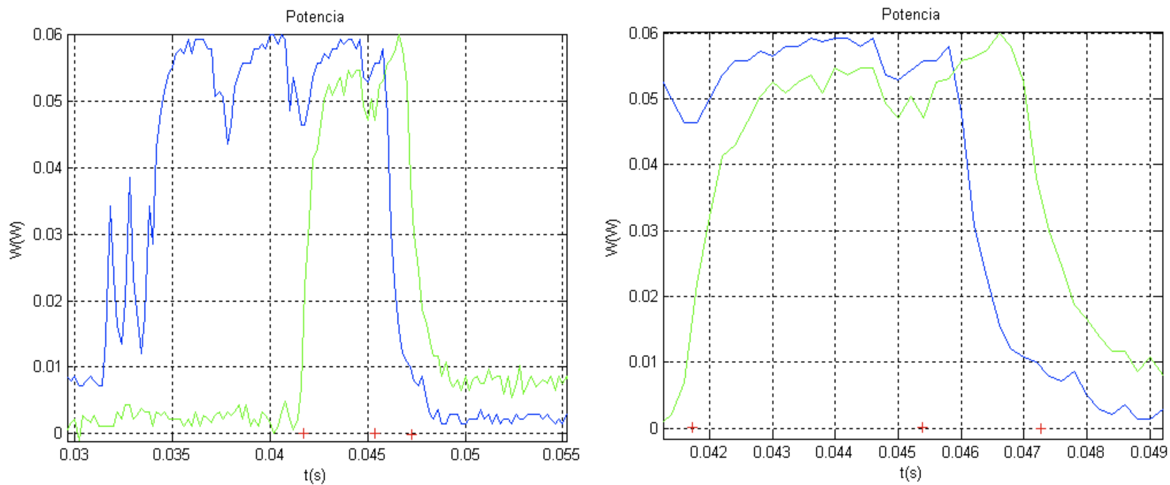


Figura 47: Medida de recepción con el osciloscopio

En la figura 53 se relevó que el intervalo para concretar la recepción es Int1=3,7ms, con un consumo aproximado de 50mW, mientras que el envío de la confirmación dura Int2=1,9ms con un consumo de 55mW. En este ejemplo el consumo total es de 289,5uJ.

Se realizaron varias medidas y se encontró que el tiempo promedio para concretar la recepción es de 6,175 ms con un consumo de 54mW y una energía de 333,5uJ. Mientras que para el envío de la confirmación de recepción el tiempo que lleva es de 2,35 ms con un consumo de 54mW y una energía de 126,9uJ.

7.7.3 Análisis del consumo de la CPU

En la figura 54 se observa el consumo del Mote con la radio apagada tanto para el transmisor como para el receptor. Asimilaremos este consumo al de la CPU. La cruces roja indican los tiempo tenidos en cuenta en ambos casos.

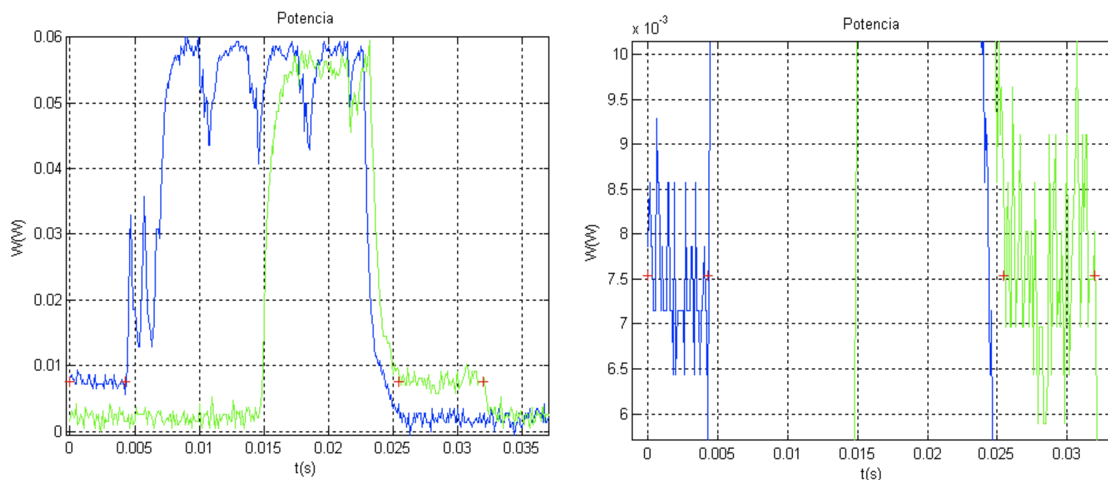


Figura 48: Caracterización del consumo de la CPU

Tomamos el consumo con la radio apagada como 7,5mW. En el caso del nodo que realiza la Tx no se observa toda la actividad de la CPU previa al encendido de la radio, mientras que para el nodo que recibe si se observa el consumo de la tarea posterior al apagado de la radio y previa al pasaje a LPM.

Por cada ms con la CPU prendida pero con la radio apagada va a consumir aproximadamente 7,5mWx1ms = 7,5uJ. En este ejemplo se observa para el nodo que trasmite estuvo en ese estado 4,3ms y en consecuencia consumió 32,25uJ. Mientras que en el caso del nodo que realiza la recepción el tiempo que estuvo en ese estado fue 6,5ms, por lo que su consumo fue de 48,75uJ.

7.7.4 Análisis del Broadcast

De manera análoga a los casos anteriores se caracterizó el un broadcast. En la figura 55 se ilustra un ejemplo. El intervalo marcado con las cruces rojas es la duración del envío, 128,4ms. El consumo se puede estimar como 0,055. 0,1284 =7062mJ =7062uJ.

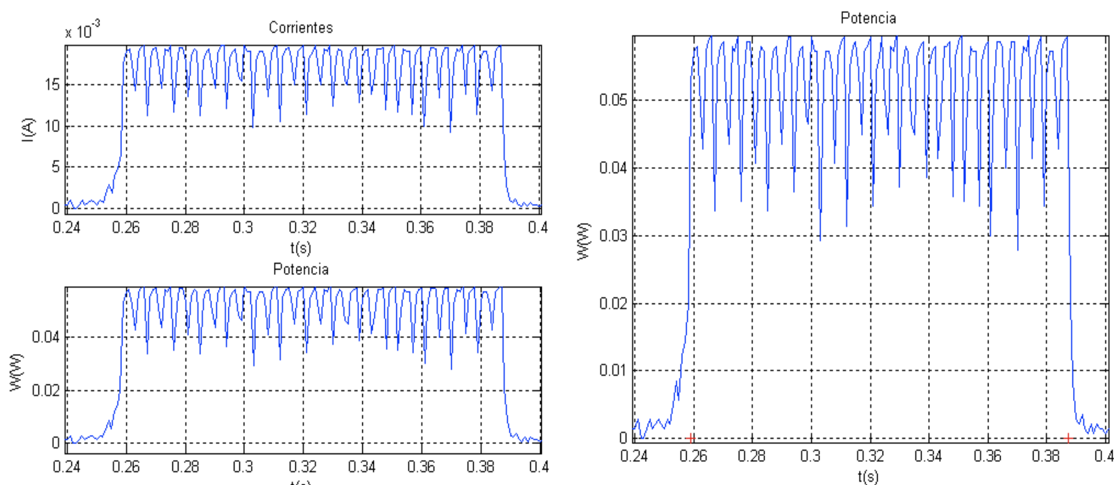
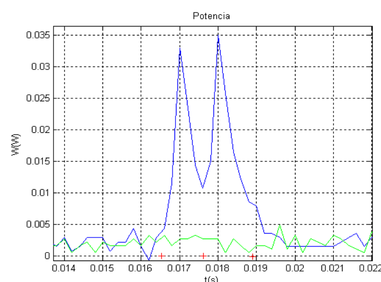
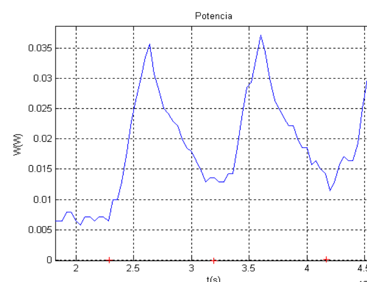


Figura 49: Análisis de broadcast

7.7.5 Análisis de las CCA con una aproximación geométrica

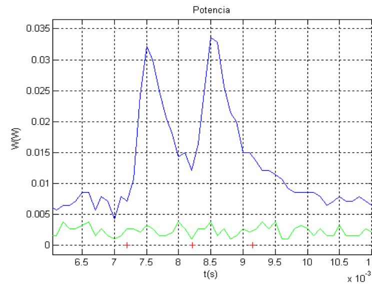
A partir de los datos relevados se observó que las formas de los componentes de las CCA's se aproximaban correctamente por triángulos. Observamos que los picos que se producían eran siempre tales que el segundo pico es levemente mayor que el primero. Se relevaron algunos casos representativos de manera de evaluar la el consumo de las CCA's.



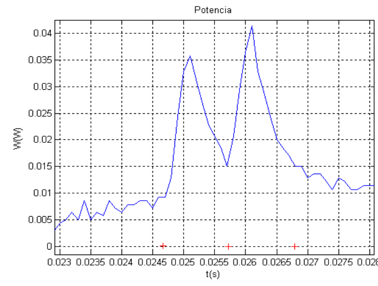


Int1 = 0.0011s, Int2 = 0.0013s

Int1 = 9.0467e-004s, Int2 = 9.7475e-004s



Int1 = 0.0010s, Int2 = 9.3599e-004s



Int1 = 0.0011s, Int2 = 0.0011s

Los promedios de cada intervalo son, Int1 = 1.026e-3s, Int2 = 1.078e-3s. En consecuencia el consumo de cada CCA es el siguiente:

$E_{CCA} = E_h = [(1,026ms).(33,5mW) + (1,078ms).(36,5mW)] /2 = 36,86$ uJ

Para estimar el valor de cada elemento de la CCA tomamos 18,43uJ.

7.8 Ecuaciones de estimación de Consumo

Hasta aquí hicimos un análisis para tener estimaciones de los consumos. A continuación de describen las ecuaciones para transmisión y recepción. La energía consumida por realizar una transmisión se obtiene de sumar la energía correspondiente a la CCA de escucha de canal para evitar eventuales colisiones y la energía necesaria para trasmitir un paquete multiplicada por la cantidad de paquetes.

$$P_{Tx} = P_h + n.P_{Tx_{paq}}$$

Mientras que la energía necesaria para realizar las escuchas de canal con cada CCA se puede medir con razonable precisión, no ocurre lo mismo con la energía necesaria para el envío de paquetes. Esto se debe que en sentido estricto va a depender del tamaño de cada paquete y el tiempo necesario para ponerlo en el aire. La corriente que va a consumir para esta tarea se puede medir pero no se puede tener en cuenta todos los casos de tiempos de paquetes.

Resulta más sencillo establecer cotas. El consumo mínimo y máximo se pueden estimar a partir del análisis de las medidas del laboratorio. Adicionalmente para el máximo también se puede obtener una cota a partir de las hojas de datos.

El consumo mínimo dependerá del tiempo teórico de radio apagada entre paquetes previsto. Este tiempo condicionará la caída de consumo debido a la descarga del condensador y el momento de encendido en que comenzará la carga del condensador hasta el valor de régimen.

Una vez conocido el tiempo necesario para poner un paquete en el aire se puede estimar la potencia necesaria para enviar n paquetes se puede expresar de la siguiente forma:

$$E_{Tx} = E_h + n.P_{Tx-paq} . t_{paq}$$

donde P_{Tx-paq} es la potencia necesaria para trasmitir un paquete. Teniendo en cuenta los mínimos y los máximos explicados anteriormente la ecuación queda de la siguiente manera:

$$E_h + n.P_{Tx-min,paq} . t_{paq} \leqslant E_{Tx} \leqslant E_h + n.P_{Tx-max,paq} . t_{paq}$$

donde t_h es la duración de una CCA, t_{paq} el tiempo necesario para trasmitir un paquete. En el caso de la recepción se realiza un análisis similar. Se tiene en cuenta la cantidad de elementos de la CCA y el tiempo necesario para completar la recepción del paquete y el envío de la confirmación

$$E_{Rx} = k.E_{e,h} + P_{Rx}.t_{Rx} + P_{ack}.t_{ack}$$

donde $E_{e,h}$ es la energía de un elemento de la CCA, P_{Rx} es la potencia necesaria para realizar una recepción, P_{ack} el la potencia necesaria para enviar una confirmación.

Teniendo en cuenta las medidas realizadas, las ecuaciones se pueden formular de la siguiente manera,

Transmisión:

$$73{,}72\,uJ + n.44\,mW.t_{paq} \leqslant E_{Tx} \leqslant 73{,}72\,uJ + n.58\,mW.t_{paq}$$

Recepción:

$$E_{Rx} = k.36{,}86\,uJ + 54\,mW.t_{Rx} + 54\,mW.t_{ack}$$

CPU:

$$E_{CPU} = 7{,}5\ uJ \text{ en } 1\text{ms.}$$

7.9 Ecuaciones expresadas en relación de los bytes para casos de tráfico RPL

Como se adelantó, las medidas del laboratorio se realizaron programando los nodos con el código del ejemplo rpl-collect que viene con Contiki. Los datos relevados en el laboratorio se analizaron con 2 herramientas, COOJA y Wireshark. En COOJA se analizó el tráfico con el Log Listener, el Radio Logger y la Timeline. En el ejemplo se observa la identificación de una ráfaga de 29 paquetes RPL-DIO.

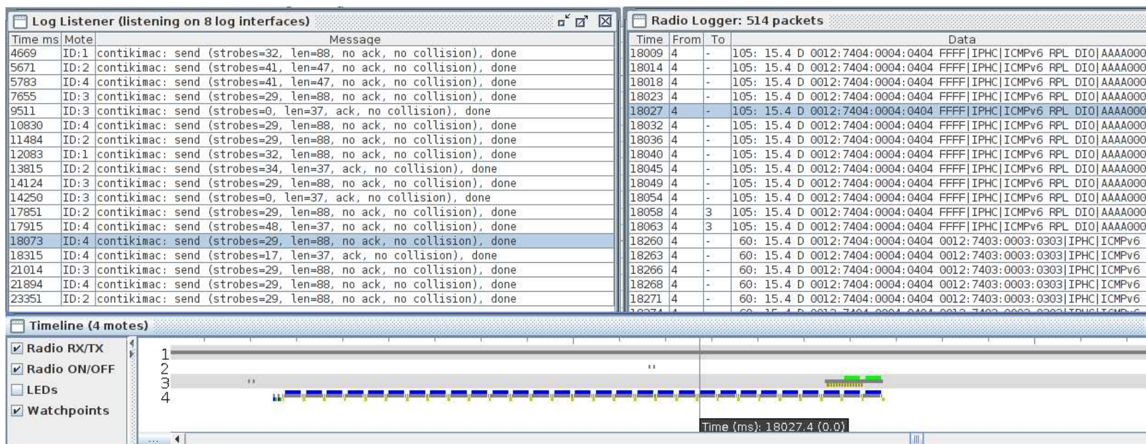


Figura 50: Log Listener, Radio Logger y Timeline de COOJA

Para realizar un análisis de la simulación hicimos verificaciones cruzadas de las tres herramientas de COOJA de manera de confirmar de que tipo de paquete se trataba. Por ejemplo en las figuras siguientes se

ilustra el broadcast de 29 paquetes medido en el osciloscopio y su representación en el simulador.

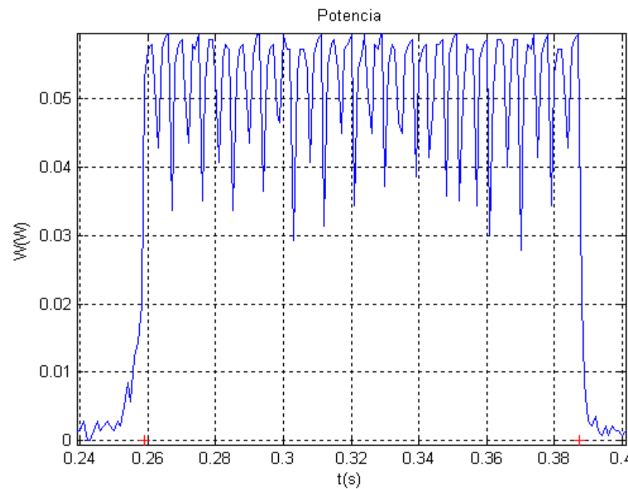


Figura 51: Broadcast medido con osciloscopio

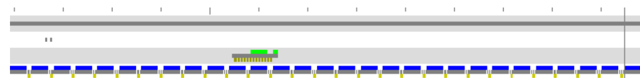


Figura 52: Broadcast en la Timeline del simulador

Durante el análisis realizado se identificaron tres ráfagas de distintos tipos de paquetes RPL. Medimos el tamaño, el tiempo necesario para trasmitirlo, el tiempo entre 2 paquetes, tiempo en el que la radio está encendida para cada paquete, la duración total de la ráfaga y contamos la cantidad de paquetes. El resultado se presenta en la siguiente tabla:

	DIS	DIO	DAO
Tamaño de paquete (B)	47	88	37
Duración de paquete (ms)	2,205	3,518	2,080
Duración inter-paq. (ms)	0,295	0,3	-
Duración radio ON	2,8	4,18	-
Duración Ráfaga (ms)	129,4	130	129,4
Cantidad de Paquetes	41	29	48
Difusión	Broadcast	Broadcast	Unicast

Tabla 4: Resultados del análisis del ejemplo rpl-collect

Se compararon los datos simulados con los datos medidos en el laboratorio. Analizamos en envío de una ráfaga de paquetes ICMPv6 RPL DIO que tiene un tamaño de 88 Bytes de carga y un tamaño total de paquete de 119 Bytes (relevado en Wireshark).

Son necesarios 3,518ms para trasmitir dicho paquete y por cada uno de estos paquetes se consumirán 58mW, es decir 58mW.3,518ms = 204uJ. Si a esto sumamos el tiempo en que la radio está encendida antes y después del envío del paquete, el tiempo de radio encendida para enviar un paquete es de 4,18ms. El consumo correspondiente es de 242,4uJ.

Debemos además sumar el consumo en el lapso en el que la radio está apagada pero que no es nulo debido al tiempo de descarga y carga del condensador (ver características del Mote) y a que la CPU sigue activa. Se trata de un valor de la configuración en ContikiMAC de los tiempos entre paquetes. Este se puede estimar a partir de las medidas del laboratorio. En el caso analizado, la configuración por defecto, aproximadamente 50mW.

Tx de un paquete ICMPv6 RPL DIO

$$E_{paq-RPL} = \frac{E[J]}{k[Bytes]} = \frac{242{,}7[uJ]}{119[Bytes]} = 2{,}039\frac{uJ}{Byte}$$

Para 29 paquetes:

$$E_{29-RPL-DIO} = 29\,.\,2{,}039\frac{uJ}{Byte}$$

7.10 Proyección del consumo para otros escenarios

Realizamos el análisis del consumo en el caso de que se varíe la duración del intervalo del ciclo básico de funcionamiento para algunos tipos de tráficos y su impacto considerando un lapso de régimen.

7.10.1 Caso: CCAs

Recordemos que el valor del consumo por cada par de CCA es 36,86uJ. Realizamos el cálculo de la cantidad ciclos de escucha que se producen en una hora y luego su consumo asociado. Llamamos T al tiempo de funcionamiento considerado, q a la cantidad de pares de CCA y c al ciclo de manera que,

$$q = \frac{T}{c}$$

En consecuencia el consumo durante un lapso T debido a las escuchas de bajo consumo se puede expresar como,

$$E_{2\mathrm{CCA}-T} = q_T\,.\,E_{2\mathrm{CCA}} = \frac{T}{c}.\,E_{2\mathrm{CCA}}$$

Realizamos una proyección del consumo para distintos valores de ciclos de escucha. Se consideró un rango que va 62,5ms a 1s. En la Figura 59 se presenta el resultado.

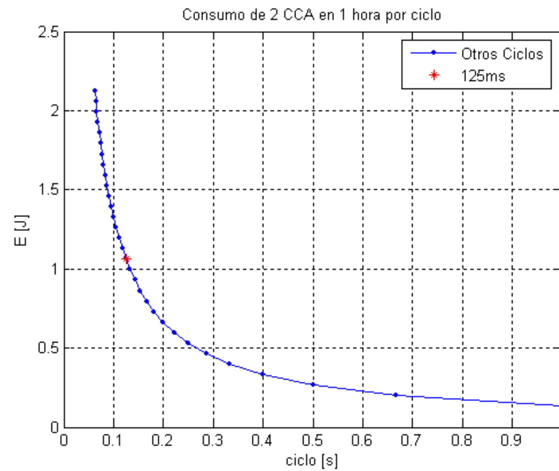


Figura 53: Consumo distintos ciclos de escucha en una hora.

