Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de un entorno de pruebas de autenticación y seguridad de redes de sensores con IPv6

Autor: José Antonio Caballero Martos

Tutor: Juan Antonio Ternero Muñoz

Dep. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de un entorno de pruebas de autenticación y seguridad de redes de sensores con IPv6

Autor:
José Antonio Caballero Martos

Tutor:
Juan Antonio Ternero Muñiz
Profesor Colaborador

Dep. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Desarrollo de un entorno de pruebas de autenticación y seguridad de redes de sensores con IPv6

Autor: José Antonio Caballero Martos

Tutor: Juan Antonio Ternero Muñiz

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi abuelo Pepe, que no podrá ver cómo acabo mis estudios, como él hubiera querido. Pero estoy seguro de que estaría muy orgulloso.

Agradecimientos

Este trabajo supone el final de mi etapa como estudiante de Grado en Ingeniería de las Tecnologías de Telecomunicación. Han sido unos años muy duros, pero también muy buenos. Y siento una gran satisfacción ahora que estoy a punto de terminar.

En primer lugar, quiero agradecerle a todo el profesorado de la escuela su dedicación. Pues me han ayudado a formarme en el ámbito de la ingeniería, y también como persona. Estoy orgulloso de haber tenido la oportunidad de aprender de ellos. Quisiera agradecer en especial a Juan Antonio Ternero Muñiz, tutor de este trabajo, por haberme ayudado y guiado en su realización.

También me gustaría dar las gracias a todos los miembros de mi familia, los que están y los que una vez estuvieron. Sobre todo a mis padres, pues sin su sacrificio no habría podido nunca llegar a escribir estas líneas. Estoy orgulloso de tener a la madre más brillante y trabajadora de este mundo. Sin los valores de esfuerzo, constancia y buen hacer de las cosas que me ha inculcado no sería quien soy hoy. Y dudo que hubiera podido llegar hasta aquí. Y gracias también a mi padre, pues sin su pasión por las nuevas tecnologías, y ese Pentium II que me regaló en su día. No sé si habría llegado a despertar en mí el interés por estudiar esta carrera sin él. No quiero olvidarme tampoco de mi hermana, que aunque siempre sepa como darme la lata y no dejarme estudiar, me quiere y aprecia muchísimo, igual que yo a ella. Y a mi tata, que todos los días preguntó por el estado de este trabajo y tuvo que escuchar cada batallita librada durante el curso.

Por último, quiero agradecer al destino por haberme cruzado con los "Telecolegas", y a ellos por haber estado disponibles y dispuestos siempre que los he necesitado. Compañeros de clase que tras grandes trabajos en equipo, prácticas y tardes de estudio, pasaron a ser grandes y buenos amigos. Me han ayudado y enseñado mucho, y por eso les estaré eternamente agradecido.

José Antonio Caballero Martos

Sevilla, 2016

Resumen

Las redes inalámbricas de sensores (WSN) han tenido un creciente impacto en el mundo de las telecomunicaciones en los últimos años. Esto se debe en parte al auge del Internet de las cosas (IoT). Estas redes permiten que dispositivos de bajo consumo puedan comunicarse entre sí formando una malla, o con otros equipos en Internet a través de una pasarela. El pequeño tamaño y la autonomía de los dispositivos que las forman permiten su despliegue en lugares antes insospechados. Y su uso en aplicaciones antes impensables.

En este tipo de redes, la seguridad muchas veces puede ser un punto crítico. Pues las aplicaciones que se nutren de la información recogida por los sensores podrían verse gravemente afectadas si éstos fallan o si los datos son interceptados o modificados.

Además, el aumento de dispositivos conectados a Internet ha traído consigo un gran aumento en la demanda de direcciones en Internet. Y es aquí donde entra en escena el protocolo IPv6. Este protocolo nos proporciona un espacio de direcciones mucho mayor que el protocolo IPv4 que es el que se viene usando hasta ahora en Internet, además de otros cambios.

Es por eso que este trabajo tiene como finalidad el estudio de distintos mecanismos de autenticación y seguridad en redes inalámbricas de sensores que funcionen con IPv6. Y así poder hacer pruebas para asegurar las que se presupone que serán las redes que predominarán en un futuro cercano en el mundo de las telecomunicaciones.

Abstract

Wireless sensor networks (WSN) had a growing impact on the telecommunications world in recent years. This is due in part to the rise of the Internet of Things (IoT). These networks allow low-power devices to communicate with each other forming a mesh, or with other devices on the Internet through a gateway. Their small size and their autonomy let this devices be deployed in unexpected places. And used in applications that we could not imagine in the past.

In this type of networks, security is a very important matter most of the times. Because the applications that collect the sensor data could be severely affected if they crash, or if the data is intercepted or modified.

Furthermore, the increase in Internet-connected devices has brought a large increase in the Internet adresses demand. And here it is where IPv6 protocol is vital. This protocol provides a larger adress space than IPv4, which at the moment is the most used on the Internet. Besides IPv6 comes with other changes.

That is why this work is oriented to study different authentication and security mechanisms in wireless sensor networks that work with IPv6. So that we can make a test environment to ensure the networks that we think will proliferate in the next years.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Este Trabajo Fin de Grado consiste en el desarrollo de un escenario de pruebas de mecanismos de autenticación y seguridad para redes WSN con IPv6. A día de hoy, a pesar de que hay muchos estándares, casi todas las soluciones de WSN tienden a ser propietarias. Y cada una resuelve sus problemas de una forma diferente. Especialmente en el ámbito de la seguridad. Por eso este trabajo intenta desarrollar un escenario sobre el que probar los distintos mecanismos de autenticación y seguridad. De esta forma se pueden probar las ventajas e inconvenientes de las distintas soluciones de seguridad. Y así evaluar cuál es la mejor opción para una WSN que se piense desplegar. Teniendo en cuenta consumo energético, dificultad de la implementación y/o mantenimiento (distribución y renovación de claves, certificados, etc.).

1.1 Motivación

El "Internet de las cosas" se ha convertido en un tema omnipresente en el ámbito tecnológico a día de hoy. Este fenómeno no tiene precedentes en lo que a temática y escala se refiere, y está transformando muchos sectores como el industrial. Los dispositivos llevan años conectándose a la red, pero ahora son más pequeños y móviles. El Internet de las cosas ofrece muchas ventajas, pero también supone un reto en el aspecto de la seguridad. Y es que resulta un riesgo tener toda nuestra información en la red si no se maneja de forma segura. Los dispositivos de este tipo de redes suelen disponer de pocos recursos. Ello hace que algunas medidas de seguridad, que requieren de alta carga computacional y un consecuente gasto de energía, deban ser usadas de manera eficiente. Por otro lado IPv6 es el futuro de Internet y parece ser que pronto será el protocolo que predomine. Con la idea de probar las distintas medidas de autenticación y seguridad en el ámbito de las redes de sensores con IPv6 nace este proyecto.

1.2 Objetivos

Este trabajo tiene como objetivo el desarrollo de un entorno de pruebas de mecanismos de autenticación y seguridad en redes de sensores con IPv6.

Se usará Cooja, que es un simulador de redes de equipos que usan Contiki OS, un sistema operativo para microcontroladores de bajo coste y consumo. Contiki permite a los dispositivos conectarse a Internet a través de protocolos de bajo consumo como 6LoWPAN (*IPv6 over Low power Wireless Personal Area Networks*). Se harán distintas pruebas en redes inalámbricas de sensores implementando medidas de autenticación y cifrado de los mensajes de la red. Comprobando que la comunicación es posible y examinando los mensajes intercambiados entre los nodos para el envío de los datos firmado o cifrado. Finalmente se evaluarán las distintas medidas probadas en base a aspectos como el consumo energético, retardo en las comunicaciones o aumento del tamaño de programa.

1.3 Fases de realización

Para llevar a cabo este trabajo se han seguido las siguientes fases en su desarrollo:

- **Documentación**: En esta fase se realizó un estudio de todo lo relacionado con el IoT. El propósito del mismo y su situación actual. Así como sus protocolos y plataformas más usadas. También se llevó a cabo un estudio de IPv6, su implementación para redes de sensores inalámbricas, 6LoWPAN. Y los distintos problemas de seguridad que existían en IPv6 y las soluciones que se dan en las redes de sensores para mitigar estas amenazas.