Se observa como decrece de manera inversamente proporcional al valor del ciclo.

7.10.2 Caso: Broadcast

De manera análoga al caso anterior se estudió el consumo asociado a los broadcast debido a que tienen una duración de un ciclo. A partir del valor relevado de consumo 55mW calculamos el consumo durante cada uno de los ciclos estudiados. A los efectos de visualizar mejor la variación de los datos se presenta un gráfico lineal y semi logarítmico.

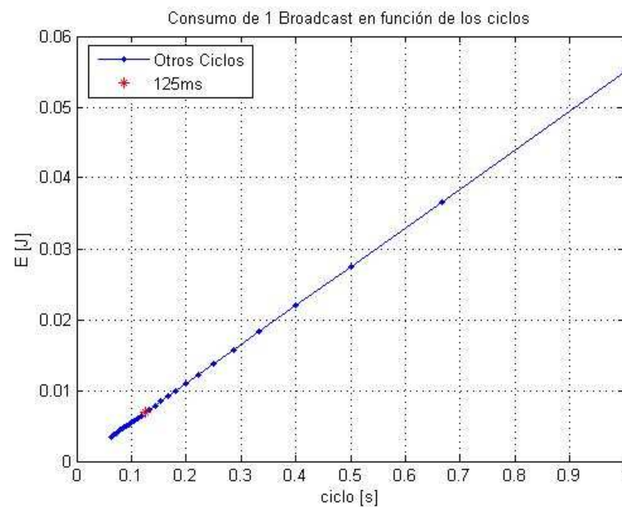


Figura 54: Consumo debido a Broadcast en una hora por ciclos.

7.10.3 Suma de los efectos

Para analizar el efecto de la variación del ciclo de ContikiMAC sobre el consumo de las escuchas y el los broadcast fue necesario considerar distintas cadencias de envío de broadcast. Consideramos el mismo rango de variación del ciclo y 5 tasas de envío de broadcast por hora.

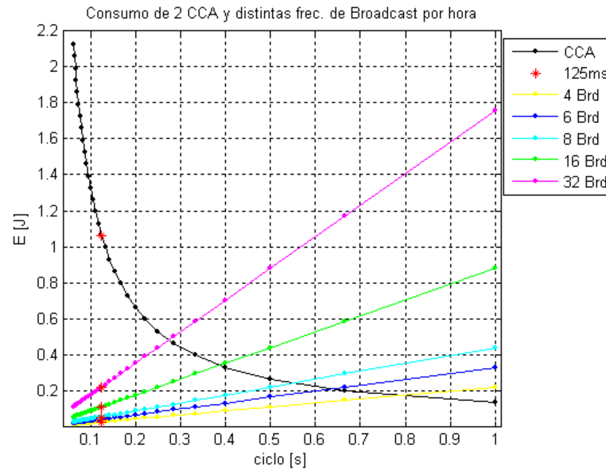


Figura 55: Efectos de escuchas de CCA y de Broadcast por ciclo.

En la Figura 61 se observa como crece el consumo debido a los broadcast. En la intersección de la curva que representa el consumo de la CCA con las de los distintas tasas de envío de broadcast se puede encontrar el consumo mínimo para un ciclo dado con una tasa de envío de broadcast dada. Las distintas tasas se pueden deber tanto a tráfico generado desde el stack de comunicaciones de Contiki para realizar tareas de gestión de la red, como debidas a la aplicación desarrollada por el usuario. Para analizar el efecto combinado sumamos ambos consumos.

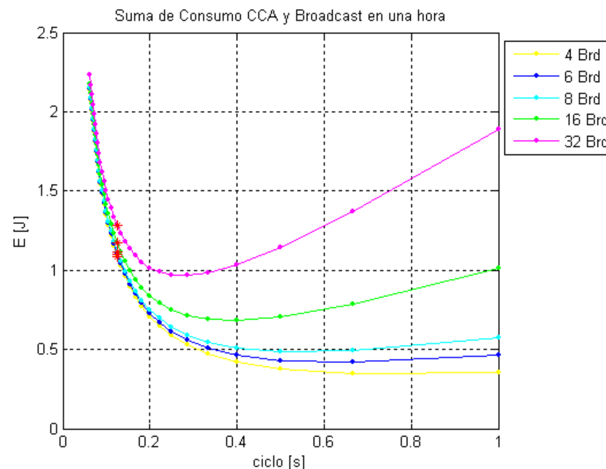


Figura 56: Suma de Consumo de CCA y Broadcast

En Figura 62 se observa que para el valor del ciclo configurado por defecto en Contiki no se observan grandes diferencias de consumo, mientras que a medida que se incrementa el valor del ciclo el consumo decrece hasta un punto en el cual se alcanza un mínimo luego del cual comienza a crecer nuevamente con distintas pendientes.

A partir del análisis del comportamiento de la red tanto en régimen como en los transitorios que se puedan producir, el diseñador de la red y desarrollador de la aplicación podrá optar por cambiar el valor del ciclo para mejorar el consumo de los nodos de la red debido a ContikiMAC.

7.11 Pruebas de Aplicación

Se realizaron simulaciones y pruebas reales con motes, con el fin de verificar el correcto funcionamiento de la aplicación implementada, ContikiWSN.

Las simulaciones se realizaron con el simulador COOJA y las pruebas se llevaron a cabo en el interior de la casa de uno de los integrantes del proyecto.

7.11.1 Prueba de Trickle. Simulación

Se realizó una simulación con 3 nodos (1 sink y 2 senders), de tal manera que estuvieran alineados y que el sink viera solamente al nodo más cercano (el número 2) y así el nodo número 3 tenga que llegar al sink a través del nodo 2.

La simulación se realizó con COOJA y se utilizó collect-view para la obtención de los datos en el sink.

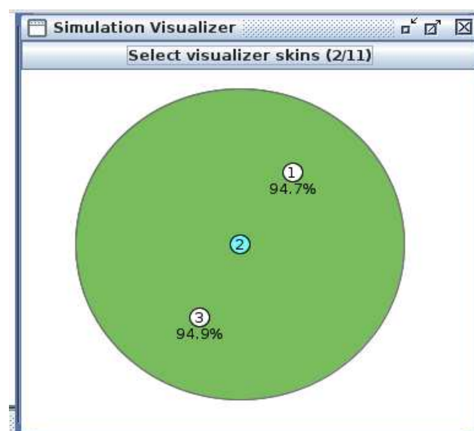


Figura 57: Red utilizada en la simulación

Estando la red en régimen con un período de muestreo de 60 segundos, se aísla al nodo más lejano (número 3) durante 10 minutos. A los 5 minutos de que se aisló se cambia el período de muestreo a 120 segundos, y luego de 5 minutos más se recupera el nodo.

La manera de aislar al nodo en la simulación fue simplemente alejarlo hasta que quedara fuera del alcance de radio del nodo 2.

7.11.2 Resultados

- Se aisló al nodo 3 a la hora 21:43 (tiempo en hora de simulación). Se empezaron a recolectar datos del nodo 2 únicamente cada 60 segundos, como se esperaba.
- A las 21:47, luego de que se recolectaron 4 datos del nodo 2, se cambió el período de muestreo a 120 segundos. Se empezaron a recolectar datos del nodo 2 únicamente cada 120 segundos, como se esperaba.
- A las 21:53, luego de que se recolectaron 3 datos del nodo 2, se recupera al nodo 3.
- Hasta las 22:00 se reciben datos cada 120 segundo del nodo 2 únicamente.
- A partir de las 22:00 se empiezan a recuperar datos del nodo 3 que estaban en el buffer. En teoría se guardaron 10 datos (tamaño máximo del buffer) con un período de 60 segundos. A las 22:00 aparecieron los datos del nodo 3 de las 21:45 y 21:46. Cabe resaltar que el dato de las 21:44 se perdió.
- Luego fueron apareciendo los datos guardados en l buffer del nodo 3. Aparecieron en total 10 datos con período de 60 segundos (de las 21:45 a 21:54). Mientras tanto el nodo 2 sigue recolectando con período de 120 segundos.
- El siguiente dato que se recupera del nodo 3 es el de las 22:02. Es lógico, ya que a las 22:00 fue el momento en que el nodo 3 empezó a liberar datos del buffer y pudo empezar a guardar nuevos.
- El nodo 3 manda un dato más cada 60 segundos (22:03) y luego cambia de período a 120 segundos.
- A las 22:15 se recuperan todos los datos que tenía el nodo 3 y vuelve a mandar datos a tiempo real como el nodo 2.
- A las 22:23 vuelve al estado de régimen.

En las siguiente imagen se observa los datos recolectados por collect-view en la ventana de Historical Power Consumption:

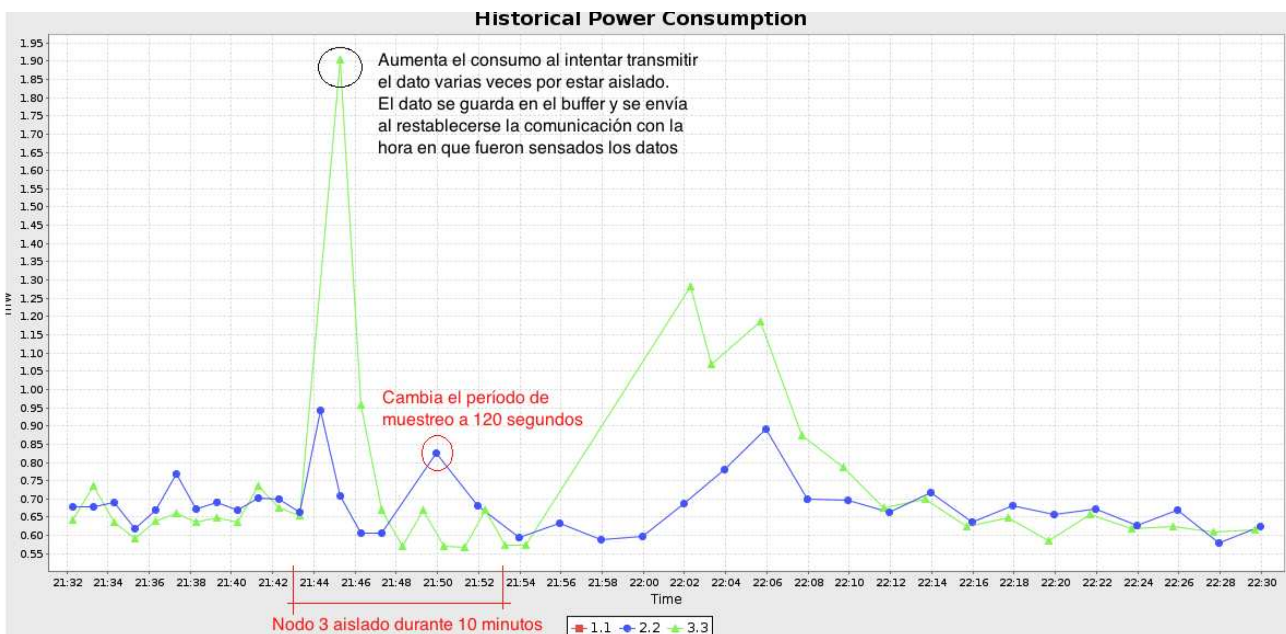


Figura 58: Registro histórico de consumo de la simulación

En la siguiente imagen se aprecia el consumo promedio de los nodos en un gráfica, discriminado los consumos de CPU, LPM (Low Power Mode), Radio Listen y Radio Transmit:

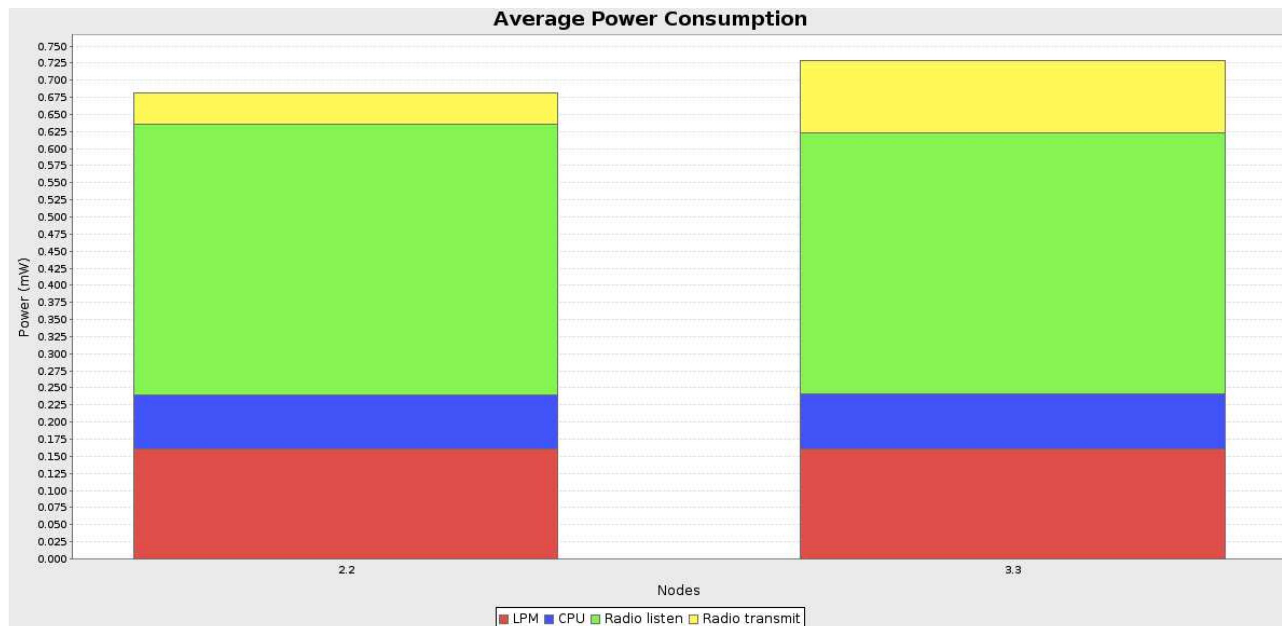


Figura 59: Discriminación del consumo promedio de la simulación

En la siguiente imagen se observa el Radio Duty Cycle promedio de los nodos que collect-view muestra:

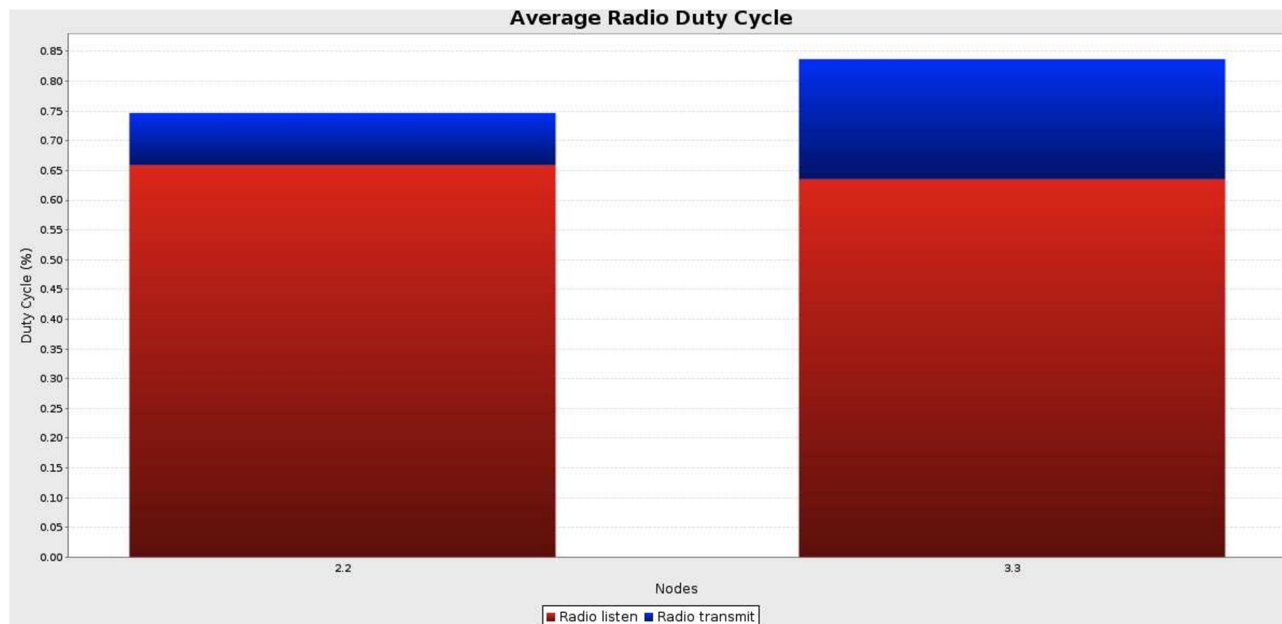


Figura 60: RDC promedio de la simulación

Se obtuvo un RDC mayor en el nodo que estuvo asilado durante 10 minutos, lo cual era de esperarse ya que al estar aislado intenta enviar los paquetes varias veces hasta darse cuenta de que no tiene comunicación lo que produce que la radio deba estar más tiempo transmitiendo.

La siguiente tabla muestra los datos más relevante que collect-view ofrece para la simulación:

Nodo	Paquetes recibidos	Paquetes perdidos	Beacon interval	CPU power	LPM power	Listen power	Transmit Power	Power	Listen duty cycle	Transmit duty cycle	Avg inter-packet time
2	37	0	34 min, 57 sec	0.079 mW	0.161 mW	0.395 mW	0.046 mW	0.681 mW	0,66%	0,09%	1 min, 33 sec
3	37	4	19 min, 31 sec	0.080 mW	0.161 mW	0.381 mW	0.107 mW	0.729 mW	0,64%	0,20%	1 min, 33 sec

Tabla 5: Datos relevantes del collect-view de la simulación

7.11.3 Pruebas con los motes

Se realizó una prueba real con motes dentro de la casa de uno de los integrantes del proyecto. La topología utilizada fue idéntica a la simulación de la parte anterior, es decir 3 nodos en línea en el cual el nodo más lejano llega al sink a través del que más cercano. Se hizo la misma secuencia en cuanto al aislamiento del nodo más lejano y el cambio de período de muestreo. Se siguió el mismo tiempo de duración para cada parte de la prueba (tiempos de aislamiento y cambio de período de muestreo).

Para aislar el nodo se introdujo el mismo dentro de una caja metálica.

Los resultados obtenidos fueron muy similares a los resultados de la simulación. En la siguiente imagen se muestra la gráfica de Historical Power Consumption de collect-view. El nodo 195.177 corresponde al sink, el nodo 57.39 es el sender más cercano al sink y el 105.63 el sender más lejano (nodo que se aísla).

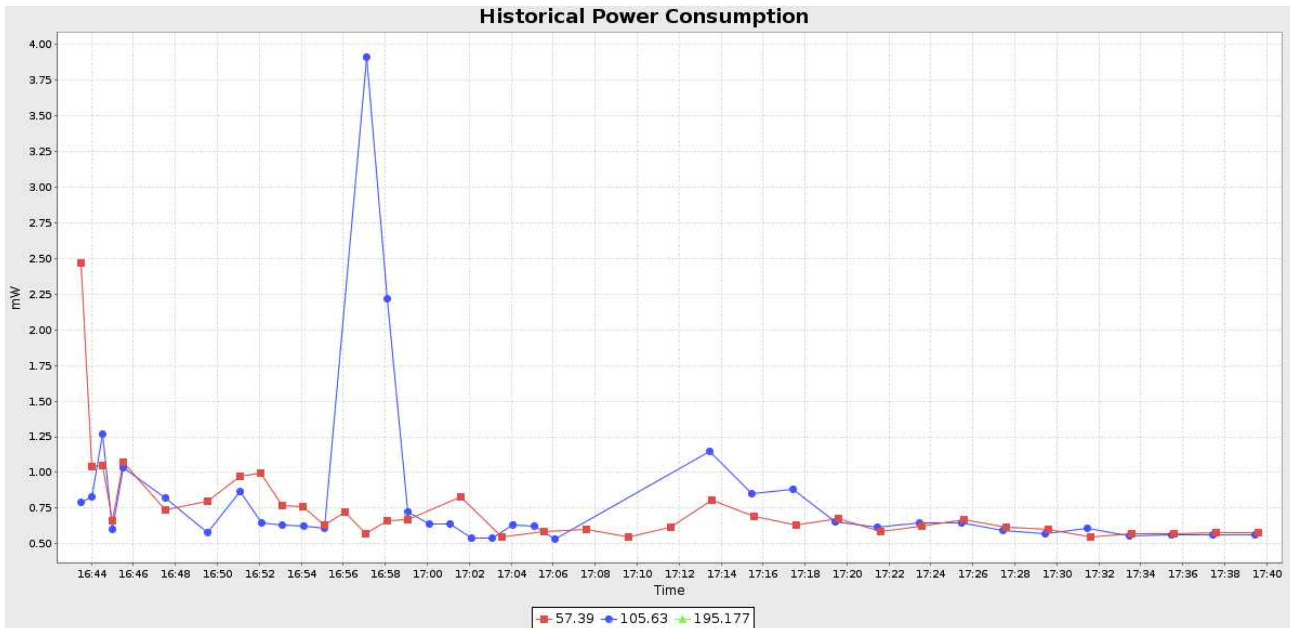


Figura 61: Registro histórico de consumo de la prueba

En la siguiente gráfica se muestra el consumo promedio para la prueba:

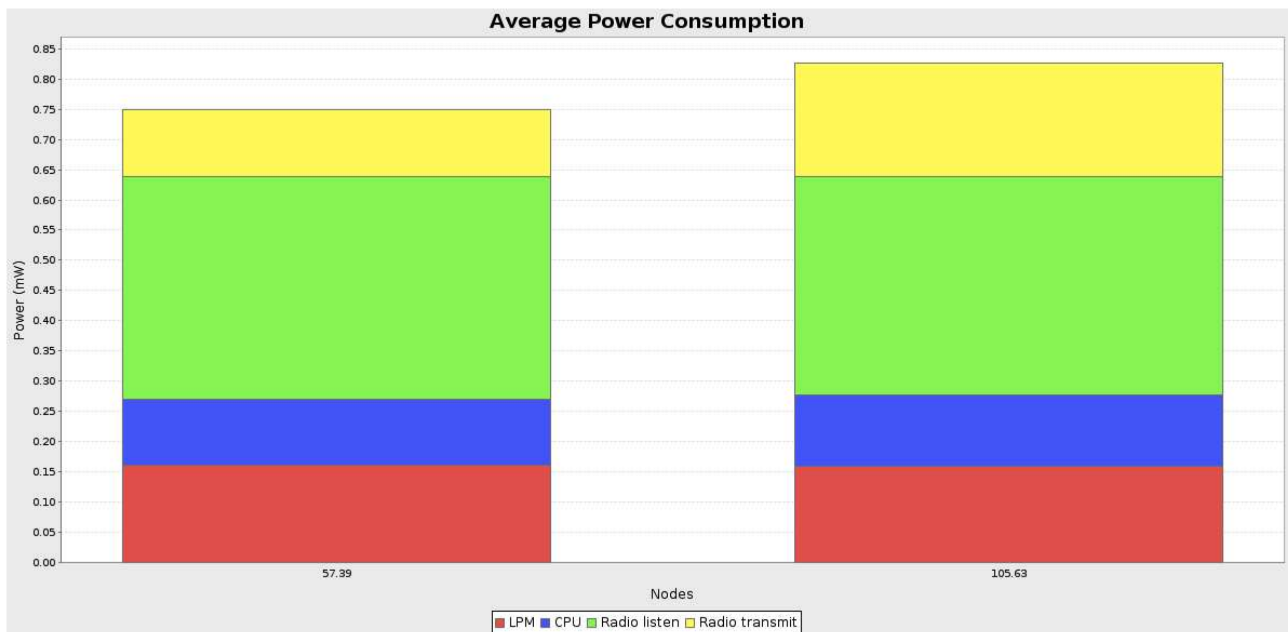


Figura 62: Discriminación del consumo promedio de la prueba

La imagen siguiente muestra el Radio Duty Cycle promedio de la prueba:

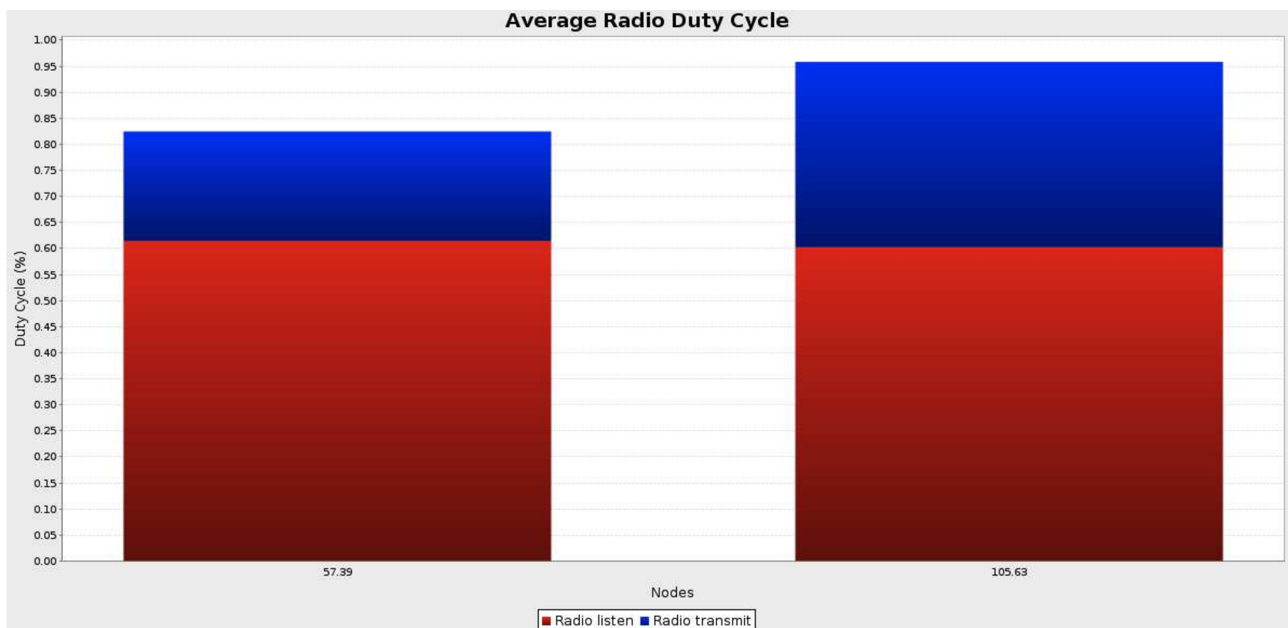


Figura 63: RDC promedio de la prueba

En la siguiente tabla se ven los datos más relevantes de la prueba:

Nodo	Paquetes recibidos	Paquetes perdidos	Beacon interval	CPU power	LPM power	Listen power	Transmit Power	Power	Listen duty cycle	Transmit duty cycle	Avg inter-packet time
57.39	36	0	28 min, 53 sec	0.110 mW	0.160 mW	0.369 mW	0.111 mW	0.750 mW	0,62%	0,21%	1 min, 33 sec
105.63	36	4	19 min, 59 sec	0.117 mW	0.160 mW	0.361 mW	0.187 mW	0.827 mW	0,60%	0,36%	1 min, 33 sec

Tabla 6: Datos relevantes del collect-view de la prueba

Se pudo observar como durante las simulaciones y durante las pruebas reales con los motes la aplicación ContikiWSN funcionó como se esperaba. Al darse cuenta un nodo que no tiene comunicación guarda los datos que va sensado, y al restablecerse la comunicación se envían con la hora a la que los datos fueron sensados.

7.12 Prueba con motes durante dos semanas

Se realizó una prueba durante 17 días con 3 motes programados con la aplicación ContikiWSN. Los motes se ubicaron dentro de la casa de uno de los integrantes del proyecto con una distancia de aproximadamente 4 metros entre motes. Un mote fue programado como sink (195.177) y los otros dos de los motes fueron programados como sender (57.39 y 105.63). A los motes senders se les asignó un período de muestreo de 10 minutos y se los ubicó de tal manera que el mote más lejano al sink, mote 105.63, tuviera que llegar al sender con un salto a través del mote 57.39. La topología de la prueba se representa en la siguiente figura:

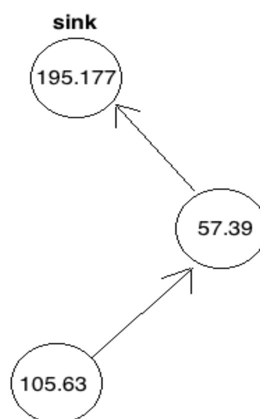


Figura 64: Topología de la prueba continuada con motes

Los resultados más relevantes obtenidos con collect-view fueron los siguientes:

- Gráfica del Power Consumption promedio obtenido:

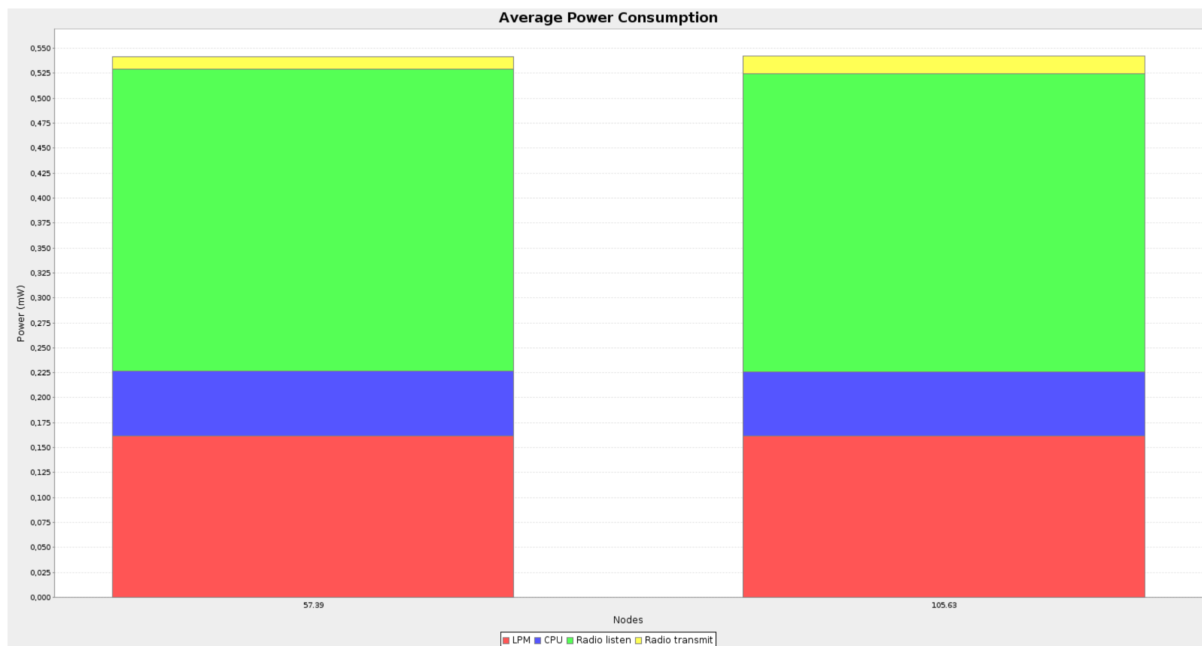


Figura 65: Discriminación del consumo promedio de la prueba

- Gráfica del RDC promedio obtenido:

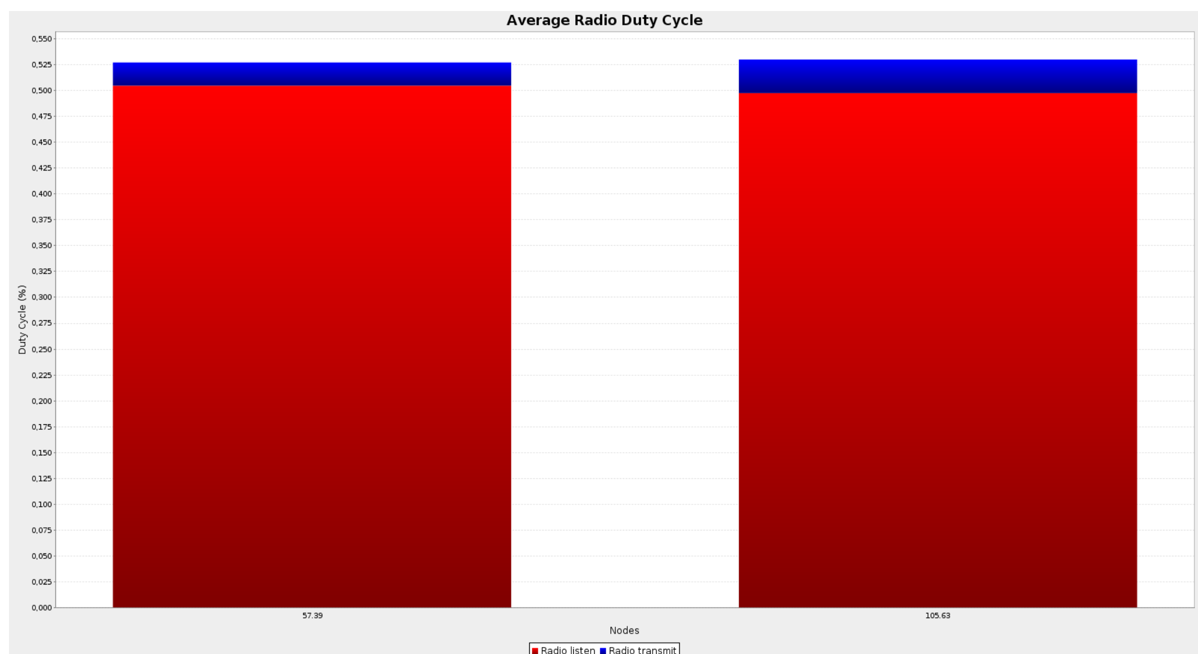


Figura 66: RDC promedio de la prueba

En la siguiente tabla se muestran los datos más interesantes obtenidos:

Nodo	Paquetes recibidos	Paquetes perdidos	Beacon interval	CPU power	LPM power	Listen power	Transmit Power	Average Power	Listen duty cycle	Transmit duty cycle	Avg inter-packet time
57.39	2432	0	34 min, 55 sec	0.065 0 mW	0.162 mW	0.303 mW	0.012 mW	0.541 mW	0,51%	0,02%	10 min, 02 sec
105.63	2425	9	31 min, 57 sec	0.065 mW	0.162 mW	0.299 mW	0.017 mW	0.542 mW	0,50%	0,03%	10 min, 04 sec

Tabla 7: Datos relevantes del collect-view de la prueba continuada

Se ve que el consumo promedio en ambos nodos fue muy similar ya que prácticamente no hubieron paquetes perdidos durante la prueba. El RDC promedio en ambos casos fue inferior a 0,6 %, lo cual resulta bueno ya en redes de sensores inalámbricas se intenta que la radio esté apagada la mayor parte del tiempo.

7.13 Prueba con mayor cantidad de nodos

Con el fin de ver como afecta el consumo de la red a medida que se agregan motes, se desplegó en el IIE (Instituto de Ingeniería Eléctrica) una red comprendida por 8 motes (1 sink y 7 senders). La distribución de la red se muestra en la siguiente figura:

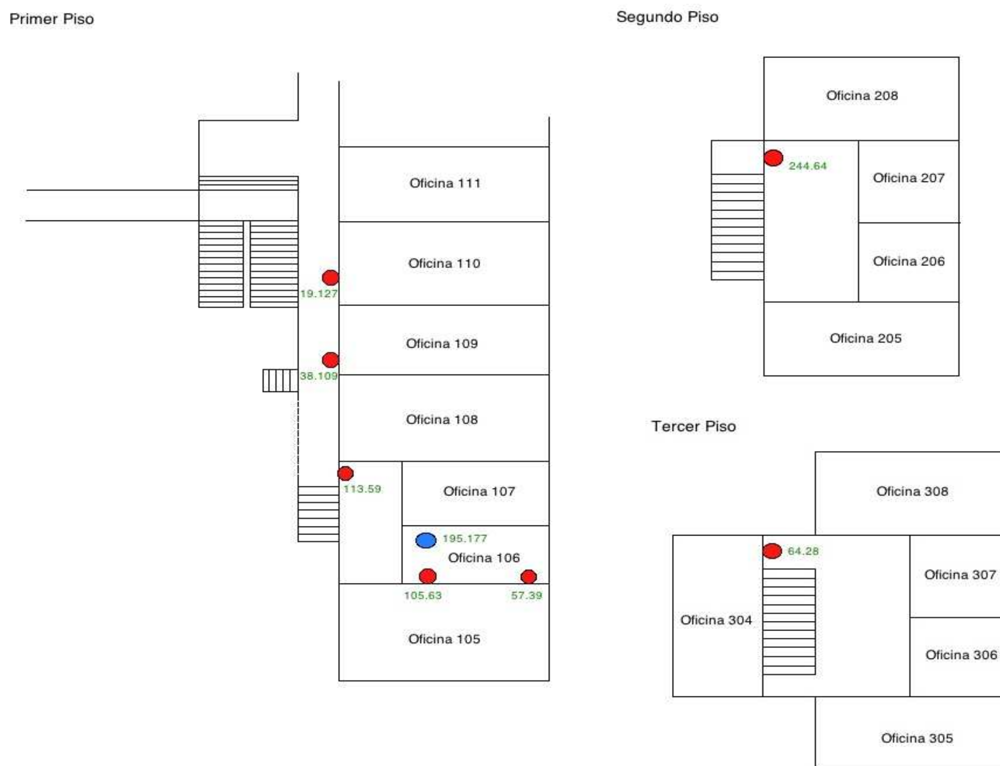


Figura 67: Distribución de la red en el IIE

Se vio como el aumento de la cantidad de motes repercute significativamente en los resultados. Los resultados obtenidos se muestran en la siguiente tabla:

Mote	RDC promedio (%)	Consumo promedio (mW)	Beacon Interval	Paquetes recibidos	Paquetes perdidos
57.39	1.67	1.354	17 min 52 sec	1144	14
64.28	1.32	1.090	15 min 41 sec	1123	38
105.63	0.97	0.865	22 min 52 sec	1146	12
113.59	1.84	1.399	15 min 31 sec	1136	152
19.127	4.34	2.932	10 min 33 sec	557	598
244.64	1.30	1.063	15 min 16 sec	1089	33
38.109	5.11	3.392	19 min 03 sec	888	260

Tabla 8: Resultados de collect-view de la prueba en el IIE

La gran cantidad de paquetes perdidos es en parte consecuencia de que en el instituto hubo un corte de energía y la computadora a la que estaba conectado el mote sink se debió apagar, y estuvo apagada por 3 días. De todas formas durante la prueba se constató que los motes más lejanos al sink, motes 19.127 y 38.109, perdían muchos paquetes, lo que repercutía en el funcionamiento de la red.

7.14 Comparación de resultados

Se realizaron varias simulaciones con COOJA con el objetivo de comparar los distintos resultados utilizando la aplicación rpl-collect de Contiki y la aplicación del proyecto ContikiWSN (simulada y con motes como se analizó en el capítulo 7.12).

También se compara con resultados obtenidos en un trabajo realizado por docentes del Instituto de Ingeniería Eléctrica en una red de sensores inalámbricas para aplicaciones agropecuarias [33], en donde se utilizaron motes TelosB y el sistema operativo TinyOS.

Se simuló con COOJA la aplicación rpl-collect y la aplicación ContikiWSN con dos períodos de muestreo, 600 y 900 segundos, durante un día aproximadamente. Las simulaciones se realizaron con 3 motes, al igual que en el capítulo 6.12 con 1 sink y 2 senders y en el cual el mote sender más lejano llega al sink a través del otro mote sender.

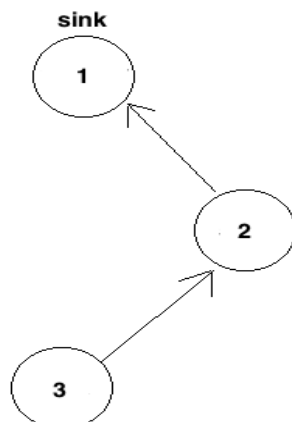


Figura 68: Topología de las simulaciones

En la siguiente tabla se muestran los datos más relevantes de las simulaciones:

Simulación	Período de muestreo (segundos)	RDC promedio (%)		Consumo promedio (mW)		Beacon Interval	
		Mote 2	Mote 3	Mote 2	Mote 3	Mote 2	Mote 3
rpl-collect	600	0.632	0.680	0.604	0.631	34 min 43 sec	25 min 58 sec
	900	0.626	0.640	0.597	0.606	34 min 18 sec	31 min 41 sec
ContikiWSN	600	0.640	0.686	0.609	0.635	34 min 49 sec	21 min 00 sec
	900	0.625	0.648	0.599	0.611	34 min 46 sec	28 min 04 sec

Tabla 9: Comparación de resultados simulados

Los resultados reflejados en el trabajo con los motes TinyOS indican que el RDC obtenido fue de 0.74 %. Cabe resaltar que la distancia entre los motes en dicho trabajo era de entre 50 y 70 metros y por un tiempo de 3 meses, mientras que en las simulaciones la distancia era alrededor de 30 metros y un tiempo de aproximadamente 1 día.