- **Preparación del entorno de trabajo**: En esta fase se barajaron las distintas alternativas para establecer el entorno de pruebas. Como no disponíamos de los recursos hardware necesarios, se optó por empezar a trabajar con el simulador Cooja de Contiki OS y así poder simular redes de sensores inalámbricas.

- **Selección y desarrollo de las medidas de seguridad**: De las medidas de seguridad buscadas en la primera fase se llevó a cabo una selección. Se usó la capa `llsec` (*link-layer security*) que incluye Contiki.

- **Realización de las pruebas:** Se probaron las distintas medidas de autenticación y cifrado en una WSN de dispositivos Contiki que se diseñó en el simulador Cooja.

- **Verificación de los resultados de las pruebas:** En esta fase se comprobó el correcto funcionamiento de las medidas de autenticación y cifrado de paquetes utilizadas.

- **Redacción de la memoria:** Por último, se llevó a cabo la redacción de este documento para transmitir la experiencia y conocimiento obtenido en el desarrollo de este Trabajo de Fin de Grado.

1.4 Estructura de la memoria

La memoria consta de las siguientes partes:

- En primer lugar, se hará un acercamiento al estado actual de Internet (Internet de las Cosas, IPv6 y seguridad TIC).

- Se evaluarán todas las tecnologías implicadas en el trabajo, así como las diferentes alternativas. Se explicará por qué se eligieron aquellas seleccionadas para formar parte del entorno de trabajo frente a las posibles alternativas. Una vez escogida la tecnología, se explicacará más exhaustivamente. Tratando aspectos técnicos.

- Después desarrollaremos las pruebas de autenticación y cifrado en la red de sensores. Para ello se usará el lenguaje de programación C.

- Una vez desarrolladas las pruebas pasaremos a realizarlas en el simulador Cooja y a analizar los resultados. Prestaremos especial atención a los mensajes intercambiados en los distintos escenarios y verificaremos el correcto funcionamiento de los mecanismos.

- Cuando se tenga un correcto funcionamiento de los escenarios, se estudiarán aspectos como el consumo energético, retardo en las comunicaciones y el incremento en el tamaño del programa que suponen las distintas medidas.

2 ESTADO DEL ARTE

"Los productos del IoT no están solamente orientados a la vida cotidiana. Tendrán un gran impacto en el mundo empresarial. E igual que aquellas compañías que ignoraron Internet al principio del siglo, aquellos que pasen del IoT se arriesgarán a quedarse atrás"

Jared Newman, en Fast Company

La evolución de las tecnologías de la información en los últimos tiempos ha provocado cambios significativos en nuestras vidas. Hoy en día es cada vez más frecuente que los objetos cotidianos estén conectados a Internet.

Internet ha contribuido bastante al desarrollo de las tecnologías de la información. Ha supuesto el acceso universal a la información. Ahora cualquiera puede tener acceso al instante a un volumen de información enorme. Pero no sólo eso, gracias a él surgieron nuevas formas de generación y transmisión de información por parte de los usuarios. De esta forma cualquiera puede aportar información de una infinita variedad de temas en la red.

Las mejoras en tecnología no sólo añaden cada vez más funcionalidades, sino que por lo general tienden a simplificar bastante lo que ya había. Y es que sin la capacidad de los usuarios para manejar de manera sencilla los dispositivos que se conectan a la red o la información que se encuentra en ésta, es imposible lograr un buen uso de los recursos.

De esta forma hemos pasado de una red orientada a clientes a una red que interconecta todas las cosas. De esta forma los clientes pueden seguir conectándose entre sí a través de múltiples dispositivos. Pero además los dispositivos se pueden valer de datos que recogen otros dispositivos, comunicándose a través de Internet. Y es aquí donde entra en juego el concepto de Internet de las Cosas. Este trabajo está pensado para usarse en el ámbito del IoT, y es por eso que lo estudiaremos a continuación.

Por otro lado el gran número de dispositivos a Internet ha sobrepasado con creces el límite que se pensó cuando se diseñó el protocolo que predomina en la red (IPv4). Y aunque existen varios factores atenuantes de este problema, el agotamiento de las direcciones IPv4 es un hecho. Por ello surgió un nuevo protocolo en Internet. IPv6 sería la solución a largo plazo para este problema. Y por eso lo estudiaremos también más adelante.

Otro aspecto por el que no se preocuparon excesivamente en el diseño original de Internet fue la seguridad. Esto hizo que cuando llegó la revolución del uso de Internet fuera muy fácil desarrollar amenazas en la red. En este aspecto no se avanzó demasiado en IPv6, no habiendo diferencias significativas en cuanto a seguridad. Más adelante hablaremos del estado de la seguridad y en concreto en redes IPv6.

2.1 IoT

La principal visión del IoT es crear un mundo en el que lo físico, lo digital y lo virtual converjan. De esta forma podremos crear entornos inteligentes que proporcionen mayor asistencia en sectores como la energía, la salud, el transporte, el control de las ciudades, industrias o edificios y muchas más cosas.

Todo ello sobre la base de una compleja red como es Internet, que interconecta miles de millones de dispositivos y personas. Internet es una red en la que participan múltiples tecnologías, múltiples protocolos y múltiples plataformas.

En el IoT los objetos recogen información del entorno y la comparten a través de Internet. Por ello este trabajo está pensado para servir de ayuda al desarrollo de este nuevo paradigma en Internet. Y así poder hacer que las redes de sensores inalámbricas del IoT dispongan de un entorno seguro para desarrollar sus funciones.

2.1.1 Introducción

Cuando Kevin Ashton, que trabajaba en el MIT (Massachussets Institute of Technology) en un grupo de investigación de identificación por radiofrecuencia (RFID), presentó este concepto en 1999[1] no tuvo demasiada repercusión. En esa época no todo el mundo tenía acceso a Internet y lo accedían habitualmente desde un ordenador en su oficina o domicilio.

En 2003 se estima que había conectados unos 500 millones de dispositivos a Internet, siendo la población global de 6300 millones de habitantes.

"Hoy en día, los ordenadores (y, por tanto, Internet) dependen casi totalmente de los humanos para obtener la información que manejan. Prácticamente la totalidad de los datos alojados en Internet han sido generados por humanos. Pero tenemos atención, precisión y tiempo limitados; luego no podemos considerarnos como el medio ideal para obtener información del mundo real".

Si hubiera equipos que pudieran saber todo acerca de todas las cosas, obteniendo los datos sin ayuda de nadie, podríamos controlarlo todo; reduciendo gastos, pérdidas y costes".

Gracias al avance de la tecnología cada vez fue posible llevar Internet a más dispositivos. El mayor impulsor fueron los teléfonos móviles que ofrecían la posibilidad de acceder a la red. Gracias a ellos en 2010 la cifra aumentó a 12500 millones de dispositivos conectados a Internet, mientras que la población global sólo había crecido un 8% (6800 millones de habitantes)[2].

Estos últimos años con el comienzo del IoT el aumento de dispositivos conectados a Internet ha sido tremendo. El número de dispositivos conectados en 2015 sea ya de 25000 millones. La población global, sin embargo, es de 7200 millones de habitantes y aún hay muchas personas que no tienen acceso a Internet.

Resulta entonces evidente que el IoT es la evolución de Internet. Tiene mucho potencial y supone ya mucho más tráfico generado que el de las propias personas.

---

[1] That 'Internet of Things' Thing. (http://www.rfidjournal.com/articles/view?4986)

[2] The Internet of Things. How the Next Evolution of the Internet Is Changing Everything. Dave Evans, Cisco April 2011 (http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf)

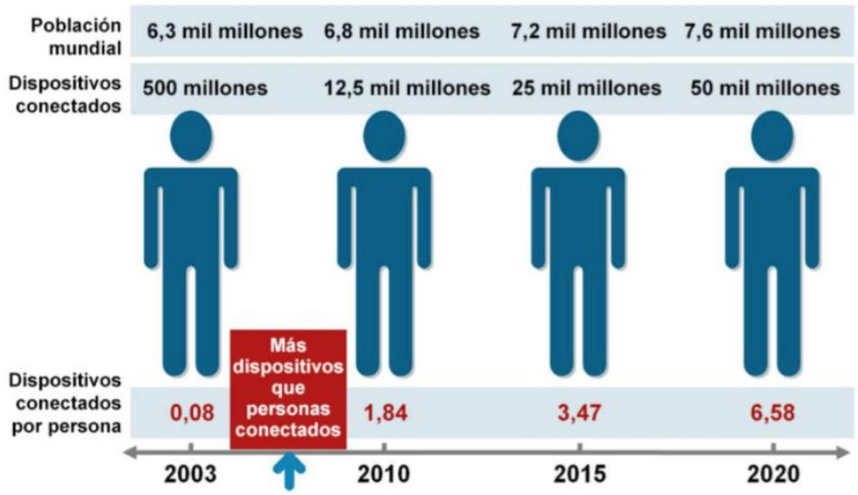

Figura 2-1. Estimación del número de dispositivos conectados a Internet desde el año 2003 al 2020. Fuente: Cisco IBSG, abril de 2011

2.1.2 Sectores de aplicación

La buena integración de estos pequeños dispositivos electrónicos en casi cualquier entorno, garantiza que veremos una gran variedad de aplicaciones para ellos.

Estos dispositivos tienen identidad virtual propia (son identificables de forma unívoca en la red) y por ello pueden interactuar de manera independiente en Internet con máquinas (M2M) o personas. Esto les permite integrarse en sectores de diversa índole.

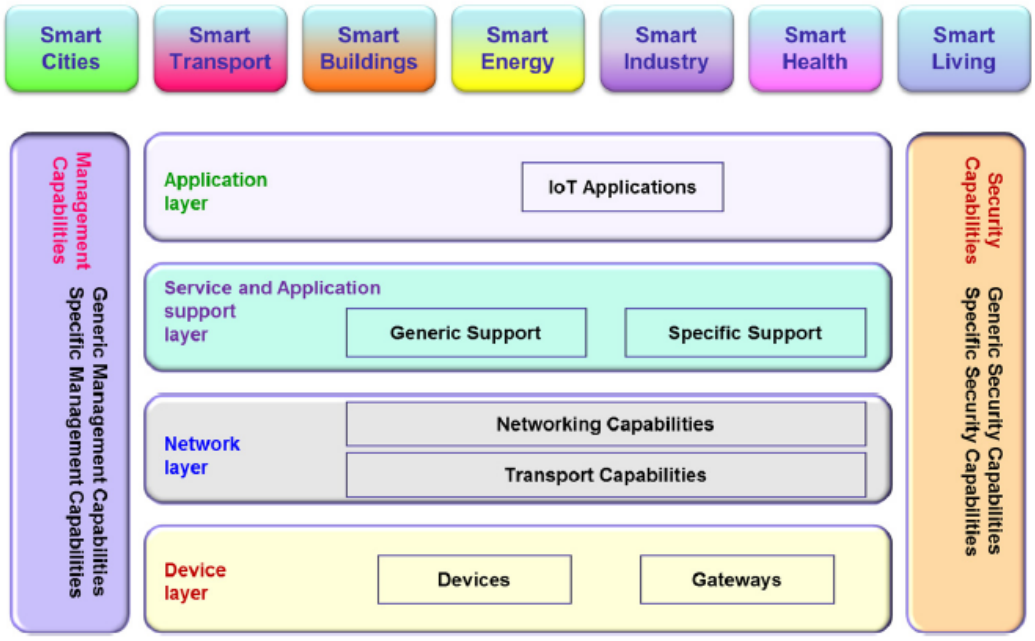

Figura 2-2. Estructura por capas y sectores de las aplicaciones del IoT. Fuente: ITU-T

Pronto habrá sectores que se vean revolucionados por el IoT.

- **Sanidad**: En el ámbito de la sanidad veremos aplicaciones como asistencia y seguimiento remoto o recordatorio de dosis de medicación. Los sensores pueden medir datos como las señales neuronales, el pulso o electrocardiogramas. Y además en algunos casos, pueden interpretarlo, filtrarlo y transmitirlo para avisar rápidamente si existe alguna anomalía.
- **Automovilismo**: Coches inteligentes que pueden comunicarse entre ellos o recopilar información sobre el tráfico.
- **Domótica**: En este ámbito existen infinidad de aplicaciones posibles para las WSN y el IoT. Casas inteligentes donde podamos controlar los enchufes para conseguir eficiencia energética. Luces y electrodomésticos cada vez más inteligentes, como por ejemplo neveras que nos avisan qué nos falta y lo piden directamente al supermercado. O también seguimiento para objetos perdidos.
- **Agricultura y ganadería**: Para gestionar y optimizar la cosecha y el ganado. Se pueden resolver muchos problemas en la gestión de granjas con las WSN. Si se tienen los datos apropiados, se puede maximizar la producción, consiguiendo además respetar el medio ambiente. Y también se puede usar para controlar el almacenamiento de mercancías que deban estar bajo condiciones controladas (temperatura, humerdad, luz, etc.).
- **Industria de la fabricación**: Haciendo más sencilla la gestión de una cadena de producción.
- **Servicios de seguridad y emergencias**: Con sensores y cámaras no intrusivas que permitan comprobar el estado de la propiedad remotamente. Así como sistemas de detección de inundación. Que permitan evacuar rápidamente edificios o zonas en riesgo.
- **Protección del medio ambiente**: También redes de sensores que estén desplegadas estratégicamente en un bosque, podrían alertar de incendios. O para hacer un seguimiento de animales. De forma que sea más fácil saber el estado y posición de animales de especies en peligro de extinción.
- **Educación**: con sistemas de enseñanza remotos.
- **Servicios de transporte:** para optimizar las rutas y permitir que los usuarios puedan saber exactamente cuánto tiempo les tardará en llegar un pedido en tiempo real.
- **Ocio y la venta de minoristas**: mejorando la satisfacción del cliente y el control de costes.
- **Ciudades inteligentes**: Controlando por ejemplo los semáforos en función del tráfico en tiempo real o del número de peatones, medir temperatura, energía, luz y humedad de forma automática, control del nivel de contenedores de basura para una recogida óptima por parte de los basureros o también monitorizar sitios libres en un aparcamiento.
- **Publicidad**: Donde se podría hacer un seguimiento a la gente que entra en la tienda y ver qué productos buscan y cuáles les llaman la atención, mostrando después los resultados por distintas edades y perfiles para conocer el público objetivo al que dirigir la publicidad.
- **Investigación geofísica**: Para medir por ejemplo actividad sísmica en zonas volcánicas para detectar posibles erupciones volcánicas.

Todas estas cosas impensables hace algunos años, son el futuro inminente de la tecnología hoy en día.

2.1.3 Principales tecnologías

Existen varios factores tecnológicos que han hecho posibles el despligue del IoT. En la figura 2-3 puede verse cómo los sensores, el procesado de los datos y la comunicación de los mismos resulta esencial en las redes IoT. A continuación hablamos de por qué son los factores más importantes.

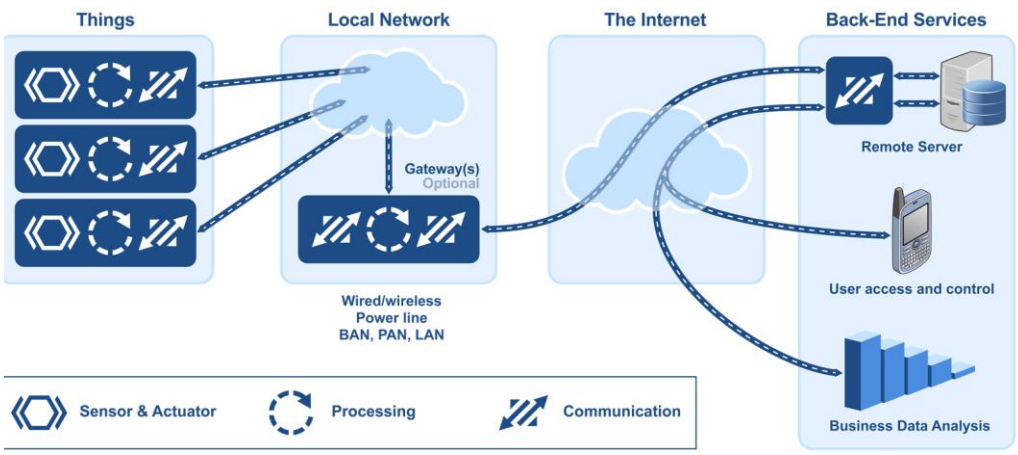

Figura 2-3. Ejemplo de red IoT. Fuente: Micrium

2.1.3.1 Procesadores

La tarea de estas unidades consiste en procesar localmente la información que se recoge del entorno y la información que otros sensores recogen en el entorno. En el presente, los procesadores están limitados en términos computacionales y energéticos.