Como se aprecia en los resultados obtenidos en las simulaciones, las aplicaciones rpl-collect y ContikiWSN tienen un RDC y un consumo promedio muy semejantes. Si bien los resultados pueden variar de una simulación a otra, los valores eran muy similares a los que se muestran en la tabla 17.

En la mayoría de las simulaciones el consumo en la aplicación ContikiWSN era apenas superior al de la aplicación rpl-collect. Esto era de esperarse, ya que en la aplicación ContikiWSN se realizan envíos de mensajes Trickle lo que hace que el consumo aumente, aunque en no en gran medida como se observa en los resultados desplegados.

En las pruebas que se realizaron durante dos semanas con los motes, los resultados son aún mejores como se muestran en la tabla 15. Esta mejora se puede atribuir a que las pruebas fueron realizadas en una casa y que las distancias entre los motes eran de 4 o 5 metros, muy inferiores a las del trabajo realizado por docentes de IIE y como también a las de las simulaciones de alrededor de 40 metros.

Capítulo 8: Conclusiones

En el transcurso del proyecto se nos presentaron distintos tipos de dificultades. El hecho de que se trate de un sistema en desarrollo permanente es un problema al trabajar con versiones intermedias ya que se trata de código que no es definitivo y por lo tanto resulta confuso e incluso puede tener fallas. Además el código de Contiki no está suficientemente comentado y en muchos casos no fue fácil entender la lógica de algunos módulos. El Sistema Operativo Contiki carece de documentación práctica del sistema y tutoriales. Los artículos académicos si bien presentan información muy valiosa, esta carece del detalle necesario para estudiar el sistema en profundidad.

Al inicio del proyecto se planificaron plazos de tiempos y metas las cuales no fueron reales debido a que no teníamos conocimiento de a que nos estábamos enfrentando. Dividimos el proyecto en tres etapas. La primer etapa consistía en el Estudio de Contiki, se estudiaron los papers disponibles pero estos no nos dieron una idea practica de como funcionaba Contiki. Para eso tuvimos que leer el código que nos requirió más trabajo del esperado al ser complejo y no estar debidamente documentado. Las dos siguientes etapas eran el desarrollo de la aplicación y la optimización del consumo. Debido a que estábamos cortos de tiempo se decidió pedir 2 meses de prorroga y dividirnos en dos grupos, un integrante se encargaría de la aplicación y los otros dos del estudió de la optimización del consumo. De esa manera pudimos terminar el proyecto exitosamente.

Fue posible cumplir con los objetivos planteados. Se generó una fuente de información para el estudio de Contiki. Además se estudió el consumo de Contiki en una aplicación de red. Se desarrolló una aplicación utilizando el sistema operativo Contiki y el protocolo IPv6, incorporándole nuevas funcionalidades al sistema. Entre las funcionalidades más destacadas se encuentra la diseminación de datos a partir de Trickle, el mejoramiento de la robustez del sistema al poder almacenar datos cuando se pierde la comunicación y enviarlos luego cuando esta se retoma, y la inclusión de la aplicación shell.

Al tratarse de un sistema en desarrollo en el futuro se podría seguir profundizando en su estudio y ampliar la documentación generada. Se pueden tomar dos perspectivas para encarar trabajos futuros. Por un lado la del estudio de la red y por otra la del análisis del consumo.

Dentro de la primera podemos citar la posibilidad de migrar Deluge [34]al stack IPv6 ya que actualmente está implementado sobre Rime e implementar una red que lo utilice. Actualmente RPLContiki implementa solamente el modo de ruteo Storing puede resultar de interés implementar el modo Non-Storing.

En relación a la estimación de consumo podemos citar el estudio y prueba de Powertrace que es un mecanismo de estimación de consumo de manera selectiva de manera que permite estudiar un tipo de tráfico específico. Por otra parte, es necesario validar el mecanismo de estimación de consumo Energest ya que el propio artículo que lo presenta deja abierto a mejoras ese punto.

También es interesante evaluar la capa "Beacon coordination" [14] que ya fue presentada en un artículo recientemente pero de la que aún no se conoce implementación en Contiki. Se trata de una capa en el stack cuya función será la de unificar y administrar el envío de los distintos beacons generados por distintas protocolos de las capas superiores de manera de producir un ahorro en las comunicaciones.

Referencias

[1] Tmote Sky Datasheet (2/6/2006).

[2] Contiki Wiki: http://www.sics.se/contiki/wiki/index.php/Main_Page

[3] Adam Dunkels, Björn Grönvall, Thiemo Voigt. Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, Florida, USA, November 2004.

[4] Adam Dunkels' homepage: http://www.sics.se/~adam/

[5] Adam Dunkels. Poster Abstract: Rime – A Lightweight Layered Communication Stack for Sensor Networks. In Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands, January 2007.

[6] Adam Dunkels. SICSlowpan – Internet-Connectivity for Low-power Radio Systems, 2010.

[7] RFC 4919, IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals, http://tools.ietf.org/html/rfc4919

[8] Neighbor Discovery Optimization for Low Power and Lossy Networks (6LoWPAN), http://tools.ietf.org/wg/6lowpan/draft-ietf-6lowpan-nd/

[9] Gutierrez JA, Naeve M, Callaway E, Bourgeois M, Mitter V, Heile B. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. IEEE *Network Magazine*. September/October 2001; 15(5):12-19.

[10] Johan Lönn, Jonas Olsson. ZigBee for wireless networking. This report is the result of a Master Thesis work done at Linköping University 2005.

[11] Germán Ferrari “Datalogger 2010” - Documentación del Proyecto “Datalogger 2010” para el Curso Sistemas Embebidos en Tiempo Real del Instituto de Ingeniería Eléctrica de la Universidad de la República.

[12] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Nicolas Finne, Thiemo Voigt. Cross-Level Sensor Network Simulation with COOJA. In Proceedings of Real-Time in Sweden 2007, Västerås, Sweden, August 2007.

[13] Contiki Wiki: http://www.sics.se/contiki/wiki/index.php/ContikiMAC

[14] Adam Dunkels, Luca Mottola, Nicolas Tsiftes, Fredrik Österlind, Joakim Eriksson, and Niclas Finne. *The Announcement Layer: Beacon Coordination for the Sensornet Stack*. In Proceedings of EWSN 2011, Bonn, Germany, February 2011.

[15] http://www.sics.se/~adam/dunkels11contikimac.pdf

[16] http://www.sics.se/contiki/wiki/index.php/ContikiMAC

[17] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In Proceedings of the International Conference on Embedded Networked Sensor

Systems (ACM SenSys), pages 95–107, Baltimore, MD, USA, 2004. ACM Press.

[18] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), pages 307–320, Boulder, Colorado, USA, 2006.

[19] A. El-Hoiydi, J.-D. Decotignie, C. C. Enz, and E. L. Roux. wisemac, an ultra low power mac protocol for the wisenet wireless sensor network. In Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), pages 302–303, 2003.

[20] D. Moss and P. Levis. BoX-MACs: Exploiting Physical and Link Layer Boundaries in LowPower Networking. Technical Report SING-08-00, Stanford University, 2008.

[21] Adam Dunkels. *Full TCP/IP for 8 Bit Architectures*. In Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003), San Francisco, May 2003.

[22] Nicolas Tsiftes, Joakim Eriksson, Adam Dunkels. Poster Abstract: Low-Power Wireless IPv6 Routing with ContikiRPL. In Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010), Stockholm, Sweden, April 2010.

[23] RPL: IPv6 Routing Protocol for Low power and Lossy Networks, http://tools.ietf.org/wg/roll/draft-ietf-roll-rpl/

[24] RFC 6282, Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks, http://tools.ietf.org/html/rfc6282

[25] IETF-ROLL, *Routing Over Low power and Lossy networks (roll)*, http://datatracker.ietf.org/wg/roll/charter

[26] RFC 5548, *Routing Requirements for Urban Low-Power and Lossy Networks*, http://tools.ietf.org/html/rfc5548

[27] RFC 5673, *Industrial Routing Requirements in Low-Power and Lossy Networks*, http://tools.ietf.org/html/rfc5673

[28] RFC 5826, *Home Automation Routing Requirements in Low-Power and Lossy Networks*, http://tools.ietf.org/html/rfc5826

[29] RFC 5867, Building Automation Routing Requirements in Low-Power and Lossy Networks, http://tools.ietf.org/html/rfc5867

[30] RFC 6206, *The Trickle Algorithm*, http://tools.ietf.org/html/rfc6206

[31] http://subversion.tigris.org/

[32] Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He. *Software-based on-line energy estimation for sensor nodes*. In Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV), Cork, Ireland, June 2007.

[33] P. Mazzara, L. Steinfeld, J. Villaverde, F. Silveira, G. Fierro, A. Otero, C. Saravia, N. Barlocco, P.

Vergara, D. Garín. Despliegue y Depuración de Redes de Sensores Inalámbricos para Aplicaciones al Agro. 2011.

[34] Jonathan W. Hui y David Culler, *The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale*, Proceedings of the 2nd international conference on Embedded networked sensor systems Computer Science Division, University of California at Berkeley (SenSys 2004).

Anexo I: Implementación de ContikiOS

Implementación de Contiki

Protothreads

Los protothreads son generados con macros del lenguaje C. Estos macros son sustituidos por el procesador con código C de manera que el protothread se comporte como lo esperado.

Dependiendo del compilador se implementan los protothreads utilizando goto o switch cases. Cuando se utiliza los switch cases el usuario no puede agregar switch case dentro de los protothreads.

Hay que tener cuidado con el uso de las variables locales en los protothreads debido a que Contiki no las guardara cuando se retorne del protothread, por lo tanto se recomienda solo usar variables globales o estáticas de la propia función.

A continuación se muestra el ejemplo "Hola mundo" con la implementación de los protothreads usando switch cases.

Implementación de Protothread Hola mundo en Contiki

```
PROCESS(hello_world_process, "Hello world process");

PROCESS_THREAD(hello_world_process, ev, data){
  static struct etimer etimer1;

  PROCESS_BEGIN();
  etimer_set(&etimer1, CLOCK_CONF_SECOND);

  for(;;){
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
    printf("Hello, world\n");
    etimer_reset(&etimer1);
  }

  PROCESS_END();
}
```

Código luego del pre-procesador

```
static char process_thread_hello_world_process(struct pt *process_pt,
                                               process_event_t ev, process_data_t data);

struct process hello_world_process = { 0, "Hello world process",
                                       process_thread_hello_world_process };

static char process_thread_hello_world_process(struct pt *process_pt,
                                               process_event_t ev, process_data_t data){
   static struct etimer etimer1;

   char PT_YIELD_FLAG = 1;
   switch((process_pt)->lc){
   case 0:;
      etimer_set(&etimer1, 128);
      for(;;){
         do { PT_YIELD_FLAG = 0; (process_pt)->lc = 28;
   case 28:;
         if((PT_YIELD_FLAG == 0) || !(ev == 0x88)) { return 1; } } while(0);
         printf("Hello, world\n");
         etimer_reset(&etimer1);
      }

   };
   PT_YIELD_FLAG = 0; (process_pt)->lc = 0;; return 3;
}
```

Este protothread lo que hará es setear un etimer para que genere un evento cada 1 segundo que lo despierte. Luego, cada vez que sea despertado por este evento imprime el texto “hola mundo”.

PROCESS(hello_world_process, "Hello world process")

Esta macro lo que hace es declarar la función del protothread y también define un proceso asociándole el protothread declarado. Se le debe ingresar como primer parámetro el nombre del proceso que se usara en el código C y como segundo parámetro un string que contenga un nombre con el cual el usuario identifique el proceso (este no se usará en el código).

Primera sentencia generada:

```
static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);
```

Esta sentencia no hace más que declarar la función process_thread_hello_world_process, la cual luego se definirá. Observar que el nombre de esta función surgió de la concatenación de “process_thread_” + el primer parámetro que se le ingreso a la macro.

Segunda sentencia generada:

```
struct process hello_world_process = { 0, "Hello world process", process_thread_hello_world_process };
```

Para entender esta sentencia hay que conocer la estructura de los procesos:

```
struct process {
  struct process *next;
  const char *name;
  PT_THREAD((* thread)(struct pt *, process_event_t, process_data_t));
  struct pt pt; //indica el lc
  unsigned char state, needspoll;
};
```

Cada proceso está compuesto por un puntero al próximo proceso, una cadena de caracteres que indica su nombre, un puntero a la función protothread, una estructura pt (que no es más que un entero cuyo número memoriza en que parte del protothread se debe entrar la próxima vez que éste se ejecute), el estado del proceso y una bandera para indicar si este necesita poll.

Como se puede notar, esta segunda sentencia generada lo único que inicializó fue la asociación del proceso hello_world_process al protothread process_thread_hello_world_process y la cadena de caracteres "Hello world process". El puntero al próximo proceso se definió "0" y éste se inicializa cuando el proceso se activa y se guarda en la lista encadenada.

PROCESS_THREAD(hello_world_process, ev, data)

Sentencia generada:

```
static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t
data){
```

Esta macro se usa para comenzar la función protothread. Lo único que hace es escribir en código C la cabecera de la función. Al primer parámetro de la macro se le tiene que pasar el nombre del proceso al cual pertenecerá el protothread que se está elaborando, el segundo y el tercero se deben dejar incambiados y expresan los nombres del segundo y tercer parámetro de la función que se está elaborando.

Los parámetros de este nuevo protothread que se tienen que pasar cada vez que éste se llame son: una estructura que contiene al entero que indica en qué parte de la función se debe comenzar (la estructura es process_pt, y el entero que contiene se llama "lc"), el evento que despertó la función (ev) y los datos que éste le envía (data).

PROCESS_BEGIN()

Sentencia generada:

```
char PT_YIELD_FLAG = 1;
  switch((process_pt)->lc){
  case 0:;
```

Siempre que se comienza el protothread se hace un switch del entero lc, el cual contiene la línea del protothread que se debe ejecutar. La primera vez que se ejecute el lc contendrá el valor cero, por lo tanto se empezara por el case 0. Además se seteará a 1 la flag PT_YIELD_FLAG que se usara para saber si se tiene que retornar del waitForEvent().

La primera vez que se ejecute el protothread se seteará el timer para que genere un evento PROCESS_EVENT_TIMER dentro de un segundo y despierte al proceso.

PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);

Sentencia generadas:

```
do { PT_YIELD_FLAG = 0; (process_pt)->lc = 28;
case 28:;
if((PT_YIELD_FLAG == 0) || !(ev == 0x88)) { return 1; } } while(0);
```

La idea es para todas las líneas que se puede retornar por un WaitForEvent() memorizar en el entero lc el numero de línea, y a la vez hacer un case: "numero de línea". Por lo tanto, si retorno en el WaitForEvent() esperando a un evento, cuando este evento aparezca y despierte al protothread, éste comenzará con el switch del lc, el cual indicará que se ejecute a partir de la última línea que se había quedado.

El PROCESS_WAIT_EVENT_UNTIL lo primero que hace es poner PT_YIELD_FLAG a 0 indicado que se va a tener que retornar, ya que hay que esperar al evento de timer, luego memoriza en el lc el número de la línea (en este caso la línea fue la 28). Luego pregunta si la PT_YIELD_FLAG es 0, lo cual es cierto porque se acaba de poner a dicho valor y por lo tanto retorna con el valor 1. La próxima vez que un evento despierte el protothread, éste seteará la PT_YIELD_FLAG a 1 y hará un switch del lc, el cual tendrá su valor en 28 y por lo tanto seguirá en el case 28. Esta sentencia lo que hace es si PT_YIELD_FLAG es 0 o si el evento es distinto al que se está esperando vuelve a retornar 1. Si el evento es el esperado se sigue (ya que PT_YIELD_FLAG = 1), luego se imprime "Hello, world" y se resetea el timer. Esto lo que hace es decirle al timer que vuelva a esperar 1 segundo desde que expiró. El for(;;) es un loop infinito para que este procedimiento se repita indefinidamente sin que nunca se ejecute el PROCESS_END().

PROCESS_END()

Sentencia generada:

```
PT_YIELD_FLAG = 0; (process_pt)->lc = 0;; return 3;
```

Esta sentencia lo que hace es retornar el valor 3, el cual se indica que el proceso se debe destruir.

Main

Para que el programa hello world funcione no basta solamente con implementar el protothread. También hay que implementar el main.

Implementación del main

int main(void){
// Stop watchdog timer to prevent time out reset
WDTCTL = WDTPW + WDTHOLD;
 	
/* Inicializo el Timer A*/
clock_init();
 	
/* Inicializo manejador de procesos */
process_init();
 	
/* Primero inicia proceso etimer */
process_start(&etimer_process, NULL);
 	
/*Inicializa proceso hola mundo*/

```
    process_start(&hello_world_process,NULL);

    for(;;){
        do {
          /* Reset watchdog. */
          watchdog_periodic();
          r = process_run();
        } while(r > 0);
        __bis_SR_register(LPM3_bits);
    }
}
```

Lo primero que se hace es parar el watch dog timer (WDTCTL = WDTPW + WDTHOLD), luego inicializar los registros del timerA para que se pueda utilizar los etimers (clock_init();).

La rutina process_init() lo que hace es inicializar la cantidad de eventos de la cola circular a 0 y definir la lista de procesos encadenados vacía.

Luego hay que inicializar los procesos que van a ser usados, el etimer_process y el hello_world_process. La rutina process_start() lo que hace es agregar el proceso a la lista encadenada, setearle el puntero al próximo proceso de la lista, el estado del proceso se pasa a activo, se inicializa el lc a 0 (para que corra el case 0 del protothread) y se hace un post sincrónico (se ejecuta de inmediato el protothread por primera vez) con el evento "PROCESS_EVENT_INIT". Esto hace que se corra el protothread por primera vez con el lc en 0 y con el evento de inicialización. Por ejemplo, en el proceso hello world, con esta primer ejecución, se seteará el timer y se dejara el protothread esperando por el evento de etimer.

Luego se hace un loop infinito, el cual pregunta si hay algún evento o poll para procesar y en caso contrario se pasa a un modo de bajo consumo que deja solo corriendo el reloj de 32 kHz.

La rutina process_run() primero pregunta si hay algún poll para procesar. En caso afirmativo recorre proceso por proceso viendo cual o cuales de los procesos requirieron el poll. Llamándolos con el evento "PROCESS_EVENT_POLL".

Luego del procesamiento de polls se procesará también un evento de la cola circular de eventos pendientes; el más antiguo cronológicamente. Se llamará al proceso con motivo de dicho evento y se le enviará datos si corresponde.

Manejo de datos compartidos

Las rutinas de atención a la interrupción (ISR) sólo deben hacer el trabajo estrictamente necesario para atender el hardware ya que deben tardar poco tiempo en ejecutarse para reducir la latencia. Por lo tanto, éstas dejan el procesamiento de datos para las tareas en segundo plano. Por lo tanto es necesario que se comuniquen las ISR con las tareas de segundo plano. El problema de datos compartidos surge cuando en el medio de la lectura de los datos compartidos se ejecuta la interrupción y se los cambian. Por lo tanto, esto genera que los datos sean erróneos.

La solución que implementa Contiki para este problema es la de dos lecturas sucesivas. Si dos lecturas sucesivas de la variable compartida dan el mismo resultado es porque no se ha producido una interrupción durante la lectura y por lo tanto el valor es válido. En caso que los valores den distintos se vuelven a hacer otras dos lecturas sucesivas.

Estructura de Carpetas

La siguiente figura muestra la estructura de carpetas de Contiki.

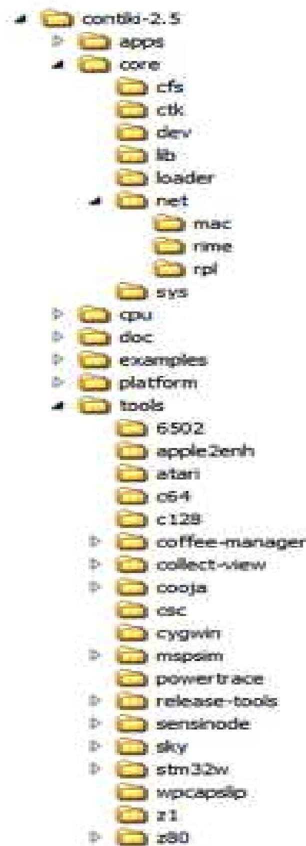


Figura 69: Estructura de carpetas de Contiki

En el primer nivel tenemos:

- **apps**: Las aplicaciones son programas auxiliares que se pueden usar en el programa principal. Hay que tener cuidado de que no se puede usar cualquier aplicación con cualquier protocolo. Por ejemplo, hay muchas aplicaciones que están diseñadas para el stack Rime y no funcionan para uIP.
- **core**: Como su nombre lo indica en este directorio se encuentra el core del sistema operativo. En este directorio se pueden encontrar la implementación de los protothreads, los etimers, uIP, Rime, RPL y todo lo que refiere al sistema operativo.
- **cpu**: En esta carpeta se encuentra todo el código que depende del microcontrolador. Algunos ejemplos pueden ser la configuración de los timer, el módulo serial, la memoria flash, las instrucciones para que el microcontrolador entre y salga del low-power mode. En nuestro caso se usa el msp430 F1611, por lo que se utiliza la carpeta msp430. Se puede cambiar fácilmente de microcontrolador eligiendo que se compile el código de una carpeta diferente.
- **doc**: Aquí se encuentran todo lo que refiere a documentación del código.
- **examples**: En esta carpeta se encuentran los programas de ejemplo que utilizan el sistema operativo Contiki.