Los procesadores tendían antaño a subir cada vez más la velocidad de reloj y elevar así su potencia. Pero a medida que los equipos portátiles fueron apareciendo, aparecieron otros requisitos. Menor consumo para aumentar la vida de las baterías. O menor tamaño para hacer equipos cada vez más pequeños y fáciles de transportar y que se calienten menos.

El caso del IoT resulta muy complicado desde la perspectiva de los procesadores. En el ámbito de los PC hay menos de una decena de factores de forma, en el ámbito de la computación móvil nos encontramos con unas decenas, pero es que en el IoT la diversidad es tan grande que puede haber millones de factores de forma. No se requiere el mismo tamaño y eficiencia energética para un dispositivo de seguimiento para la salud que para un sistema de aire acondicionado automatizado o un controlador de un motor eléctrico.

La arquitectura de microprocesadores ARM vino a satisfacer estas necesidades. Aunque no son tan potentes como los de otras arquitecturas como x86 ó x64. Pero aportan mucho menos consumo y están disponibles en muchos tamaños para adaptarse a las necesidades que surjan.

2.1.3.2 Sensores

Los procesadores gestionarán la información, pero ésta han de obtenerla a través de dispositivos sensores. Son elementos hardware que captan los datos del entorno que nos interese medir (temperatura, humedad, luz, altitud, presión, botones, movimiento, presencia, etc.).

Antes la electrónica estaba muy orientada al ámbito profesional. Pero gracias a plataformas de hardware libre como Arduino ahora mucha más gente puede experimentar en el ámbito doméstico. En parte gracias al bajo coste de los componentes y la gran cantidad de dispositivos que existen en el mercado para infinidad de usos. Esto hace que cada día veamos nuevas aplicaciones que antes resultaban impensables.

Otra clave en este tipo de sensores es la energía que consumen. Ya que si estos sensores no son autosuficientes habría que estar cambiando periódicamente las baterías o pilas de todos los dispositivos. A día de hoy ya hay prototipos de nanogeneradores viables que generan electricidad a partir de elementos del medio ambiente. Pero aún no se ha extendido su uso. Aun así a día de hoy los sensores suponen por lo general muy poco consumo energético para la batería.

2.1.3.3 Comunicaciones

Las comunicaciones son vitales en el IoT para transmitir todos los datos que se van recolectando. Existen muchas tecnologías y protocolos de comunicación, desde las tradicionales por cable como Ethernet ó PLC o inalámbricas

como WiFi, Bluetooth, 3G ó RFID.

En la mayoría de aplicaciones del IoT no se requerirá una alta tasa de transmisión de datos. Y es ahí donde entran en juego nuevas tecnologías de comunicación como Zigbee, Bluetooth 4.0 (*LE – Low Energy*) ó NFC que tienen su enfoque en la movilidad y el bajo consumo. En este proyecto trabajaremos con 802.15.4 que es la capa física y de enlace en la que se apoya Zigbee. Pero el protocolo de red que usaremos por encima de ella será 6LoWPAN que es una adaptación de IPv6 para redes de bajo consumo.

Además tenemos protocolos de aplicación como MQTT y CoAP por encima de los anteriores para hacer que las comunicaciones sean ligeras de lo que lo son con protocolos extendidos como HTTP. MQTT (*Message Queuing Telemetry Transport*) es un protocolo M2M ampliamente usado en el IoT para transmitir los datos que los sensores captan a un nodo central y éste repartirlo a los suscriptores del recurso en cuestión. Consume muy poco ancho de banda y requiere de muy pocos recursos por parte de los dispositivos. En el caso de CoAP funciona sobre UDP. Se basa en el modelo REST de HTTP con el conjunto de operaciones CRUD (Create, Read, Update, Delete).

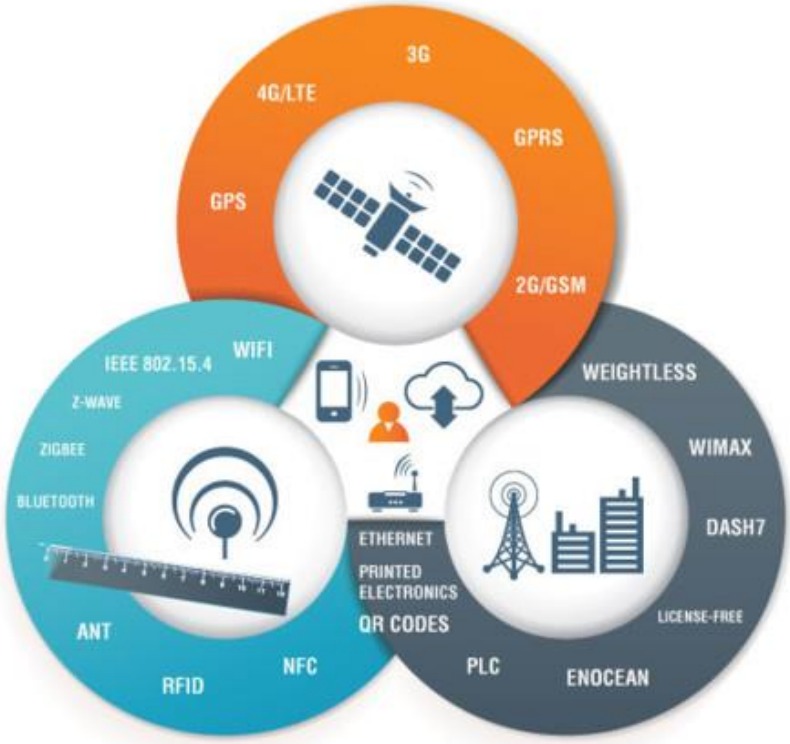

Figura 2-4. Tecnologías de comunicaciones. Fuente: Xataka

2.2 IPv6

IPv6 (*Internet Protocol version 6*) es el protocolo sucesor de IPv4, que es el estándar de facto en Internet. IP fue estandarizado por el IETF (*Internet Engineering Task Force*). Se diseñó como una solución para interconectar redes de datos que funcionaban con diferentes tecnologías. Hoy en día IP está presente en la mayoría de dispositivos que envían o reciben información digitalmente, no sólo a través de Internet.

La mayoría de sistemas operativos y librerías de red soportan IP para enviar y recibir datos. Esto es en parte

gracias a que está bastante estandarizado y se garantiza la interoperabilidad entre software de distintos fabricantes. Y es por ello que es de vital importancia para este nuevo paradigma del IoT donde todas las cosas están conectadas a Internet.

2.2.1 Introducción

ARPANET fue el primer intento de crear una gran red descentralizada capaz de comunicar diferentes sistemas. Su diseño y despliegue lo llevó a cabo el Departamento de Defensa de los Estados Unidos (DoD). El primer enlace de ARPANET se estableció en 1969. No fue hasta 1983 cuando la nueva pila de protocolos TCP/IP fue lanzada. El primer protocolo de red cuyo uso se extendió entre las personas particulares fue IPv4. Al principio las aplicaciones más usadas eran el correo electrónico y la transferencia de ficheros. Pero no fue hasta el desarrollo de la World Wide Web basada en HTML cuando Internet empezó a ser usado por grandes masas de personas. Esto hizo que el tráfico se disparara y como consecuencia se fueron agotando las direcciones que había disponibles. El protocolo IPv4 no estaba pensado para escalar a esos niveles.

Debido al agotamiento de direcciones IPv4, se lleva mucho tiempo insistiendo en la necesidad de implementar el protocolo IPv6. Y también gracias a las mejoras que aporta para extensiones del protocolo. El número de direcciones únicas en IPv6 es suficientemente grande como para no agotarse nunca, ya que cada habitante del planeta tendría a su disposición miles de millones de direcciones IPv6. Si bien es cierto que gracias al uso de redes privadas, NAT, DHCP y DNS (atendiendo por nombres, en vez de direcciones) entre otros, se ha conseguido poder seguir usando IPv4 y alargar lo máximo posible su uso. Sin embargo no dejan de ser soluciones a corto plazo. Con la llegada del IoT y el consecuente aumento del número de dispositivos conectados a la red, las conexiones "*always-on*" y el uso ineficiente de direcciones, el protocolo IPv4 tiene los días contados.

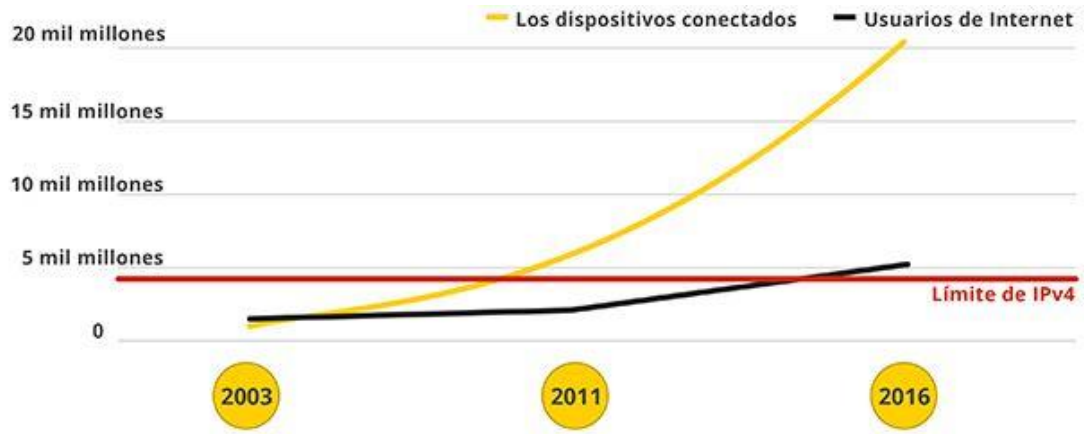

Figura 2-5. Agotamiento direcciones IPv4. Fuente: Google

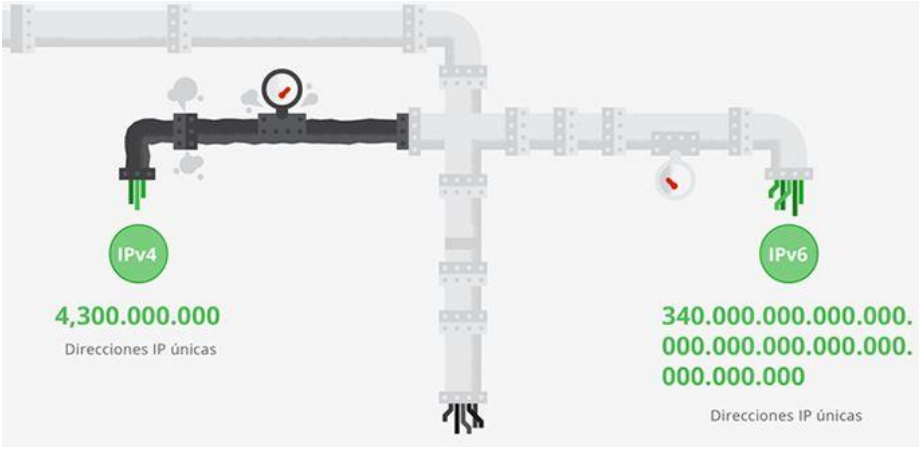

Figura 2-6. Número de direcciones IPv4 frente a IPv6. Fuente: Google

La realidad es que el protocolo IPv6 está cada vez más extendido, y así lo demuestran por ejemplo las estadísticas de Google.

Alrededor de un 10% del tráfico que accede a su portal lo hace a través de IPv6. Es decir, ya no es una cifra nada despreciable. Particularmente en países como EEUU la cifra aumenta a más de un 24% ó un 43% en países como Bélgica.[3]

Este protocolo encaja muy bien con el IoT. Ya que si vamos a tener conectados a Internet la mayoría de objetos nos será imposible direccionarlos con el actual protocolo IPv4. Tiene también bastantes mejoras en el ámbito de la movilidad tanto de los nodos finales como de los nodos encaminadores.

Resuelve la barrera del NAT y ahora sí, cada equipo podrá tener su propia IP pública. El uso de NAT suponía que no hubiera una conexión IP extremo a extremo. En lugar de eso había un equipo intermedio que cambiaría la IP origen/destino por la suya según si los paquetes salen/entran de la red. De esta forma, con el uso de IPv6, los equipos serán directamente accesibles desde la red.

Existe una versión comprimida de IPv6 llamada 6LoWPAN. Esta versión está pensada para dispositivos con limitaciones en recursos como memoria, batería, capacidad de procesamiento, etc. 6LoWPAN reduce el espacio de direcciones de IPv6 y requiere de encaminadores que traduzcan esas direcciones comprimidas a direcciones IPv6 normales.

Por eso lo escogimos para este Trabajo de Fin de Grado. Porque es esencial para evitar que se frene el crecimiento de Internet y permite que todos los dispositivos se comuniquen entre sí directamente. IPv6 abre un nuevo campo de posibilidades infinitas para el desarrollo de nuevos servicios.

[3] Google IPv6 estadísticas 2016 (https://www.google.com/intl/es/ipv6/index.html)