- **platform**: En dicha carpeta se encuentra todo el código que depende de la plataforma, como ser la configuración de los botones y los sensores. Además en esta carpeta se encuentra la rutina main.

tools: En esta carpeta se encuentra varias herramientas extras que no forman parte del sistema operativo Contiki. Por ejemplo, esta carpeta tiene software complementario como el collect-view y el simulador COOJA [12], los cuales se compilan por separado al sistema operativo Contiki.

Anexo II: Estructura de los mensajes de Control de RPL

ICMPv6 RPL Control Message

El draft define el Mensaje de Control de RPL, que se identifica con un código, y formado por información que depende de ese código, y una serie de opciones.

La mayoría de los Mensajes de Control de RPL tienen el alcance de un enlace. La única excepción es la de los mensajes DAO y DAO-ACK en el modo de non storing, que se intercambian mediante una dirección única en múltiples saltos y por lo tanto utiliza direcciones globales o locales únicas, tanto para el origen como para el destino. Para todos los mensajes de control de RPL, la dirección de origen es una dirección de enlace local, y la dirección de destino es una dirección de multidifusión RPL o una dirección local unicast. La dirección de multidifusión de todos los nodos RPL es una dirección nueva con un valor de FF02:: 1 A, de reciente adjudicación por parte de la IANA.

De acuerdo con el RFC4443 (Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification), el mensaje de control RPL se compone de una cabecera ICMPv6 seguida por un cuerpo de mensaje. El cuerpo del mensaje se compone de una base de mensajes y, posiblemente, una serie de opciones como se muestra en la figura siguiente (las figuras 15, 16, 17, 18 y 19 fueron sacadas de: RPL: IPv6 Routing Protocol for Low power and Lossy Networks, http://tools.ietf.org/wg/roll/draft-ietf-roll-rpl/),

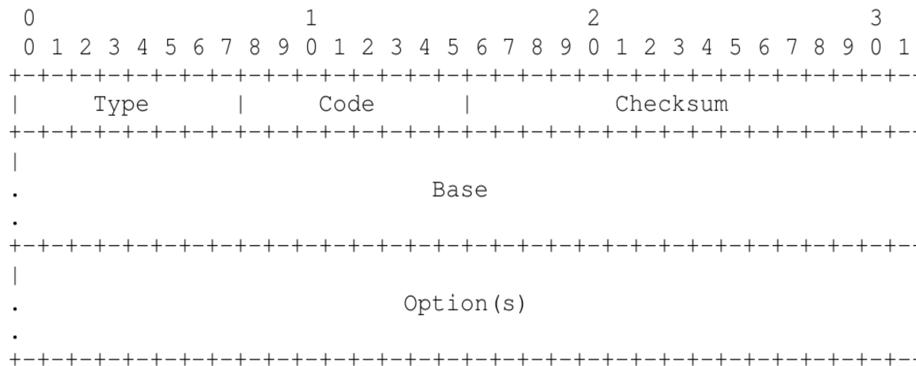


Figura II.1: RPL Control Message

El campo Code identifica el tipo de mensaje de control de RPL. El draft [19] define los códigos de los siguientes tipos de mensajes de control de RPL (todos los códigos han de ser confirmados por la IANA):

- 0x00: DODAG Information Solicitation (DIS)
- 0x01: DODAG Information Object (DIO)
- 0x02: Destination Advertisement Object (DAO)
- 0x03: Destination Advertisement Object Acknowledgment
- 0x80: Secure DODAG Information Solicitation
- 0x81: Secure DODAG Information Object
- 0x82: Secure Destination Advertisement Object
- 0x83: Secure Destination Advertisement Object Acknowledgment
- 0x8A: Consistency Check

En nuestro caso los mensajes más relevantes son los tres primeros, DIS, DIO y DAO, debido a que

fueron los que utilizamos para el análisis de tráfico en el laboratorio y para desarrollar la aplicación.

“DODAG Information Solicitation” - DIS

El mensaje DIS se pueden utilizar para solicitar un objeto de información de un nodo DODAG RPL. Su uso es análogo al de una solicitud del router como se especifica en “IPv6 Neighbor Discovery”; un nodo puede utilizar los mensajes DIS para investigar su entorno.

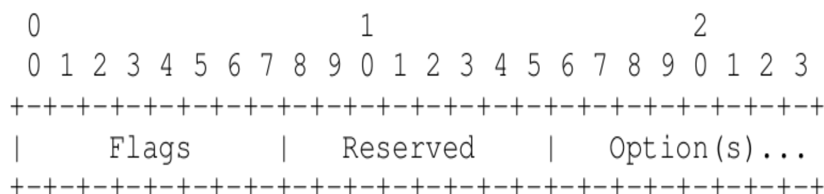


Figura II.2: Mensaje DIS

“DODAG Information Object” - DIO

El mensaje DIO lleva la información que permite al nodo que lo recibe descubrir una instancia de RPL, conocer sus parámetros de configuración, elegir un conjunto de padres en el DODAG, y mantener la información DODAG

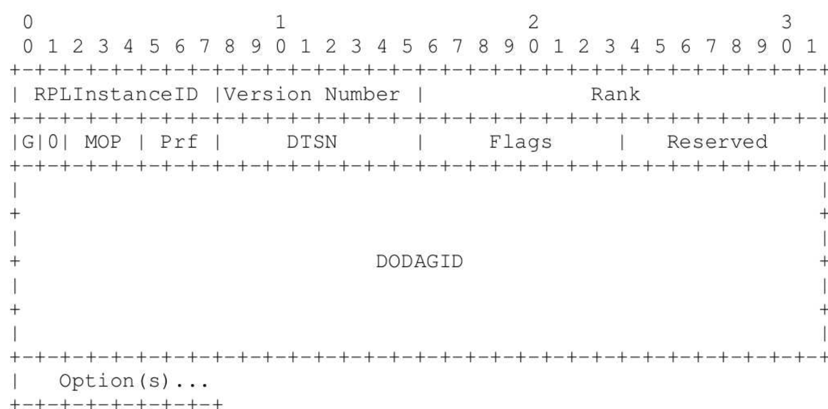


Figura II.3: DIO Base Object

RPLInstanceID	Campo de 8-bit establecido por el nodo raíz que indica la Instancia RPL del DODAG a la que pertenece.
Version Number	Campo de 8 bits sin signo establecidos por el nodo raíz del DODAG en el DODAGVersionNumber.
Rank	Entero sin signo de 16 bits que indica el rank del nodo que envía el mensaje DIO.
DTSN	Entero sin signo de 8 bits configurado por el nodo originador del mensaje DIO. El flag Destination Advertisement Trigger Sequence Number se utiliza en el procedimiento para mantener las rutas aguas abajo.
DODAGPreference (PRF)	Un entero sin signo de 3 bits que define la preferencia del nodo raíz de este DODAG y lo compara con otros nodos raíz en la instancia del DODAG. El rango de DAGPreference va de 0x00 (menos preferido) a 0x07 (preferido). El valor predeterminado es 0 (menos preferido).

Mode of Operation (MOP)	El campo MOP identifica el modo de funcionamiento de la instancia RPL que se suministra y es distribuido por el nodo raíz del DODAG. Todos los nodos que se unen al DODAG debe ser capaz de cumplir el MOP para poder participar plenamente como un router, o de lo contrario sólo podrá unirse como una hoja.

MOP se codifica como en la figura siguiente:

MOP	Significado
0	RPL no mantiene rutas aguas abajo
1	No guarda información
2	Guarda información sin soporte multicast
3	Guarda información con soporte multicast

Los demás valores no están asignados.

“Destination Advertisement Object” (DAO)

Los mensajes DAO se utilizan para propagar aguas arriba información sobre los destinos en el DODAG. En el modo de storing el mensaje de DAO es enviado por unicast desde el nodo hijo al padre seleccionado. En el modo non storing el mensaje DAO es enviado por unicast al nodo raíz del DODAG. El mensaje de DAO puede opcionalmente, a petición explícita ser reconocido por su destino con un mensaje de confirmación (DAO-ACK) dirigido al remitente del mensaje DAO.

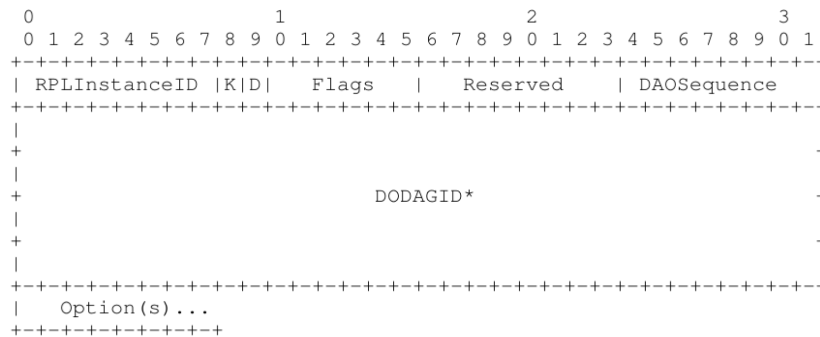


Figura II.4: DAO Base Object

- RPLInstanceID: campo de 8 bits que indica la instancia de topología asociada al DODAG, tal como se descubrió del DIO.
- K: indica que el receptor espera que le envíen una copia de DAO-ACK.
- D: Indica que el campo DODAGID está presente.
- Reserved: Campo no utilizado de 8 bits. El campo debe ser inicializado a cero por el remitente y debe ser ignorado por el receptor.
- DAOSequence: Se incrementa en cada mensaje DAO de un nodo y se confirma en el mensaje DAO-ACK.
- DODAGID (opcional): Es un entero de 128 bits sin signo fijado por el nodo raíz del DODAG,

identifica un DODAG unívocamente. Este campo sólo está presente cuando la bandera 'D' está activa.

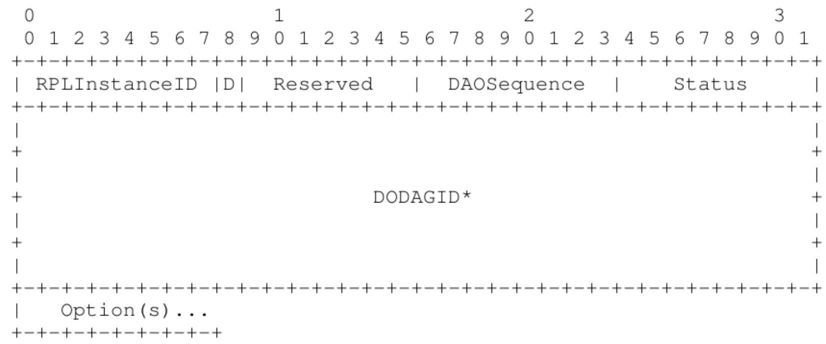


Figura II.5: DAO ACK Base Object

Anexo III: Ejemplo de Perdida de comunicación

El nodo 2 comienza entre el nodo 3 y el 1. Luego se retira según se muestra en la siguiente Figura.

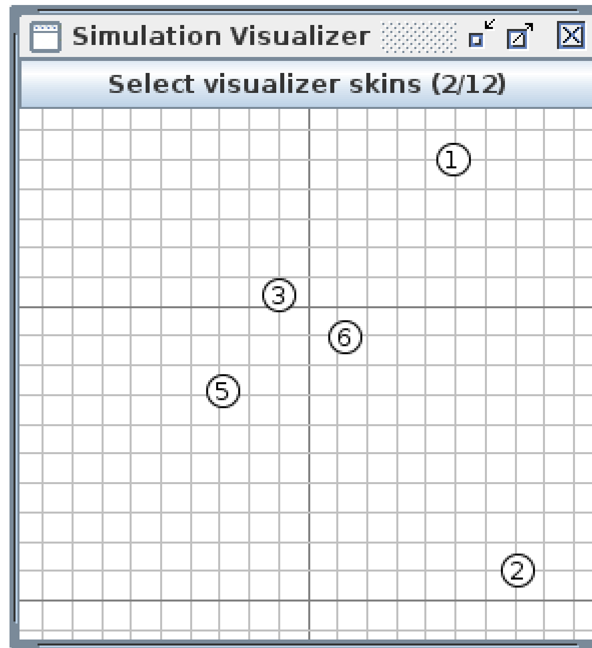


Figura III.1: Topología de Perdida de Comunicación

El nodo 3 pierde a su best parent por lo tanto estamos en el CASO 1 de pérdida de comunicación. La siguiente tabla muestra su comportamiento:

NODO 3 - CASO 1		
T(ms)	**ID**	**Evento**
El link del nodo 3 con su preferred parent (nodo 2) es bueno. Se envía un paquete al nodo 2 y se recibe el ack con el primer intento (numtx=1) y el status es 0 (MAC_TX_OK). Se pasa el ETX con el nodo 2 al valor 1.		
11524	ID: 3	neighbor-info: packet sent to 2.2, status=0, numtx=1
11527	ID: 3	neighbor-info: The neighbor is already known
11533	ID: 3	neighbor-info: ETX changed from 15 to 1 (packet ETX = 1) 2
El enlace entre el nodo 3 y el nodo 2 es de buena calidad no hay perdidas. El valor link_metric = 16 (es el mínimo). Entonces se calcula el rank del nodo 3, tomando como base el de preferred parent (512) y se lo incrementa en 1 paso discreto (256)		
17631	ID: 3	rpl-of-etx: link_metric 16, min_hoprankinc 256, address fe80::212:7402:2:202
17638	ID: 3	rpl-of-etx: new_rank 768, base_rank 512, rank_increase 256
Se vuelve a recibir respuesta del nodo 2.		
18402	ID: 3	neighbor-info: packet sent to 2.2, status=0, numtx=1

<table>
<tr><td>18405</td><td>ID: 3</td><td>neighbor-info: The neighbor is already known</td></tr>
<tr><td>18410</td><td>ID: 3</td><td>neighbor-info: ETX changed from 1 to 1 (packet ETX = 1) 2</td></tr>
<tr><td colspan="3">Se envía a todos los vecinos un mensaje DIO y se le avisa que el rank del nodo 3 es 768</td></tr>
<tr><td>120368</td><td>ID: 3</td><td>RPL: Sending prefix info in DIO for aaaa::</td></tr>
<tr><td>120372</td><td>ID: 3</td><td>RPL: Sending a multicast-DIO with rank 768</td></tr>
<tr><td colspan="3">SE QUITA NODO 2 - Se envía paquete al nodo 2 pero no se recibe ninguna respuesta. Se enviaron 5 intentos (numtx=5) pero no se recibió ack (status 2 = MAC_TX_NOACK). El ETX con el nodo 2 paso de 1 a 2. Ojo no confundir (packet ETX = 15) con el valor de ETX. El valor 15 en el packet ETX indica que se perdió comunicación y se va a penalizar el valor de la métrica (ETX_NOACK_PENALTY = 15).</td></tr>
<tr><td>256282</td><td>ID: 3</td><td>neighbor-info: packet sent to 2.2, status=2, numtx=5</td></tr>
<tr><td>256287</td><td>ID: 3</td><td>neighbor-info: ETX changed from 1 to 2 (packet ETX = 15) 2</td></tr>
<tr><td colspan="3">La métrica del link con el nodo 2 se empeora en un paso (256), se incrementa de 256 a 512. Por otro lado también se puede observar que el rank del preferred parent (nodo 2) sigue siendo el mismo (512) ya que como no se recibió mensajes de este no se actualizó.</td></tr>
<tr><td>257224</td><td>ID: 3</td><td>rpl-of-etx: link_metric 38, min_hoprankinc 256, address fe80::212:7402:2:202</td></tr>
<tr><td>257231</td><td>ID: 3</td><td>rpl-of-etx: new_rank 1024, base_rank 512, rank_increase 512</td></tr>
<tr><td colspan="3">El DAG rank se pasa de 3 a 4 el nodo 3 esta un paso más lejos del sink, el cálculo es 1024/256 = 4)</td></tr>
<tr><td>257235</td><td>ID: 3</td><td>RPL: Moving in the DAG from rank 3 to 4</td></tr>
<tr><td>257244</td><td>ID: 3</td><td>RPL: The preferred parent is fe80::212:7402:2:202 (rank 2)</td></tr>
<tr><td colspan="3">Se envía a todos los vecinos un mensaje DIO y se le avisa que el rank del nodo 3 es 1024</td></tr>
<tr><td>259507</td><td>ID: 3</td><td>RPL: Sending a multicast-DIO with rank 1024</td></tr>
<tr><td colspan="3">Se vuelve a perder otro mensaje contra el nodo 2 y se vuelve a penalizar el valor de la métrica del link.</td></tr>
<tr><td>292109</td><td>ID: 3</td><td>neighbor-info: packet sent to 2.2, status=2, numtx=5</td></tr>
<tr><td>292115</td><td>ID: 3</td><td>neighbor-info: ETX changed from 2 to 3 (packet ETX = 15) 2</td></tr>
<tr><td>292225</td><td>ID: 3</td><td>rpl-of-etx: link_metric 58, min_hoprankinc 256, address fe80::212:7402:2:202</td></tr>
<tr><td>292232</td><td>ID: 3</td><td>rpl-of-etx: new_rank 1280, base_rank 512, rank_increase 768</td></tr>
<tr><td>292236</td><td>ID: 3</td><td>RPL: Moving in the DAG from rank 4 to 5</td></tr>
<tr><td>295728</td><td>ID: 3</td><td>RPL: Sending prefix info in DIO for aaaa::</td></tr>
<tr><td>295733</td><td>ID: 3</td><td>RPL: Sending a multicast-DIO with rank 1280</td></tr>
<tr><td colspan="3">Este proceso se repite múltiples veces…</td></tr>
<tr><td>310242</td><td>ID:</td><td>neighbor-info: packet sent to 2.2, status=2, numtx=5</td></tr>
</table>

<table>
<tr><td></td><td>3</td><td></td></tr>
<tr><td>310247</td><td>ID: 3</td><td>neighbor-info: ETX changed from 3 to 4 (packet ETX = 15) 2</td></tr>
<tr><td>310261</td><td>ID: 3</td><td>rpl-of-etx: link_metric 76, min_hoprankinc 256, address fe80::212:7402:2:202</td></tr>
<tr><td>310268</td><td>ID: 3</td><td>rpl-of-etx: new_rank 1536, base_rank 512, rank_increase 1024</td></tr>
<tr><td>310272</td><td>ID: 3</td><td>RPL: Moving in the DAG from rank 5 to 6</td></tr>
<tr><td>310281</td><td>ID: 3</td><td>RPL: The preferred parent is fe80::212:7402:2:202 (rank 2)</td></tr>
<tr><td colspan="3">A pesar que se ha deteriorado el rank y se perdió comunicación con el nodo 2. Se sigue intentando mandar los mensajes para el nodo 1 a través del nodo 2.</td></tr>
<tr><td>345339</td><td>ID: 3</td><td>DS6: Looking up route for aaaa::1</td></tr>
<tr><td>345341</td><td>ID: 3</td><td>DS6: No route found</td></tr>
<tr><td>345348</td><td>ID: 3</td><td>Defrt, IP address fe80::212:7402:2:202</td></tr>
<tr><td>345356</td><td>ID: 3</td><td>Defrt found, IP address fe80::212:7402:2:202</td></tr>
<tr><td colspan="3">Luego de 150s….. se sigue deteriorando el link</td></tr>
<tr><td>495438</td><td>ID: 3</td><td>neighbor-info: packet sent to 2.2, status=2, numtx=5</td></tr>
<tr><td>495444</td><td>ID: 3</td><td>neighbor-info: ETX changed from 7 to 8 (packet ETX = 15) 2</td></tr>
<tr><td>496225</td><td>ID: 3</td><td>rpl-of-etx: link_metric 131, min_hoprankinc 256, address fe80::212:7402:2:202</td></tr>
<tr><td>496233</td><td>ID: 3</td><td>rpl-of-etx: new_rank 2560, base_rank 512, rank_increase 2048</td></tr>
<tr><td>496237</td><td>ID: 3</td><td>RPL: Moving in the DAG from rank 9 to 10</td></tr>
<tr><td>496246</td><td>ID: 3</td><td>RPL: The preferred parent is fe80::212:7402:2:202 (rank 2)</td></tr>
<tr><td colspan="3">De vuelta el nodo 2 no responde y se penaliza la métrica del link, la penalización no alcanza como para aumentar el ETX en un dígito</td></tr>
<tr><td>594892</td><td>ID: 3</td><td>neighbor-info: packet sent to 2.2, status=2, numtx=5</td></tr>
<tr><td>594898</td><td>ID: 3</td><td>neighbor-info: ETX changed from 8 to 8 (packet ETX = 15) 2</td></tr>
<tr><td>595225</td><td>ID: 3</td><td>rpl-of-etx: link_metric 141, min_hoprankinc 256, address fe80::212:7402:2:202</td></tr>
<tr><td>595233</td><td>ID: 3</td><td>rpl-of-etx: new_rank 2560, base_rank 512, rank_increase 2048</td></tr>
<tr><td colspan="3">Se envía a todos los vecinos un mensaje DIO y se le avisa que mi rank es 2560</td></tr>
<tr><td>598748</td><td>ID: 3</td><td>RPL: Sending a multicast-DIO with rank 2560</td></tr>
<tr><td colspan="3">Expira el REACHABLE timer y el nodo 3 pasa del estado REACHABLE a STALE</td></tr>
<tr><td>607221</td><td>ID: 3</td><td>REACHABLE: moving to STALE (fe80::212:7402:2:202)</td></tr>
</table>