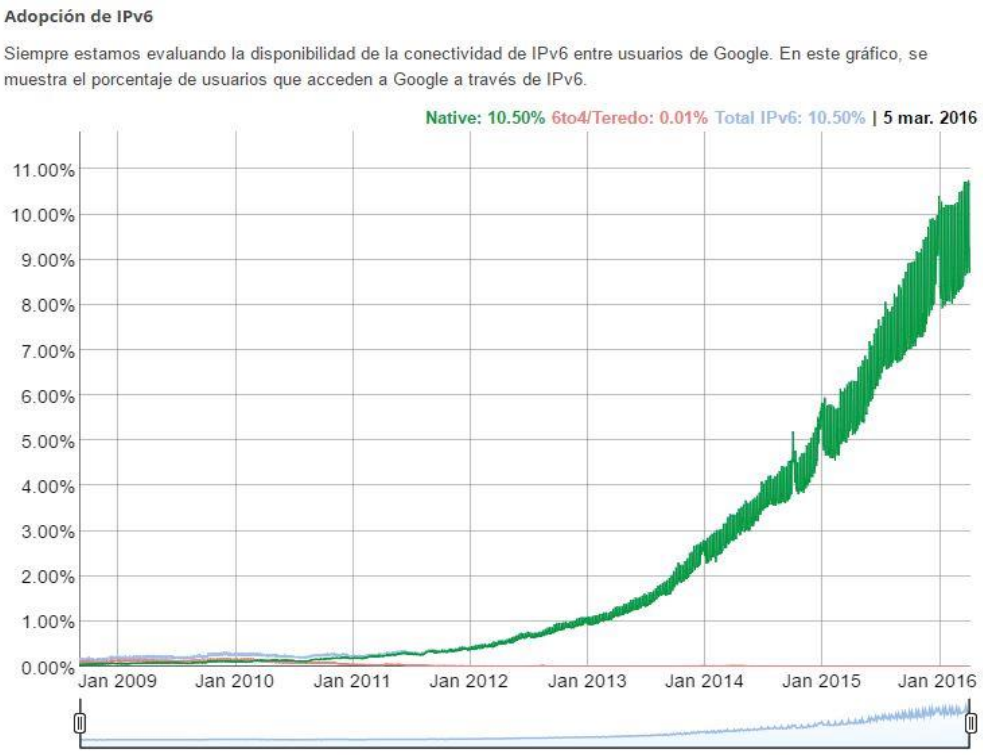

Figura 2-7. Tráfico IPv6 global en el acceso a Google. Fuente: Google

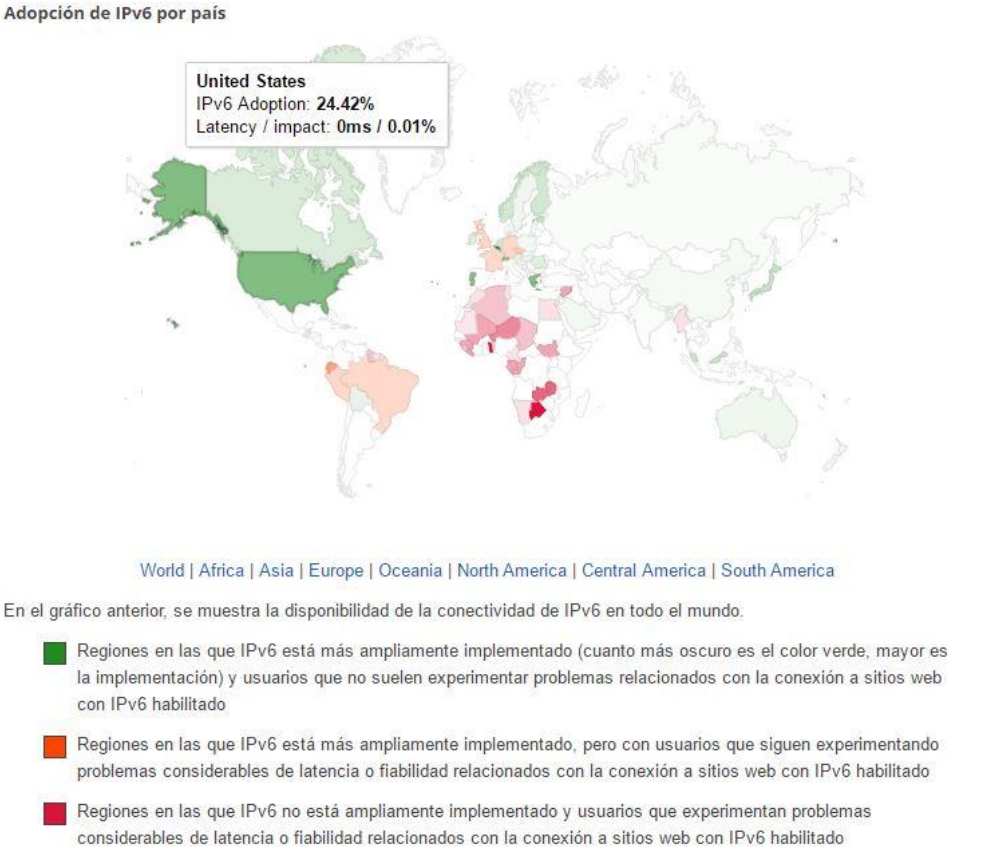

Figura 2-8. Mapa del tráfico IPv6 global en el acceso a Google. Fuente: Google

2.2.2 Conceptos

En este apartado se repasarán los conceptos mínimos de IPv6 para entender por qué es útil para el IoT. Veremos cómo está relacionado con el protocolo 6LoWPAN que veremos en el siguiente capítulo. Y su utlidad por tanto en el ámbito de las WSN.

La cabecera de los paquetes IPv6 ha cambiado con respecto a la de IPv4:

- **Número de campos**: Ha sido reducido de 12 a 8.
- **Tamaño de la cabecera**: La cabecera IPv6 se ha fijado en 40 bytes y se ha alineado a 64 bits. Esto permite que se pueda llevar a cabo un reenvío de paquetes más rápido en los routers basados en hardware.
- **Tamaño de las direcciones**: Se ha incrementado de 32 bits a 128 bits.

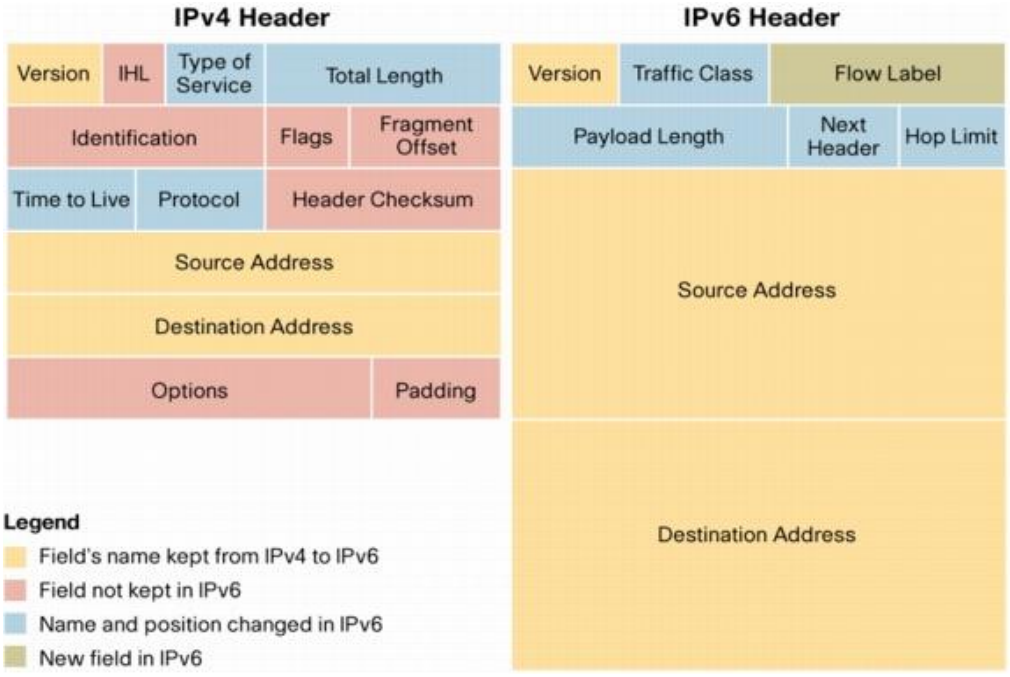

Figura 2-9. Diferencias entre las cabeceras IPv4 e IPv6. Fuente: Cisco

Los campos más importantes son la dirección destino y origen. Ya que son los que permiten identificar a cada uno de los dispositivos conectados a Internet. Como ya vimos en el apartado anterior, esto permite que haya $2^{128} = 3{,}4 \times 10^{38}$. En vez de la cantidad que había en IPv4 $(2^{32} = 4.294.967.296)$, que se ha quedado pequeña para el aumento de direcciones que supone el IoT.

Por otra parte tenemos las cabeceras de extensión. Que mantienen la cabecera con un tamaño fijo y permiten añadir funcionalidades a IPv6 después de la cabecera. Existen muchas cabeceras de extensión definidas, en la figura podemos ver el orden en el que deben aparecer.

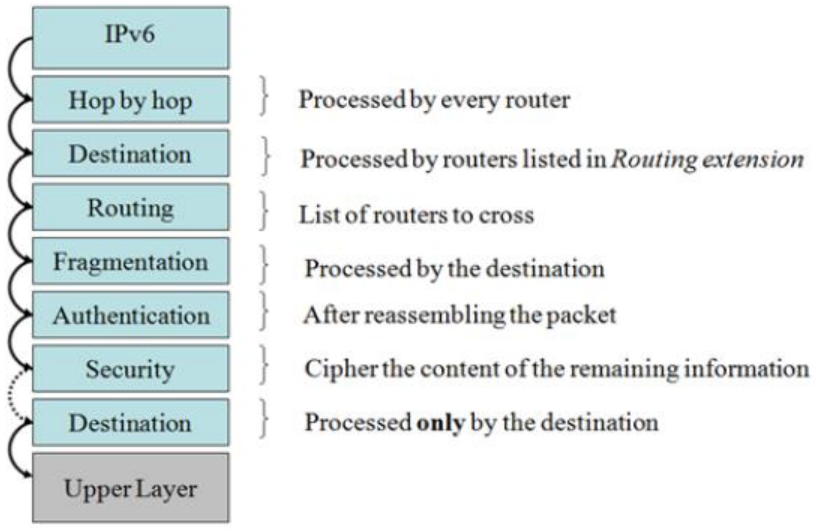

Figura 2-10. Cabeceras de extensión en IPv6

Las cabeceras de extensión por tanto proporcionan:

- **Flexibilidad**: Permitiendo por ejemplo el uso de seguridad para cifrar los datos del paquete.
- **Optimización en el procesado del paquete**: Con excepción de las cabeceras de extensión salto a salto (*hop by hop*), todas las demás se procesan solamente en los nodos extremos de la comunicación. Esto permite que cada router en el camino no tenga que procesar el paquete.
- **Cadena de cabeceras de extensión**: Las cabeceras de extensión se van añadiendo una detrás de otra. Empezando por la cabecera básica se va apuntando a la siguiente cabecera con el campo de siguiente cabecera.

En cuanto al direccionamiento ampliado de IPv6, sus beneficios son los siguientes:

- **Cantidad de direcciones**: Ya se ha visto que se proporcionan muchas más direcciones. Que permitirán satisfacer la demanda actual y futura.
- **Autoconfiguración**: Se simplifican los mecanismos de autoconfiguración para la asignación de direcciones.
- **Gestión**: La gestión de las direcciones resulta más sencilla para los organismos que se encargan de repartir las direcciones. Pues ahora se tiene espacio suficiente para asignarlas jerárquicamente. Se simplificará la tarea de enrutamiento gracias a la agregación de rutas.
- **Seguridad**: Al eliminar la NAT, se podrá usar IPsec de extremo a extremo.

Los tipos de direcciones IPv6 son los siguientes:

- **Unicast**: Se usan para enviar un paquete desde un origen a un solo destino. Éstas son las más habituales.
- **Multicast**: Sirven para enviar un paquete desde un origen a varios destinos. Es posible gracias a los protocolos de enrutamiento multicast, que permiten a los routers saber dónde se encuentran los destinos que esperan recibir los paquetes. También se usan en el mismo enlace para establecimiento de vecindades y otras tareas.
- **Anycast**: Permite el envío de un paquete desde un origen hasta el destino más cercano dentro de un conjunto de posibles destinos.

- **Reserved**: Son grupos de direcciones que se encuentran reservados para usos especiales. Por ejemplo los que se usan para documentación o ejemplos.

Las direcciones se componen de dos partes: el identificador de red (64 bits) y el identificador de interfaz (64 bits). El identificador de red viene a su vez a dividirse en dos partes: el prefijo de enrutamiento (48 bits o más) y el identificador de subred (16 bits o menos).

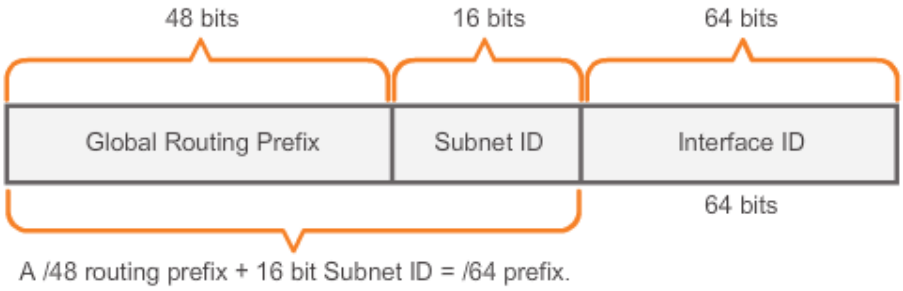

Figura 2-11. Estructura direcciones IPv6. Fuente ciscolinux.co.uk

Y dentro de las direcciones unicast encontramos también varios tipos

- **Link-local**: Direcciones locales al enlace. Están siempre presentes en una interfaz IPv6 que esté conectada a una red. Y todas empiezan por el prefijo `FE80::/10`. Los dispositivos las usan para comunicarse con otros equipos en la misma red local. Pero no se pueden usar para mantener comunicaciones con otras redes.
- **Unique Local**: Llamadas también ULA (*Unique Local Address*). Todas empiezan por `FC00::/8` ó `FD00::/8`. Se usan para comunicaciones locales dentro de una misma organización. No son enrutables dentro de Internet.
- **Global Unicast**: Son el equivalente a las direcciones públicas IPv4. Son únicas en todo Internet y se pueden usar para enviar paquetes desde un sitio a cualquier destino en Internet.

Con lo visto podemos apreciar el potencial de IPv6 para el IoT. Podemos ponerle direcciones IPv6 unicast globales a todo dispositivo que queramos. De esta forma todos los dispositivos pueden colaborar entre sí de una forma antes impensable. Podemos imaginar lo que supone para las redes de sensores desplegadas por todo el mundo. Que gracias a esta tecnología podrán enviar todos los datos recolectados o ser accedidos desde cualquier parte del mundo para crear una red a escala global que tenga las mediciones de los valores físicos. Si todos esos valores se almacenan y se procesan, se pueden tener aplicaciones como las que vimos en el apartado anterior.

Para obtener las direcciones ha surgido un nuevo mecanismo llamado SLAAC (*Statless Addresss Autoconfiguration*) que no existía en IPv4. De forma que ahora las formas de asignar una dirección a una interfaz son las siguientes:

- **Estáticamente**: Se configura manualmente la dirección IPv6 que le quieres asignar a tu dispositivo. Normalmente también tendrán que configurarse manualmente otros parámetros. Como por ejemplo la pasarela para enviar los paquetes fuera de la red.
- **DHCPv6**: Tiene la misma función que el DHCP en IPv4. Necesitas un servidor que asigne dinámicamente las direcciones a los equipos. El servidor mantiene un registro de las direcciones asignadas. Por eso se llama autoconfiguración de las direcciones con estado.

- **SLAAC**: Autoconfiguración de la dirección independiente del estado [4]. Es un nuevo mecanismo introducido en IPv6. Permite configurar automáticamente todos los parámetros de un dispositivo IP usando el router que da conectividad a la red. La ventaja de SLAAC es que simplifica la configuración de dispositivos con pocos recursos.

La ventaja de usar SLAAC es que se simplifica mucho la configuración de dispositivos con menos recursos. Como por ejemplo los sensores que usaremos en este trabajo.

2.3 Seguridad

Si hay una cosa en la que la gran mayoría de la comunidad de Internet está de acuerdo es en que la seguridad es una prioridad ahora mismo en Internet. Es necesario tenerla en cuenta a la hora de desarrollar nuevos sistemas y protocolos.

Después de ver los grandes avances que ha supuesto y supondrá en el futuro para la sociedad el IoT, hay que tener en cuenta los grandes riesgos que conllevará su implantación si este tipo de equipos y redes son objeto de ciberataques. Al fin y al cabo, las organizaciones dependen para su funcionamiento de la información que manejan. Si no la tienen disponible, las organizaciones no podránn trabajar. O en el caso de que ésta se vea alterada por alguien que no debería tener acceso a ella, la organización estaría funcionando sobre datos inválidos.

He aquí los distintos servicios que se deben conseguir para tener un nivel de seguridad razonable en un sistema:

- **Confidencialidad**: Si la información que se transmite resulta importante y se pretende que sea secreta para dispositivos intrusos, habrá que hacerla confidencial. Normalmente ésto se consigue cifrando los datos con una clave que ambos dispositivos (origen y destino de la comunicación) conocen.
- **Autenticación**: Permite que los dispositivos que participan en una transmisión puedan asegurarse de que el dispositivo que transmite es realmente quien dice ser. De esta forma se evita que dispositivos ilegítimos puedan enviar información falsa en la red. Así como tampoco podrían suplantar la identidad de dispositivos legítimos. Para conseguirlo normalmente se acompaña el mensaje de un código de autenticación de mensaje (MAC). Este código es un *hash* del mensaje cifrado con una clave conocida por ambos. La clave puede estar asignada para un dispositivo en concreto o usarse la misma en toda la red.
- **Integridad**: Es importante que el mensaje se reciba exactamente de la misma forma en que fue enviado. De manera que nada lo altere por el camino a su destino o no haya pérdidas que alteren la información. Esto se consigue con un código redundante cíclico (CRC). Como este CRC podría ser alterado por un atacante, se suele usar el MAC, que además de autenticación proporciona integridad.
- **Disponibilidad**: Se encarga de asegurar que los nodos de la red de sensores presten el servicio para el que están pensado. Incluso aunque sufran ataques de denegación de servicio (DoS). Conseguir esto en las redes de sensores inalámbricas resulta extremadamente difícil, pues los ataques DoS pueden tener como objetivo muchas capas. Por ejemplo, atacando con interferencias a la señal de radio para evitar las comunicaciones. También pueden tratar de agotar la batería de los sensores enviando mensajes falsos para hacer que los sensores los procesen.
- **Autorización**: Se asegura de que sólo los nodos autorizados pueden acceder a los servicios y recursos de la red.

Si pensamos en los usuarios particulares u organizaciones, los peligros que pueden ocasionar fallos de seguridad en el IoT también son cuanto menos preocupantes. Hacer que todos los objetos se conecten a Internet tiene sus ventajas, siempre y cuando éstos lo hagan de forma segura.

Un atacante podría aprovecharse de la falta de seguridad en un dispositivo cuantificador de salud, obtener la localización de la víctima. O tener el control de dispositivos como marcapasos, bombas de insulina, etc. Poniendo en riesgo la salud de las personas.

[4] RFC 4862 (http://tools.ietf.org/html/rfc4862)

En las viviendas por ejemplo podría controlar las cerraduras o luces. O enviar spam a través de malware instalado en una nevera inteligente. En el ámbito de la conducción podrían aprovechar fallos de seguridad para abrir las puertas de un coche remotamente. O desactivar remotamente el sistema de control de frenos. Siendo un peligro para la seguridad de las personas.