En el momento que el nodo 3 intenta enviar un mensaje al sink se pasa el nodo del estado STALE al DELAY. En este estado se empiezan a enviar probes para chequear si hay comunicación con el nodo 2		
642184	ID: 3	DELAY: moving to PROBE + NS 1
El primer PROBE no obtuvo respuesta, se envía el segundo…		
652205	ID: 3	PROBE: NS 2
El segundo PROBE no obtuvo respuesta, se envía el tercero…		
662151	ID: 3	PROBE: NS 3
Tampoco se obtuvo respuesta del tercero, se da por perdido nodo 2.		
662288	ID: 3	PROBE END
Se empieza un DAG repair. En el segundo 662 el rank del nodo pasa a infinito (0xFFFF).		
662293	ID: 3	RPL: Starting a local DAG repair
662296	ID: 3	RPL: Removing parents (minimum rank 0)
662305	ID: 3	RPL: Removing parent fe80::212:7402:2:202
662313	ID: 3	RPL: Removing parent fe80::212:7406:6:606
662321	ID: 3	RPL: Removing parent fe80::212:7405:5:505

El nodo 5 pierde la comunicación con el sink, pero no a su best parent por lo tanto estamos en el CASO 2 de pérdida de comunicación. La siguiente tabla muestra su comportamiento:

NODO 5 - CASO 2		
T(ms)	**ID**	**Evento**
El link del nodo 5 con su preferred parent (nodo 3) es bueno. Se le envía un paquete y se recibe el ack con el primer intento (numtx=1) y el status es 0 (MAC_TX_OK). Se pasa el ETX con el nodo 3 al valor 1.		
18190	ID: 5	neighbor-info: packet sent to 3.3, status=0, numtx=1
18193	ID: 5	neighbor-info: The neighbor is already known
18198	ID: 5	neighbor-info: ETX changed from 15 to 1 (packet ETX = 1) 3
El enlace entre el nodo 5 y el nodo 3 es de buena calidad no hay perdidas. El valor link_metric = 16 (es el mínimo). Entonces se calcula el rank del nodo 3, tomando como base el de preferred parent (768) y se lo incrementa en 1 paso discreto (256)		
18674	ID: 5	rpl-of-etx: link_metric 16, min_hoprankinc 256, address fe80::212:7403:3:303
18681	ID: 5	rpl-of-etx: new_rank 1024, base_rank 768, rank_increase 256
El DAG rank se pasa a 4 el cálculo es 1024/256 = 4		
18685	ID: 5	RPL: Moving in the DAG from rank 8 to 4
18694	ID:	RPL: The preferred parent is fe80::212:7403:3:303 (rank 3)

	5	
Se envía a todos los vecinos un mensaje DIO y se le avisa que el rank del nodo 5 es 1024		
27724	ID: 5	RPL: Sending prefix info in DIO for aaaa::
27728	ID: 5	RPL: Sending a multicast-DIO with rank 1024
256000	**SE QUITA NODO 2**	
El link entre el nodo 5 y 3 sigue siendo bueno. Pero el rank del preferred parent empieza a empeorar por lo tanto el rank del nodo 5 también empieza a empeorar		
259703	ID: 5	rpl-of-etx: link_metric 16, min_hoprankinc 256, address fe80::212:7403:3:303
259710	ID: 5	rpl-of-etx: new_rank 1280, base_rank 1024, rank_increase 256
El DAG rank se pasa de 4 a 5 el nodo 5 esta un paso más lejos del sink, el cálculo es 1280/256 = 5		
259714	ID: 5	RPL: Moving in the DAG from rank 4 to 5
259723	ID: 5	RPL: The preferred parent is fe80::212:7403:3:303 (rank 4)
Se repite este proceso…		
295829	ID: 5	rpl-of-etx: link_metric 16, min_hoprankinc 256, address fe80::212:7403:3:303
295836	ID: 5	rpl-of-etx: new_rank 1536, base_rank 1280, rank_increase 256
295840	ID: 5	RPL: Moving in the DAG from rank 5 to 6
295849	ID: 5	RPL: The preferred parent is fe80::212:7403:3:303 (rank 5)
Se sigue repitiendo el proceso hasta que el rank del preferred parent sea mayor al mínimo rank del nodo 5 sumándole 3 pasos de rank (3 * 256). Entonces según la condición de parada: Rank del Preferred Parent (2048) mayor a el mínimo rank que tuvo el nodo alguna vez (1024) sumado a 3 pasos de rank (3 * 256) => La condición se cumple 2048 > 1792		
370702	ID: 5	rpl-of-etx: link_metric 16, min_hoprankinc 256, address fe80::212:7403:3:303
370709	ID: 5	rpl-of-etx: new_rank 2304, base_rank 2048, rank_increase 256
Se pone el Rank en infinito (0xFFFF) y se empieza un local DAG repair.		
370724	ID: 5	RPL: Sending DAO with prefix aaaa::212:7405:5:505 to fe80::212:7403:3:303
370730	ID: 5	RPL: Removing parents (minimum rank 0)
Se envía a todos los vecinos un mensaje DIO y se le avisa que el rank del nodo 5 es infinito		
374678	ID: 5	RPL: Sending prefix info in DIO for aaaa::
374682	ID: 5	RPL: Sending a multicast-DIO with rank 65535

Anexo IV: Simulador COOJA

Introducción

COOJA es un simulador flexible basado en Java diseñado para simulación de redes de sensores ejecutando el sistema operativo Contiki. COOJA simula redes de nodos sensores donde cada nodo puede ser de un tipo diferente; no difieren sólo en el software, sino también en el hardware simulado. COOJA es flexible en que muchas partes del simulador pueden ser fáciles de sustituir o ampliar con funciones adicionales. Ejemplos de partes que se pueden extender son la simulación del medio de radio, nodo de hardware simulado y plugins para la simulación de entradas/salidas.

Un nodo simulado en COOJA tiene tres propiedades básicas: sus datos de memoria, el tipo de nodo y sus periféricos de hardware. Se pueden crear varios nodos del mismo tipo los cuales compartirán las mismas propiedades. Por ejemplo, los nodos del mismo tipo ejecutan el mismo código del programa en el mismo hardware periférico simulado. Además los nodos del mismo tipo se inicializan con los mismos datos de memoria. Durante la ejecución, sin embargo, los datos de la memoria de los nodos pueden eventualmente diferir debido a por ejemplo diferentes insumos externos.

COOJA es capaz de ejecutar programas en Contiki de dos maneras diferentes. Ejecutando el código del programa como código nativo compilado directamente en la CPU del host, o ejecutando el programa compilado en código de una instrucción a nivel del emulador TI MSP430. COOJA también es capaz de simular nodos no pertenecientes a Contiki, tales como nodos implementados en Java o incluso nodos que ejecutan otros sistemas operativos. Todos los enfoques diferentes tienen ventajas y desventajas.

- Nodos basados en Java permiten simulaciones mucho más rápidas, pero no ejecutan código de despliegue. Por lo tanto, son útiles para el desarrollo por ejemplo, de algoritmos distribuidos.
- Nodos emulados proporcionan detalles de ejecución más finos en comparación con nodos basados en Java o nodos que ejecutan código nativo.
- Simulaciones de código nativo son más eficientes que las emulaciones de nodos y aún así simulan el código de despliegue.

Puesto que la necesidad de la abstracción entre los diferentes nodos simulados de una red heterogénea pueden ser diferentes, hay ventajas en la combinación de varios niveles diferentes de abstracción en una simulación. Por ejemplo, en una red simulada unos pocos grandes nodos pueden ser simulados a nivel de hardware, mientras que el resto se aplican en el nivel de Java puro. Con este enfoque se combinan las ventajas de los distintos niveles. La simulación es más rápida que cuando se emulan todos los nodos, pero al mismo tiempo permiten a un usuario recibir detalles finos de los pocos nodos emulados.

COOJA ejecuta código nativo utilizando Java Native Interface (JNI) desde el entorno de Java en un sistema de Contiki compilado. JNI está integrado en la máquina virtual de Java (JVM) y proporciona una manera de localizar e invocar a métodos nativos en una plataforma. Este código se ejecuta dentro de la forma JVM puede operar con las aplicaciones escritas en otros lenguajes de programación tales como C. Una razón utilizar JNI es volver a utilizar las bibliotecas y las API no implementadas en Java. Una biblioteca es cargado en una clase Java utilizando el método System.loadLibrary, y ciertos métodos de Java, nativos, se asignan a las funciones de la biblioteca. Como un simple ejemplo, podemos tener un método nativo “tick”en que residen la clase “Lib1”en el paquete “se.sics.cooja.corecomm”. El código Java asociado sería algo como esto:

```
package se.sics.cooja.corecomm;
...
public class Lib1 ... {
static {
System.loadLibrary("mySharedLibrary");
}
native void tick();
}
```

La correspondiente función de C en la biblioteca sería:

```
JNIEXPORT void JNICALL
Java_se_sics_cooja_corecomm_Lib1_tick(JNIEnv *env, jobject obj)
{
...
...
}
```

El sistema de Contiki consiste en todo el núcleo de Contiki, los procesos de usuario preseleccionados, y una serie de drivers especiales de simulación. Esto hace que sea posible desplegar y simular el mismo código, sin ningún tipo de modificaciones, minimizando la demora entre la simulación y el despliegue.

El simulador de Java tiene un control total sobre la memoria de los nodos simulados. Por lo tanto el simulador puede ver o cambiar las variables del proceso de Contiki en todo momento, permitiendo posibilidades de interacción muy dinámicas desde el simulador. Otra consecuencia interesante de usar JNI es la capacidad de depurar el código Contiki usando cualquier depurador regular, como gdb, uniendo a todo el simulador de Java y realizando breaks cuando se realiza la llamada a JNI. También se puede guardar estados completos de simulación y ser restaurados tiempo después, retrocediendo las simulaciones en el tiempo.

Los periféricos de hardware de los nodos simulados se llaman interfaces, y permiten que el simulador de Java detecte y active eventos tales como tráfico entrante de radio o un LED que se enciende. Las interfaces también representan propiedades de los nodos simulados tales como posiciones que el nodo actual no es consciente.

Todas las interacciones con simulaciones y nodos simulados se realizan a través de plugins. Un ejemplo de un plugin es un control simulado que permite a un usuario iniciar o pausar una simulación. Ambas interfaces y plugins se pueden agregar fácilmente al simulador, permitiendo a los usuarios agregar rápidamente una funcionalidad personalizada para simulaciones específicas.

Simulación en diferentes niveles

COOJA permite simulaciones simultáneas en tres niveles diferentes:

- Nivel de red (o aplicación)
- Nivel de sistema operativo
- Nivel de instrucción de código de máquina.

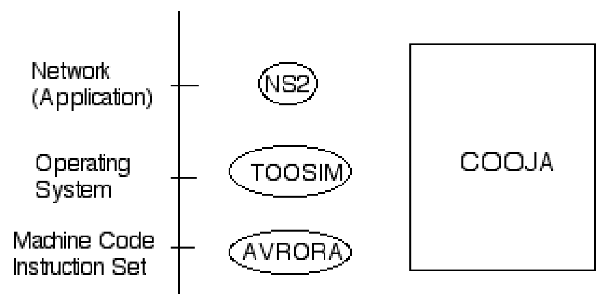


Figura IV.1: Niveles de simulación en COOJA

- **Nivel de Red**

Durante el diseño y aplicación de, por ejemplo, los protocolos de ruteo las especificaciones de hardware son a menudo no tan importantes como la propia red. El factor más importante puede afectar el medio de radio, los dispositivos de radio y tal vez el ciclo de trabajo en que los nodos sensores están dormidos. Al realizar una tarea de diseño e implementación puede ser posible, pero no necesario, utilizar una fina simulación de medio ambiente, como un simulador a nivel de instrucción.

COOJA soporta desarrollo de código que permite al usuario intercambiar fácilmente ciertos módulos de simulación como el dispositivo controlador o módulos del medio de radio. Una simulación puede ser guardada y cargada utilizando más o menos detalles de módulos, aún con los otros parámetros de simulación sin alteraciones. Además, los nuevos medios de radio y las interfaces, tales como los dispositivos de radio pueden ser desarrollados en Java y luego agregarlos al entorno de simulación de COOJA.

A fin de simplificar y acelerar el desarrollo de tales escenarios, COOJA también admite la adición de nodos de código Java. Sin ninguna conexión con Contiki, éstos pueden ser útiles en el desarrollo de algoritmos de alto nivel que al ser probados y evaluados serán portado para el despliegue de código de sensores. Nodos de código Java también se pueden utilizar en redes heterogéneas en las que el usuario sólo tiene que centrarse en un subconjunto de todos los nodos simulados. Dado que tales nodos de código Java requieren menos memoria y menos potencia de procesamiento, las redes heterogénea más largas pueden ser simuladas más eficientemente. Por ejemplo, utilizando nodos de código Java, los usuarios pueden implementar rápidamente las funcionalidades de varios nodos diferentes entre sí formando una red. Así entonces los usuarios pueden más tarde, nodo por nodo, portar el código Java a código de despliegue del nodo Contiki, manteniendo al mismo tiempo la funcionalidad completa de la red.

- **Nivel de Sistema Operativo**

COOJA simula el sistema operativo mediante la ejecución de código nativo del sistema como se describió anteriormente. Como todo el sistema operativo Contiki se ejecuta, incluidos los procesos de usuario, también es posible alterar la funcionalidad del núcleo de Contiki. Esto es útil para, por ejemplo, probar y evaluar los cambios en las librerías incluidas en Contiki.

- **Nivel de Instrucción de Código de Máquina conjunto**

Utilizando COOJA es posible crear nuevos nodos con muy diferentes estructuras de los nodos típicos. Como ejemplo puede ser la adición de nodos conectados a un emulador de un micro-controlador basado en Java en lugar de un sistema de Contiki compilado. El emulador representa un nodo de ESB (Embedded Sensor Board), emulando a nivel de bits.

Los nodos emulados son controlados en forma similar a los nodos de código nativo. A cada nodo

simulado se le permite correr una duración máxima de un período de tiempo fijo o el tiempo suficiente para manejar un evento. Los eventos son entonces, utilizando la memoria del nodo, transferidos a través de la interfaz de hardware y del simulador o viceversa.

- **Simulación en COOJA utilizando niveles cruzados**

Como se explicó antes COOJA soporta simulaciones en estos tres diferentes niveles de abstracción. Cada nodo individual siempre es simulado en uno de estos niveles. La principal ventaja de las simulaciones en COOJA utilizando niveles cruzados es que los nodos de cada uno de los niveles pueden coexistir e interactuar en la misma simulación. Así, por ejemplo, un nodo emulado puede enviar un paquete de radio a un nodo basado en Java.

Modelos de Radio

Cada simulación en COOJA utiliza un modelo de radio que caracteriza la propagación de ondas de radio. Nuevos modelos de radio pueden ser añadidos en el entorno de simulación. El modelo de radio es elegido cuando se crea una simulación. Esto permite a un usuario, por ejemplo, desarrollar un protocolo de red utilizando un simple modelo de radio, y luego probarlo mediante un modelo más realista, o incluso un modelo hecho a medida para poner a prueba el protocolo en condiciones muy especificas de red. A menudo, un modelo de radio proporciona uno o varios plugins con el fin de configurar y ver las condiciones actuales de la red simulada.

COOJA soporta un modelo simple que utiliza una interferencia y parámetros de rango de transmisión que se pueden cambiar durante la ejecución de una simulación, excepto en un modelo completamente en silencio.

Implementación

- **Compilación de Contiki en COOJA**

El sistema operativo Contiki está orientado a eventos. Cada caso manejado en el sistema se puede ejecutar hasta el final. Se trata de un enfoque común bien adaptado para dispositivos de escasa memoria, como nodos de sensores y también se utiliza en ,por ejemplo, en TinyOS . COOJA explota esta propiedad llamando al sistema Contiki cargado para que cada nodo simulado maneje solamente un evento a la vez. Todas estas llamadas comienzan en el loop de simulación, donde los “ticks” de simulación habilitan los nodos. Durante un tick de un nodo , tanto antes como después de que realmente se ejecute el código de Contiki, cada interfaz de nodo puede comprobar si tiene nuevos eventos. Por ejemplo, una interfaz de radio puede recibir nuevos datos de entrada del medio de radio, y se asegurará de que los nuevos datos son descubiertos por el software del nodo. Además de decidir qué nodos deben actuar, el loop de simulación también aumenta el tiempo global de simulación y avisa al simulador que otro loop se llevó a cabo.

Cada sistema Contiki es siempre compilado por una plataforma específica de hardware. La plataforma tiene disposición de los drivers y por lo tanto define la forma de comunicarse con el hardware. Cuando un nodo se emula, por ejemplo en el emulador MSP430, el sistema Contiki sigue siendo compilado por la arquitectura del procesador MSP430. Pero cuando un nodo debe ser simulado utilizando el enfoque de código nativo, el sistema Contiki es compilado para una plataforma especial de simulación. Esta plataforma de simulación ofrece una variedad de drivers, permitiendo que la mayoría de los procesos de Contiki se compilen de inmediato. La compilación y la carga de toda la biblioteca se controla desde el interior del simulador, y un usuario puede elegir qué procesos y dispositivos periféricos de hardware (interfaces) deben estar disponibles. El código fuente principal de Contiki entonces se genera, compila y carga en el simulador.

- **Simulación del Sistema Operativo**

Cuando se simulan nodos de código nativo, el tipo de nodo actúa como un enlace entre un nodo y el sistema Contiki compilado. El tipo de nodo carga la biblioteca compilada compartida y toda interacción entre el simulador y la biblioteca es a través de éste tipo. El tipo de nodo sólo tiene unas pocas funciones por las cuales interactúa con el sistema Contiki cargado. Estas incluyen funciones para copiar y reemplazar segmentos de memoria, la inicialización del sistema Contiki y finalmente una función que le dice al sistema que debe controlar un evento - que marque al nodo en el nivel del sistema Contiki.

Todos los nodos del mismo tipo comparten el mismo código de programa de memoria - el sistema operativo Contiki y un conjunto de programas de Contiki - mientras que la memoria de datos (el estado del programa) es diferente para cada nodo sensor simulado. Para cada nodo, COOJA guarda una copia de la memoria del entorno de C.

Cuando un nodo sensor está programado para ejecutarse, su memoria se copia en el entorno de C, y se llama a JNI a través del núcleo de eventos manejados en Contiki. El núcleo de Contiki despacha un único evento de su cola de eventos a uno de los procesos que se ejecutan en el sistema, tras el cual se devuelve el control de nuevo a COOJA. En este punto los datos de memoria potencial actualizados se descargan en una copia de seguridad en el entorno Java.

Para cada sistema Contiki cargado, COOJA debe ser capaz de encontrar las direcciones de las funciones y variables. Esto es realizado mediante el análisis del map-file generado en el tiempo de enlace. El map-file contiene, entre otras cosas, información sobre direcciones de símbolos en la biblioteca. Mediante la comparación de la dirección de la memoria absoluta de una variable en tiempo de ejecución, con su dirección relativa especifica en el map-file, COOJA es capaz de calcular las direcciones de todas las variables de la biblioteca.

El resto del análisis del map-file permite al simulador buscar y modificar los valores de variables durante las simulaciones. Esta es la principal forma en la que una interfaz o un plugin se comunica con el sistema Contiki; ve y altera los valores de determinadas variables. Por ejemplo, un plugin puede ser seteado para que al ver una variable específica a través de una simulación dispare una acción dependiente de ella.

La única memoria para segmentos que COOJA necesita copiar de la biblioteca, son los datos y los segmentos BSS (datos no inicializados). Debido a la limitada memoria de plataformas con que el sistema operativo Contiki está diseñado, un enfoque orientado a eventos es utilizado, donde cada evento se puede ejecutar hasta el final. Esto, entre otras ventajas, permite que los procesos compartan un solo stack. Las aplicaciones de Contiki correctamente escritas no deben por lo tanto nunca utilizar el stack como un punto de almacenamiento entre los eventos. Por lo tanto no hay necesidad para COOJA de guardar y restaurar la memoria del stack al simular diferentes nodos utilizando el mismo sistema Contiki. Y puesto que la memoria de texto de segmento es idéntica para todos los nodos del mismo tipo simulados, los únicos segmentos de memoria que COOJA debe copiar son los datos y los segmentos del BSS.

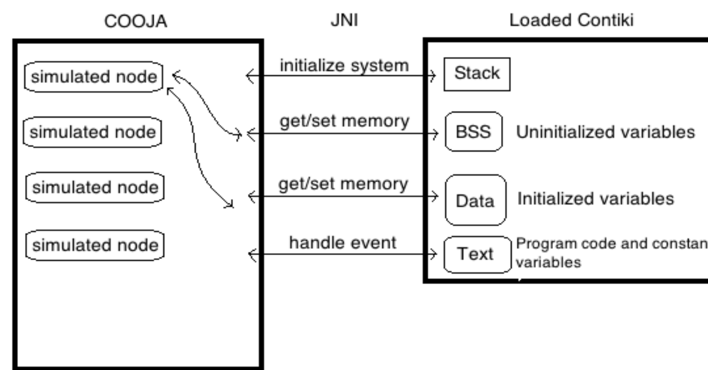


Figura IV.2:

Configuración del sistema COOJA

COOJA utiliza un sistema de configuración que permite a un usuario modificar las partes del entorno de simulación sin cambiar cualquier código principal de COOJA. El sistema puede ser utilizado tanto para agregar nuevos componentes como interfaces, plugins y medios de radio, o para reconfigurar componentes existentes. Los nuevos archivos son puestos en directorios del proyecto y se pueden activar y utilizar desde dentro del simulador.

- **Plugins e Interfaces**

La interacción típica con un nodo simulado es a través de una interfaz de nodo disponible. Cada interfaz representa alguna propiedad, por ejemplo, un radio o una posición. Como el nodo en sí mismo no sabe su posición simulada esta interfaz no tiene necesidad de comunicación con la memoria del nodo o el sistema subyacente Contiki. Cuando la posición de un nodo se altera, la interfaz señala un cambio para sus observadores. Ejemplos de esos observadores podrán ser el medio de radio actual y un plugin visualizador de nodo activo.

Las interfaces de nodos que necesitan comunicarse con el código Contiki suelen tener una parte correspondiente en el sistema de Contiki, no sólo en el entorno Java. La comunicación entre estas partes se realiza siempre por la manipulación de la memoria del nodo. Por ejemplo, una interfaz de botón que es presionada señala una bandera en la memoria del nodo. Cuando la memoria se ha copiado en Contiki una interfaz de botón correspondiente descubre que se ha presionado e informa a Contiki acerca de la misma manera que un hardware común realiza una interrupción. Al igual que una interfaz de posición, la interfaz de botón señala un cambio cuando el botón es presionado o soltado.

Un plugin se utiliza para interactuar con una simulación. A menudo provee al usuario con alguna interfaz gráfica. Los plugin pueden ser de pocos tipos diferentes, se refiere a una simulación, o un nodo, o a ninguno de ellos (llamado plugin GUI – Graphical User Interface). El nodo plugin por ejemplo puede ver algunas variables de contador de un nodo y pausa la simulación cuando se llega a 100. Una simulación plugin puede mostrar las posiciones de todos los nodos en una simulación. Y una interfaz gráfica de usuario plugin puede iniciar nuevas simulaciones, llevar a cabo algunas pruebas, registro de los resultados y repetir.

A lo largo de COOJA un enfoque observador observable es utilizado. Diferentes partes de COOJA puede tener observadores, por ejemplo el estado de simulación, el loop de simulación, el medio de radio y todas las interfaces del nodo. Esto permite interacciones muy dinámicas, por ejemplo el medio de radio simplemente observa todas las posiciones e interfaces de radio.

Anexo V: Ejemplo Mensajes de Control de RPL

Figura III.1: Disposición de los nodos para la prueba

Serial nodo 1:

Rime started with address 0.18.116.1.0.1.1.1
MAC 00:12:74:01:00:01:01:01 Contiki 2.5 started. Node id is set to 1.

CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26

RPL started

Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7401:0001:0101

Starting 'UDP server process' 'collect common process'

I am sink!

UDP server started

created a new RPL dag

Server IPv6 addresses: ::

Dirección IP global del nodo sink (nodo 1)

aaaa::1

Dirección link-local del nodo sink (especie de dirección MAC pero en IP)

fe80::212:7401:1:101

Created a server connection with remote address :: local/remote port 5688/8775

Se setea como root el nodo por lo tanto manda un multicast de DIO

RPL: Sending prefix info in DIO for aaaa::

RPL: Sending a multicast-DIO with rank 256

RPL: Neighbor fe80::212:7402:2:202 is known. ETX = 5

Received an RPL control message

El nodo 2 recibe el DIO del nodo 1 y lo agrega como parent por dicho motive primero enviará un multicast DIO para diseminar su nueva estructura dag y luego enviara un mensaje DAO pero solo a el parent (nodo 1)

RPL: Received a DIO from fe80::212:7402:2:202

RPL: Neighbor added to neighbor cache fe80::212:7402:2:202, 00:12:74:02:00:02:02:02

RPL: Incoming DIO rank 1024

RPL: DIO Conf:dbl=8, min=12 red=10 maxinc=768 mininc=256 ocp=1 d_l=255 l_u=65535

RPL: Copying prefix information

Received an RPL control message

Este es el mensaje DAO que en el comentario anterior se adelanto que sería enviado por el nodo 2. Al setear al nodo 1 como preferred parent le envía su dirección global y le dice que para llegar a ella tiene que enviar a

la dirección “mac” de él mismo.

RPL: Received a DAO from fe80::212:7402:2:202

RPL: DAO lifetime: 4294967295, prefix length: 128 prefix: aaaa::212:7402:2:202

Línea agregada en la tabla de ruteo del nodo 1

RPL: Added a route to aaaa::212:7402:2:202/128 via fe80::212:7402:2:202

Received an RPL control message

Se recibe un segundo mensaje DAO del nodo 2. Pero este es un reenvió de un mensaje DAO que fue enviado del nodo 3 al 2. Con este mensaje el nodo 1 sabrá que para ir al nodo 3 el next-hop es la dirección mac del nodo 2.

RPL: Received a DAO from fe80::212:7402:2:202

RPL: DAO lifetime: 4294967295, prefix length: 128 prefix: aaaa::212:7403:3:303

Línea agregada en la tabla de ruteo del nodo 1

RPL: Added a route to aaaa::212:7403:3:303/128 via fe80::212:7402:2:202

RPL: Sending prefix info in DIO for aaaa::

RPL: Sending a multicast-DIO with rank 256

Serial nodo 2

Rime started with address 0.18.116.2.0.2.2.2

MAC 00:12:74:02:00:02:02:02 Contiki 2.5 started. Node id is set to 2.

CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26

RPL started

Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7402:0002:0202

Starting 'UDP client process' 'collect common process'

UDP client process started

Dirección IP global del nodo 2

Client IPv6 addresses: aaaa::212:7402:2:202

Dirección link-local del nodo 2 (especie de dirección MAC pero en IP)

fe80::212:7402:2:202

Created a connection with the server :: local/remote port 8775/8775

RPL: Neighbor fe80::212:7401:1:101 is known. ETX = 5

Received an RPL control message

Se recibe el mensaje DIO del root que envío por setearse como root

RPL: Received a DIO from fe80::212:7401:1:101

RPL: Neighbor added to neighbor cache fe80::212:7401:1:101, 00:12:74:01:00:01:01:01

RPL: Incoming DIO rank 256

RPL: DIO Conf:dbl=8, min=12 red=10 maxinc=768 mininc=256 ocp=1 d_l=255 l_u=65535

RPL: Copying prefix information

RPL: Neighbor fe80::212:7403:3:303 is known. ETX = 5

Received an RPL control message

Se recibe mensaje DIS del nodo 3, este es enviado porque no tiene ningún dag asociado y se le venció el timeout para disparar el DIS.

RPL: Received a DIS from fe80::212:7403:3:303

Se setea el timer DIO para enviar un mensaje de este tipo inmediatamente debido a la recepción del mensaje DIS.

RPL: Multicast DIS => reset DIO timer

RPL: Sending prefix info in DIO for aaaa::

RPL: Sending a multicast-DIO with rank 1024

Received an RPL control message

RPL: Received a DIO from fe80::212:7403:3:303

RPL: Neighbor added to neighbor cache fe80::212:7403:3:303, 00:12:74:03:00:03:03:03

RPL: Incoming DIO rank 1792

RPL: DIO Conf:dbl=8, min=12 red=10 maxinc=768 mininc=256 ocp=1 d_l=255 l_u=65535

RPL: Copying prefix information

Se envía un mensaje DAO al nodo 1 porque fue seteado como preferred parent, se le enviara información de cómo llegar al nodo 2. Esto será agregado en la tabla de ruteo del nodo 1.

RPL: Sending DAO with prefix aaaa::212:7402:2:202 to fe80::212:7401:1:101

RPL: Neighbor fe80::212:7401:1:101 is known. ETX = 4

Received an RPL control message

Debido al mensaje multicast DIO que envío el nodo 2, el nodo 3 lo agregó como preferred parent y entonces el nodo 3 le envía un mensaje DAO al nodo 2.

RPL: Received a DAO from fe80::212:7403:3:303

RPL: DAO lifetime: 4294967295, prefix length: 128 prefix: aaaa::212:7403:3:303

Línea agregada a la tabla de ruteo del nodo 2

RPL: Added a route to aaaa::212:7403:3:303/128 via fe80::212:7403:3:303

Los mensajes DAO se forwardean a el preferred parent. De esta manera el nodo 1 tendrá una entrada en la tabla de ruteo para saber como llegar al nodo 3.

RPL: Forwarding DAO to parent fe80::212:7401:1:101

Serial nodo 3

Rime started with address 0.18.116.3.0.3.3.3

MAC 00:12:74:03:00:03:03:03 Contiki 2.5 started. Node id is set to 3.

CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26

RPL started

Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7403:0003:0303

Starting 'UDP client process' 'collect common process'

UDP client process started

Dirección IP global del nodo 3

Client IPv6 addresses: aaaa::212:7403:3:303

Dirección link-local del nodo 3 (especie de dirección MAC pero en IP)

fe80::212:7403:3:303

Created a connection with the server :: local/remote port 8775/8775

RPL: Neighbor fe80::212:7404:4:404 is known. ETX = 5

Received an RPL control message

Recibe un DIS del nodo 4. Porque este no tiene asignado un dag y venció su timeout. De todos modos este mensaje no será procesado porque en nodo 3 tampoco tiene asignado un dag.

RPL: Received a DIS from fe80::212:7404:4:404

Envía un DIS. Porque no tiene asignado un dag y venció su timeout.

RPL: Sending a DIS

RPL: Neighbor fe80::212:7402:2:202 is known. ETX = 5

Recibe un mensaje DIO del nodo 2 a pedido del mensaje DIS que fue enviado.

Received an RPL control message

RPL: Received a DIO from fe80::212:7402:2:202

RPL: Neighbor added to neighbor cache fe80::212:7402:2:202, 00:12:74:02:00:02:02:02

RPL: Incoming DIO rank 1024

RPL: DIO Conf:dbl=8, min=12 red=10 maxinc=768 mininc=256 ocp=1 d_l=255 l_u=65535

RPL: Copying prefix information

Se envía un mensaje DIO para propagar el nuevo dag asociado (dag joined)

RPL: Sending prefix info in DIO for aaaa::

RPL: Sending a multicast-DIO with rank 1792

A partir del mensaje DIO recibido se setea al nodo 1 como preferred parent por lo tanto se le enviara un mensaje DAO

RPL: Sending DAO with prefix aaaa::212:7403:3:303 to fe80::212:7402:2:202

RPL: Neighbor fe80::212:7402:2:202 is known. ETX = 4

Received an RPL control message

Anexo VI: Protocolo Deluge

Deluge fue creado para favorecer la programación de redes. Se trata de un protocolo de difusión de datos, confiable para la propagación multihop de archivos de gran tamaño desde uno o más nodos de origen hacia muchos nodos destino en una red de sensores inalámbricos. Fue pensado para diseminar imágenes binarias en una red.

El protocolo fue diseñado teniendo en cuenta cinco problemas. Primero, que las imágenes que se quiere enviar son mucho más grandes que los datos para los cuales se diseñaron protocolos de diseminación anteriores. Segundo, la diseminación debe funcionar sin importar la densidad de nodos de que se trate. En tercer lugar, es necesaria absoluta confiabilidad debido a que se debe garantizar que cada byte llegue correctamente para que sea posible programar cada nodo de la red, aún en condiciones de alta tasa de pérdidas y malas calidades de enlace. En cuarto lugar, se debe asegurar que todos los nodos de la red tengan actualizada su versión de código, aún en casos en que la cantidad de nodos varíe debido a fallas temporales y agregado de nodos. Finalmente la diseminación debe insumir la menor cantidad de tiempo posible de manera de minimizar las interrupciones de servicio que impactan en el despliegue y en el ciclo de depuración y testing de la aplicación.

Deluge se inspira en Trickle pero lo modifica para que ser capaz de transferir un mayor volumen de información. Utiliza Trickle para controlar la transmisión de mensajes potencialmente redundantes. En Deluge un nodo solicita un dato al último nodo que lo anunció, también permite el anuncio de páginas de código aún antes de haberlas recibido por completo permitiendo la diseminación de páginas recién recibidas. A esta técnica le llama Multiplexado Espacial y permite que las páginas de código sean transferidas a través de la red. Esta técnica reduce drásticamente el tiempo necesario para atravesar la red.

Este incremento de velocidad se paga con mayor consumo ya que es necesario que la red debe mantenerse encendida de manera de que se pueda lograr el mayor beneficio del Multiplexado Espacial. Es posible aplicar esta técnica por zonas de la red de manera de resolver el compromiso entre consumo y velocidad de transferencia.

Con la intención de lograr aprovechar totalmente el beneficio del Multiplexado Espacial Deluge cuida que la transferencia de diferentes páginas no interfieran entre si. Para esto limita a los nodos, exigiendo que la solicitud de datos sea en orden secuencial. Esto le permite a los nodos vecinos aprovechar el medio compartido de manera de que dos nodos reciban la misma página en lugar de competir para recibir páginas diferentes.

En Contiki Deluge está implementado encima del stack Rime. Desde el punto de vista del software esta pensado como una aplicación (/apps/deluge/) encima de Contki y está compuesto por dos archivos deluge.c(.h). La lógica principal es un protothread:

```
PROCESS_THREAD(deluge_process, ev, data);
```

que utiliza varias funciones inaccesibles para el usuario. La interfaz pública proporciona las estructuras de datos para manejar la lógica y una única función,

```
int deluge_disseminate(char *file, unsigned version);
```

Para programar una versión de Deluge que funcione encima del stack IPv6 de Contiki. Es necesario volver a escribir las funciones que implementan la interfaz con Rime, para la gestión de las conexiones unicast y broadcast.

Anexo VII: Rime

El stack de Rime tiene una arquitectura de capas. Los protocolos más complejos están implementados usando los menos complejos.

abc (Anonymous Best-effort Single-hop Broadcast)

Es la primitiva de comunicación más básica. Permite a las capas superiores enviar paquetes a todos los vecinos que escuchan el canal en el que se envió el paquete. No se envía información de quien envió el paquete. El resto de las primitivas están basadas en esta.

ibc (Identified Best-effort Single-hop Broadcast)

Utiliza la primitiva abc y agrega la dirección del nodo remitente.

uc (Best-effort Single-hop Unicast)

Envía un paquete a un nodo vecino determinado. Usa la primitiva ibc y agrega la dirección del destinatario. Cuando un nodo recibe un paquete chequea la dirección del destinatario, si esta no coincide con la propia, descarta el paquete.

stuc (Stubborn Single-hop Unicast)

Envía repetidamente un paquete usando uc hasta que la capa superior cancela la transmisión.

ruc (Reliable Single-hop Unicast)

Utiliza acks y retransmisiones para asegurar que el nodo vecino recibió exitosamente un paquete. Cuando el destinatario responde el ack, el modulo ruc notifica a la capa superior. Utiliza la primitiva stuc para realizar retransmisiones y números de secuencia para confirmar los acks de los paquetes enviado.

polite (Polite Single-hop Broadcast)

Está diseñado para reducir la cantidad de paquetes enviados no repitiendo un mensaje que otro nodo ya envió. La capa superior que usa este protocolo setea un intervalo de tiempo en el cual se debe evitar el paquete. Si cuando este intervalo expira no se recibió el paquete que se está por enviar, entonces es enviado.

Ipolite (Identified Polite Single-hop Broadcast)

Utiliza la primitiva polite y agrega el identificador de remitente.

mh (Best-effort Multi-hop Unicast)

Envía un paquete a un nodo especifico de la red usando reenvió de multi-hop en cada nodo de la red. La capa superior (o aplicación) que usa mh pone el algoritmo de ruteo para seleccionar el próximo el próximo paso. Si a la primitiva se le solicita enviar un paquete para el cual no se encuentra ningún nodo vecino adecuado, se le notifica a la capa superior y esta pude iniciar el proceso de descubrimiento de ruta. Cuando el next-hop neighbor es encontrado la primitiva mh manda el paquete usando uc.

rmh (Hop-by-hop Reliable Multi-hop Unicast)

Es similar a mh salvo que usa la primitive ruc en vez de uc para comunicarse confiablemente con los nodos vecinos.

nf (Best-effort Network Flooding)

Envía un paquete a todos los nodos de la red. Usa un polite broadcast para cada hop para reducir el número de transmisiones redundantes. Usa el atributo "time to live" para eliminar paquetes.

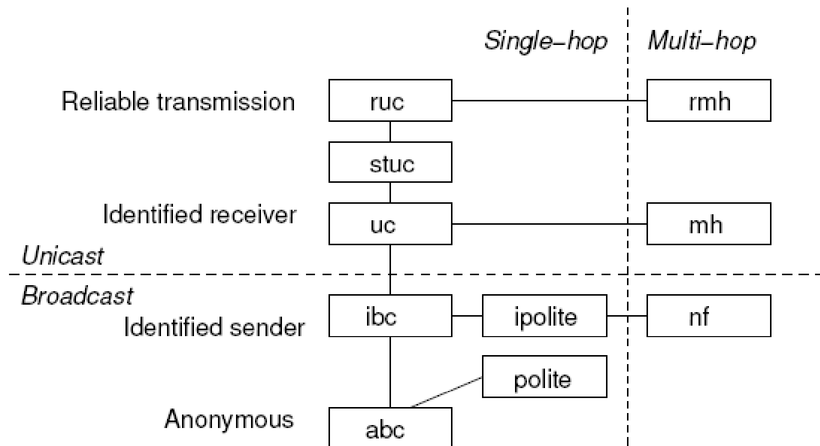


Figura VII.1: Stack Rime

Anexo VIII: Gestión

Resumen

- **Estudiantes**
  - **Gabriel Firme**
    CI: 1.737.538-1
    e-mail: gfirme@adinet.com.uy
  - **Ignacio de Mula**
    CI: 3.208.803-0
    e-mail: nachodm@gmail.com
  - **Germán Ferrari**
    CI: 4.398.201-1
    e-mail: gferrari86@gmail.com
- **Cliente**

  Instituto de Ingeniería Eléctrica de la UdelaR
- **Tutor**

  Leonardo Steinfeld

Plazos

La fecha de inicio del proyecto es el 16/08/2010 y la fecha prevista para su finalización es el 30/09/2011. La carga horaria media prevista por estudiante es de 10 horas semanales. Por lo tanto el total de horas previstas para dedicar en el proyecto es de 1770hs.

Entregables

1. **15 febrero 2011 – Presentación de avance 1**

Para este entregable se presentará Contiki OS y los resultados obtenidos al aplicar este OS a una red de sensores inalámbricos.

2. **15 junio 2011 – Presentación de avance 2**

Para este entregable se presentará una aplicación desarrollada por nosotros que corre sobre Contiki OS para el monitoreo de fenómenos climáticos en una red de sensores inalámbricos.

3. **30 setiembre 2011 – Presentación Final del Proyecto**

Se presentarán todos los puntos anteriores y además se habrá estudiado y optimizado el consumo del Contiki OS para rede de sensores inalámbricos.

Descripción del Proyecto

Introducción

Históricamente en las redes de sensores inalámbricas es muy importante minimizar el consumo de los nodos de forma de aumentar la autonomía energética de modo de que no sea necesario el cambio de la batería de cada nodo, ya que es una tarea tediosa en redes de gran porte.
En este proyecto se utilizará Contiki, un OS que está diseñado para redes de sensores inalámbricos los cuales son sistemas con recursos limitados de procesamiento, energía y memoria.

Contiki propone utilizar el protocolo TCP/IP para la comunicación entre los nodos. La introducción de éste stack de protocolos si bien brinda grandes posibilidades de interconexión presenta un desafío debido a que no fue concebido inicialmente para este tipo de redes. Contiki ofrece una versión liviana que se adapta a las WSN sin perder las funcionalidades de interconexión.

Se propone estudiar las diferentes alternativas para transmitir datos a través de la red. Se estudiará el sincronismo, la arquitectura de la red y la jerarquización de los nodos evaluando distintas formas de trasmitir datos a través de red y el impacto en el consumo de las distintas estrategias. Para bajar el consumo se trata de que los nodos estén la mayor parte del tiempo en modo de bajo consumo. Por esto es importante mantener la sincronía para que se despierten simultáneamente de forma de minimizar el tiempo que dura la comunicación. El Contiki ya tiene implementada una solución para el sincronismo entre nodos la que podremos estudiar y decidir si utilizar o no.

Se dispone de simuladores para redes de sensores inalámbricos MSPSim y COOJA. Estos parecen muy potentes ya que es posible indicar el código que debe ser ejecutado por cada nodo y simular cómo se comporta la red pudiendo observar el código que es ejecutado en cada momento por cada nodo y también pudiendo estimar el consumo de cada nodo.

Contiki ya viene portado para varios tipos de plataformas en particular porta la que nosotros pensamos usar (TmoteSky).

Se dispone documentación del código y artículos académicos escritos por el grupo de Contiki. Nosotros proponemos crear una referencia la que introduzca los conceptos fundamentales para trabajar con el OS Contiki.

Antecedentes

En el IIE de la UdelaR existen varios antecedentes de proyectos de fin de carrera de redes de sensores inalámbricos en los que se utilizan TinyOS. Este sistema operativo utiliza el lenguaje nesC y es orientado a eventos.

Objetivo

Objetivo General

Implementar una aplicación en una red de sensores inalámbricos utilizando Contiki buscando optimizar el consumo y mejorar la eficiencia de la comunicación entre nodos de la red.

Finalidades

- Desarrollar una red de sensores inalámbricos de mayor autonomía, gestionable remotamente y que se pueda conectar a una red convencional.

- Profundizar el estudio del Sistema Operativo Contiki para futuros trabajos.
- Emprender nuestro primer proyecto de ingeniería a modo de desarrollar una tarea de síntesis de los conocimientos adquiridos y realizar experiencias de integración en una estructura de trabajo en grupo.

Criterios de éxito

- Mejorar el consumo de la red con respecto a los proyectos anteriores realizando las mismas tareas así como obtener una tabla comparativa de las distintas alternativas propuestas de manera de poder extraer conclusiones que permitan tomar decisiones de diseño.

- Entender en profundidad y poder implementar una aplicación en una red de sensores inalámbricos utilizando Contiki y crear un documento que sirva como referencia del OS y como referencia de aprendizaje del mismo que será validada tanto por el tutor como luego por el tribunal.

- Para alcanzar los objetivos se podrán implementar estrategias tanto a nivel de la capa de enlace como de la de red, o una combinación de ambas.

- Aprobación del proyecto por parte del tribunal.

Actores

Estudiantes de grado

- Germán Ferrari

- Ignacio de Mula

- Gabriel Firme

Tutor

- Leonardo Steinfeld

Creadores del Contiki

- Se trata de un RTOS OpenSource y por lo tanto disponemos de los fuentes, de artículos académicos y de una lista de mail para consultas.

Docentes del IIE

- Integrantes de grupo de microcontroladores que pueden estar interesados en el proyecto. En especial Fernando Silveira y Pablo Mazzara.

Supuestos

Sobre los estudiantes

- Los estudiantes trabajaran de forma activa y constante en el proyecto hasta su finalización.
- Los estudiantes dedicarán, en promedio, un mínimo de 10 horas por semana al proyecto.
- El tutor orientará el trabajo y mantendrá reuniones regulares con los estudiantes.

Sobre el tiempo

- No se modificaran los plazos establecidos para la entrega del proyecto.

Sobre los recursos

- Los nodos para implementar la red estarán disponibles cuando se necesiten.
- El código fuente del Contiki es claro y esta comentado.
- El Contiki entrará en el Tmote. En caso de no ser así se dispondrá de una plataforma con memoria suficiente.

Restricciones

- Se cuenta con un presupuesto limitado.
- En caso de necesitar la compra de nodos para prueba de campo la misma deberá ser aprobada por el Instituto.
- El tiempo disponible para realizar el proyecto es de 2 semestres.
- El lenguaje de programación debe ser C.

Especificación Funcional del Proyecto

El proyecto tendrá como funcionalidad entender en profundidad y poder implementar una aplicación en una red de sensores inalámbricos utilizando el Sistema Operativo Contiki.
También se creará un documento que sirva como referencia del OS y como referencia de aprendizaje del mismo para futuros posibles proyectos en redes de sensores inalámbricas.

Alcance

Alcance del Proyecto

- La red se implementará con los nodos disponibles, entre 6 y 10, pero será escalable y se harán simulaciones con una mayor cantidad de nodos.
- No se diseñará hardware ni modificará el disponible.

- Tampoco es de interés, en un principio, crear una interfaz gráfica a efectos de desplegar los resultados obtenidos durante las pruebas de campo o simulaciones.

Objetivos Específicos

1. Aprender a utilizar y comprender el funcionamiento del Sistema Operativo Contiki para microcontroladores.

2. Desarrollar una aplicación que corra sobre Contiki OS para el monitoreo de fenómenos climáticos en una red de sensores inalámbricos.

3. Optimizar el consumo del Sistema Operativo Contiki.

Entregables Asociados

- **Entregables del Objetivo Especifico 1**
  - Tutorial de Contiki
  - Poner en funcionamiento el Contiki en una redes de sensores Inalámbricos
- **Entregables del Objetivo Especifico 2**
  - Aplicación Desarrollada y en funcionamiento
  - Documentación de la Aplicación desarrollada
- **Entregables del Objetivo Especifico 3**
  - Consumo mejorado con respecto a años anteriores
  - Consumo mejorado con respecto a versión original de Contiki

Work Breakdown Structure

Introducción

Contiki Aprendido

En esta etapa se busca la comprensión completa del SO, tanto desde el punto de vista de su estructura, de las funcionalidades que ofrece. Para lograr esto identificamos tareas y sub-tareas que necesariamente deben realizarse para completar el objetivo.

- Estudio
  a. Documentación Leída
  b. Código Leído
  c. Ambiente de Desarrollo

- Pruebas y Evaluación
    - Simulaciones
        - Evaluar Simulador
            - Debugging
            - Consumo
            - Performance
        - Simular Ejemplos
    - Protocolos
            - IPv4
            - IPv6
            - RPL
    - Evaluación
            - Tamaño
            - Funcionalidad
            - Portabilidad
            - Consumo Básico
- Tarea Documentada

Aplicación Desarrollada

A partir de lo estudiado hasta este momento buscamos concebir, desarrollar e implementar la aplicación. Identificamos las siguientes tareas y sub-tareas para conseguir este objetivo específico.

- Requerimientos Definidos
- Aplicación Diseñada
- Aplicación Implementada
- Aplicación Simulada
- Aplicación Validada
- Tarea Documentada

Consumo Optimizado

Este objetivo consiste en evaluar las alternativas para optimizar el consumo, verificarlo y comparar con otras implementaciones.

- Consumo evaluado en todas las capas
- Consumo Medido y Comparado
  - Planificar pruebas de campo
  - Realizar pruebas de campo
  - Comparar los resultados
- Tarea Documentada

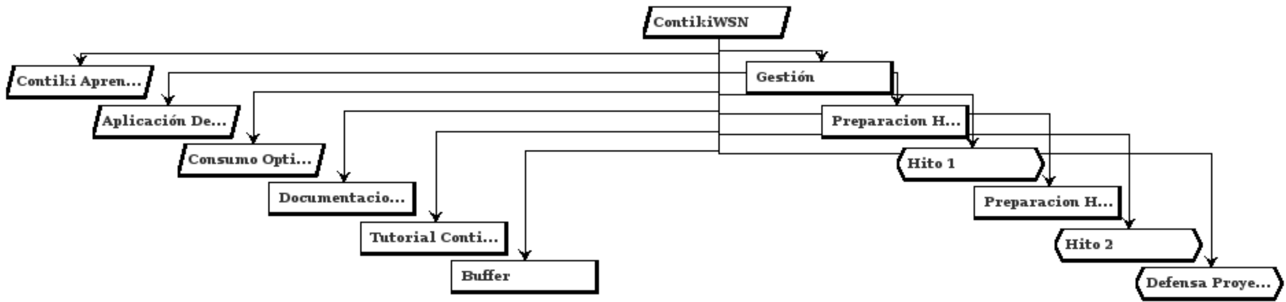
Documentación final y Gestión

A partir la documentación de las distintas etapas del proyecto se realizará la documentación final del proyecto. Se adjuntan también las tareas asociadas a la gestión del proyecto

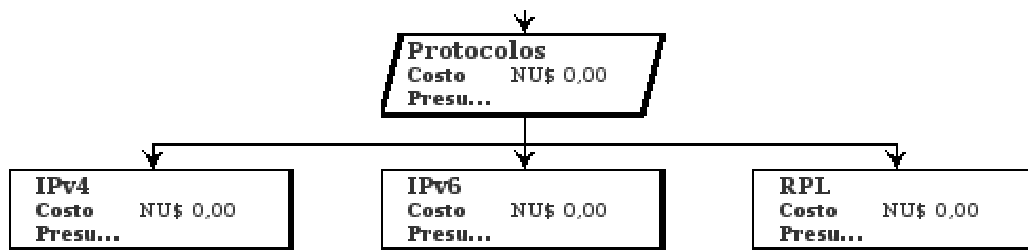
- Documentación Final
- Tutorial de Contiki Realizados
- Buffer
- Gestión
- Preparación Hito 1
- Hito 1
- Preparación Hito 2
- Hito 2
- Defensa del Proyecto

WBS en OpenProject

Presentamos la WBS en varios diagramas para facilitar la legibilidad.

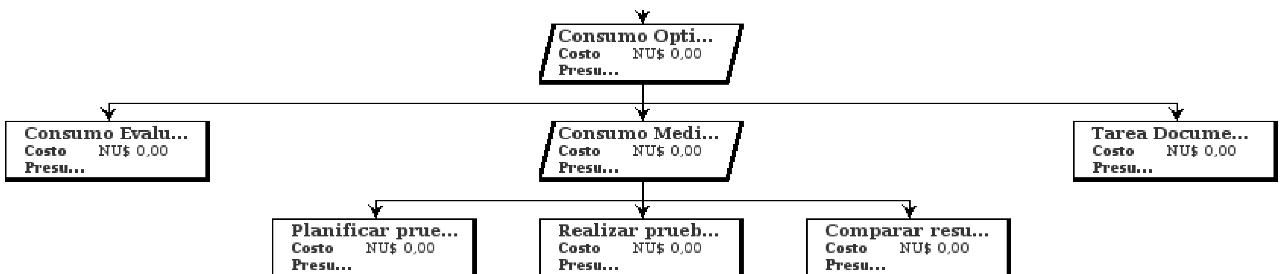


Contiki Aprendido



Aplicación Desarrollada

Consumo Optimizado



Análisis de Riesgos

Estimación y evaluación inicial de Riesgos

Nº	Riesgo	Prob. ocurrencia	Impacto
1	Tutor Renuncia	Poco Probable	Extremo
2	Estudiante Abandona	Poco Probable	Extremo
3	Estudiante le surge un percance	Moderado	Medio
4	Varios Motes se rompen	Poco Probable	Alto
5	Los Motes no están disponibles	Poco Probable	Medio
6	Se pierde la información almacenada digitalmente	Poco Probable	Extremo
7	Atraso debido a dificultades con las tareas previstas	Moderado	Alto
8	No es posible optimizar las capas dentro de Contiki	Moderado	Medio
9	No es posible realizar pruebas de campo	Moderado	Alto
10	El código fuente de Contiki no es claro	Moderado	Medio

		Probabilidad de Ocurrencia		
		Poco Prob	Moderado	Muy Prob
Nivel de Impacto	Ninguno			
	Bajo			
	Medio	5,9	3,8,10	
	Alto	4	7	
	Extremo	1,2,6		

Plan de Respuesta

1. **Tutor Renuncia**

   Se contacta a otros tutores del grupo de microelectrónica.

2. **Estudiante Abandona**

   Re planificar tiempos, dedicar mayor carga horaria y eventualmente pedir prorroga.

3. **A un estudiante le surge un percance**

   Repartir tareas del estudiante, dedicar mayor carga horaria y pedir prórroga en caso necesario.

4. **Varios Motes se Rompen**

   Comprar Motes y mientras utilizar el simulador COOJA.

5. **Los Motes no están disponibles**

   Evaluar disponibilidad, en caso necesario comprar motes de lo contrario esperar a que estén disponibles. Simular en COOJA hasta que se consigan.

6. **Se pierde la información almacenada electrónicamente**

   Hacer respaldos periódicos y nunca tener una sola copia de archivos vitales.

7. **Atraso debido a dificultades con las tareas previstas**

Re planificar tiempos, dedicar mayor carga horaria y eventualmente pedir prorroga.
Se asume y se pondrá énfasis en el desarrollo las comunicaciones de red y en nuestra aplicación.

8. **No es posible realizar pruebas de campo**

Se intentará enriquecer las simulaciones, con mayor variedad de escenarios y topologías.

9. **El código fuente de Contiki no es claro**

Se re planifica los tiempos de manera de dedicarle mayor tiempo al estudio del código.

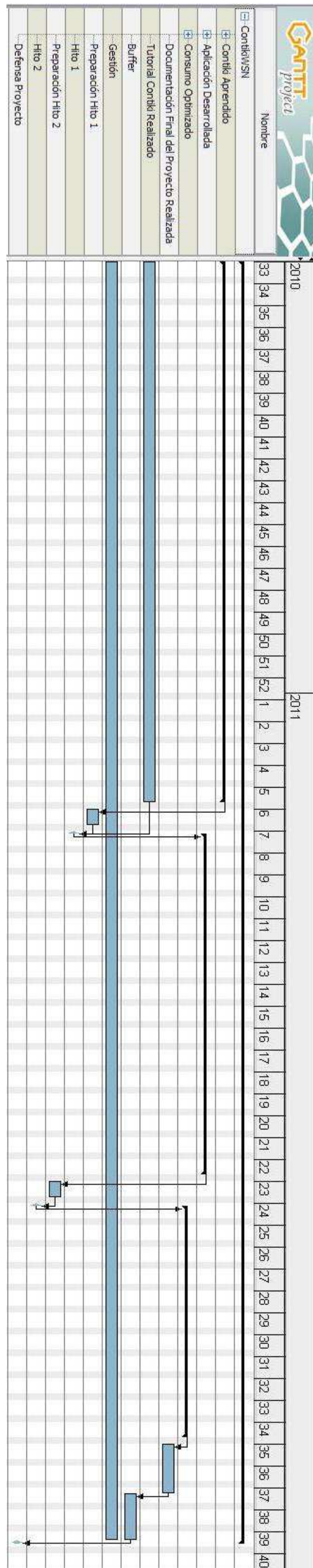
Estimación y Evaluación Final de Riesgos

		Probabilidad de Ocurrencia		
		Poco Prob	Moderado	Muy Prob
Nivel de Impacto	Ninguno			
	Bajo	9	3,8,10	
	Medio	5,4	7	
	Alto	1,2		
	Extremo			

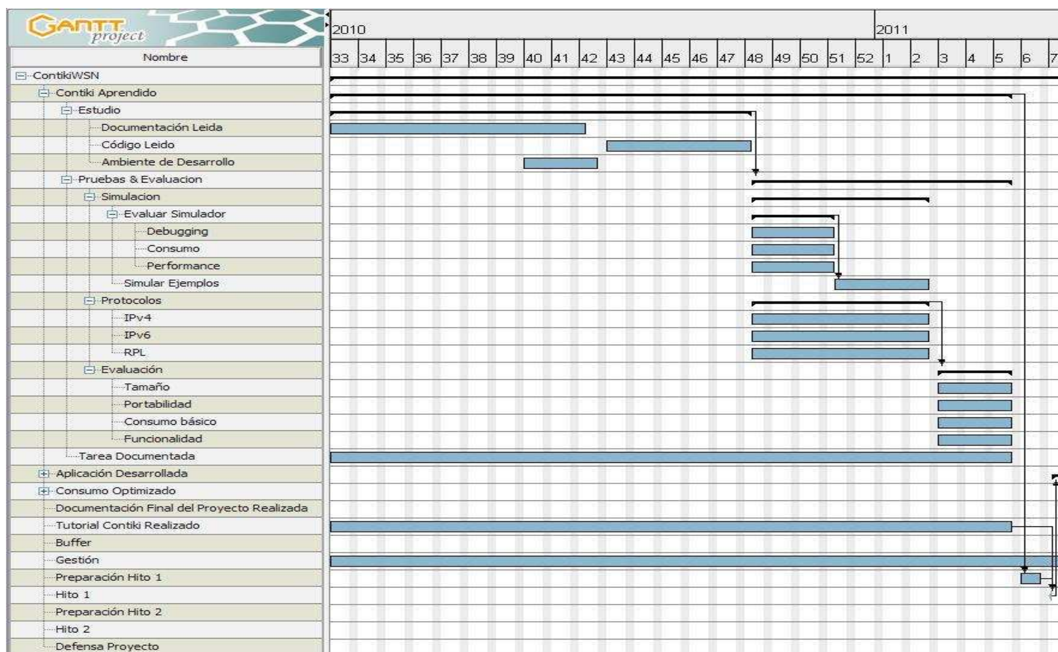
Cronograma detallado del Proyecto

Gantt

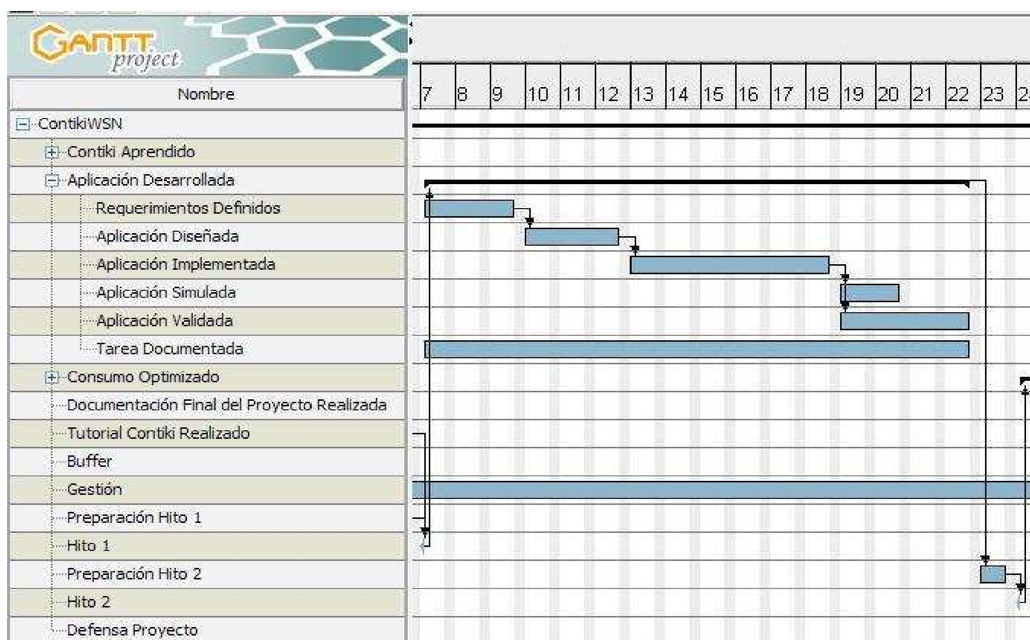
ContikiWSN



Contiki Aprendido



Aplicación Desarrolladla



Consumo Optimizado