Por tanto, vemos que la seguridad juega un papel crucial en el despliegue del IoT y en su éxito. Ya que si no se toman las medidas adecuadas se pueden llegar a dar lugar hechos totalmente catastróficos.

Un 73% de los ejecutivos del sector esperan que el IoT produzca un aumento de las amenazas de seguridad en los próximos dos años. [5] Por otro lado un 78% de los profesionales de la seguridad de las tecnologías de la información piensan que no tienen los medios suficientes y les falta control y visibilidad para gestionar y asegurar nuevos tipos de dispositivos conectados a la red. [6] A esto hay que añadir que un 46% piensan que las actuals políticas de seguridad no son válidas para los dispositivos del IoT. [7]Con esto podemos hacernos una idea de lo importante que va a ser el tema de la seguridad en este tipo de redes para los negocios pronto.

En cualquier caso, dependiendo de la naturaleza de la red y aplicación que tenga, necesitará estar más o menos asegurada. Normalmente, aplicar medidas de seguridad conlleva mayor dificultad para el despliegue y mantenimiento de la red, y también un mayor gasto de la batería de los equipos para, por ejemplo, llevar a cabo operaciones de cifrado y descifrado. Así que las medidas de seguridad se deben de usar en consonancia con el objetivo y las capacidades de la red y los equipos.

Las políticas de seguridad intentan garantizar la autenticidad, confidencialidad, integridad y disponibilidad de los datos que se manejan. En el caso de las redes de sensores puede que no nos interese que los datos que generan los sensores sean leídos por otro, porque sean confidenciales. Tampoco que alguien los falsifique haciéndose pasar por otro sensor o modificando los datos que generan los sensores. Ni que se lleve a cabo una denegación de servicio de los sensores y no dispongamos de sus lecturas. Todo depende de la aplicación y por eso en este trabajo se intentará montar un escenario para probar los distintos tipos de políticas para elegir la que más se adecue a la aplicación que se quiera desarrollar.

Existen varios tipos de mecanismos de seguridad: Preventivos, defectivos y correctivos.

- **Mecanismos preventivos**: Detienen ataques y por tanto actúan antes de que ocurra ningún incidente de seguridad
- **Mecanismos detectores**: Ayudan a detectar agentes no deseados. Normalmente emiten alertas y registrar incidencias. Actúan también antes de que ocurra algún incidente.
- **Mecanismos correctivos**: Corrigen las consecuencias de una intrusión o ataque. Actúan después de que el hecho haya ocurrido.
- **Mecanismos disuasivos**: Intentan desalentar a los agentes no deseados de que cometan un ataque.

En este trabajo nos centraremos en los mecanismos preventivos para las redes de sensores que funcionen con IPv6. Hay que tener en cuenta que estas redes pueden estar ubicadas en sitios apartados como vías de tren, subestaciones eléctricas, etc. Estos sitios pueden no estar lo suficientemente vigilados. Es por eso que la seguridad física aunque se vaya fuera del alcance de este trabajo, es importante tenerla en cuenta también a la hora de desplegar una red de sensores. Pues si un atacante tiene acceso físico a los equipos finales el riesgo de intrusión y sabotaje es bastante alto.

Por ello una buena solución de seguridad debe:

- Permitir que los sistemas críticos para el desarrollo del negocio sigan funcionando incluso aunque se esté siendo víctima de un ataque

[5] Global Market Insite (GMI), a division of Lightspeed Research, from a Cisco-sponsored study 2015. Cisco IoT System Security: Mitigate Risk, Simplify, Compliance, and Build Trust. Cisco White Paper (http://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/iot-system-security-wp.pdf)
[6] ibid
[7] SANS Institute, Securing the Internet of Things. Cisco 2015 (http://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/iot-system-security-wp.pdf)

- Proporcionar visibilidad en las aplicaciones, usuarios, protocolos y anomalías. De esta forma en caso de error o de ataque, éste pueda ser rápidamente detectado y arreglado.
- Escalar de forma rentable. De manera que se pueda dar soporte a muchos dispositivos y mucha información. Si por cada dispositivo que añadamos a la red tenemos que gastar mucho dinero, esfuerzo o tiempo para mantenerlo seguro, no nos vamos a poder permitir añadir muchos dispositivos a la red. Como consecuencia dispondremos de menos información. Por eso para una solución de seguridad sería deseable que escalase bien al añadir más equipos a la red, ya que cada vez tendremos más dispositivos conectados.
- Tener conocimiento de la situación. Para ello conviene tener identificados los dispositivos y usuarios de la red, disponer de registros, tener cámaras de vigilancia para detectar sabotaje de equipos, etc. Es importante para que en el caso de que un incidente ocurra se pueda acotar rápidamente el error y detectar las causas para poder arreglarlas con rapidez.

Los sistemas del IoT suelen presentar muchos puntos de vulnerabilidades. Desde comunicaciones sin cifrar hasta el diseño de la aplicación que llevan a cabo, pasando por débiles sistemas de autenticación, almacenamiento de datos inseguro y la ausencia de registros de eventos. Muchas veces por ahorrar recursos, pero en otras ocasiones simplemente por desconocimiento o para acelerar el desarrollo. Solamente el 48% de las compañías tienen en cuenta la seguridad desde el principio del desarrollo de sus productos para el IoT.[8]

Los dispositivos del IoT necesitan estar constantemente actualizándose para estar protegidos frente a las nuevas amenazas que van surgiendo. Aunque parezca una medida obvia para mitigar las amenazas solo un 49% de las organizaciones proporcionan actualizaciones remotas.[9]

[8] Securing the Internet of Things Opportunity: Putting Cybersecurity at the Heart of the IoT. Capgemini Consulting, Sogeti High Tech (https://www.capgemini.com/resource-file-access/resource/pdf/securing_the_internet_of_things_opportunity_putting_cyber_security_at_the_heart_of_the_iot.pdf)

[9] ibid

3 6LoWPAN sobre 802.15.4

El protocolo 6LoWPAN permite el uso de IPv6 en redes que funcionen bajo el estándar IEEE 802.15.4. Es un acrónimo de IPv6 sobre redes de área personal de sensores de bajo consumo. Fue desarrollado por un grupo de trabajo del IETF en su división del área de Internet. El concepto de 6LoWPAN viene de la idea de que el protocolo de Internet debería poder usarse incluso en el más pequeño de los dispositivos. De esta forma podremos comunicarnos directamente a través de una red inalámbrica de sensores con otros equipos que funcionen con IP. Esto se consigue gracias a una serie de mecanismos de compresión de cabeceras que permiten a los paquetes IPv6 poder ser enviados y recibidos en redes 802.15.4. Esto resulta un reto, pues este tipo de redes tienen muy poco espacio para carga útil en sus paquetes.

3.1 Otras tecnologías

Lo primero que hice fue investigar para ver cuáles eran las tecnologías más usadas en el ámbito de las redes de sensores. Pero uno de los requisitos fundamentales de este trabajo es que estuvieran basadas en IPv6. De forma que los sensores pudieran comunicarse directamente o bien a través de una pasarela con el resto de Internet. Después de llevar a cabo la búsqueda creí que la mejor opción para este trabajo era 6LoWPAN. Pero ahora veremos sus ventajas y desventajas frente al resto de tecnologías.

3.1.1 ZigBee

Actualmente la tecnología de bajo coste para redes de bajo consumo que más popularidad tiene es ZigBee. Existen muchos productos comerciales que hacen uso de esta tecnología y existen cada día muchas más compañías que apuestan por él.

ZigBee concretamente resulta ser un protocolo de capa de red como 6LoWPAN. Y se apoya en el estándar IEEE 802.15.4 de redes inalámbricas de área personal de bajo consumo. El mismo que usa 6LoWPAN. Por esto 6LoWPAN ha aparecido en el mercado como un competidor directo de ZigBee.

6LoWPAN aunque usa el estándar 802.15.4, puede funcionar en otras tecnologías de capas de nivel físico distintas a las que soportan el uso de ZigBee. Pero lo más importante es su integración con otros sistemas basados en IP. Y es que la interoperabilidad es un factor importante a la hora de escoger un protocolo de red inalámbrica.

En el caso de ZigBee se definen las comunicaciones entre nodos 802.15.4 (capa de enlace en el mundo IP) y después se definen unas capas superiores hasta llegar a la capa de aplicación. De esta forma los dispositivos ZigBee pueden trabajar con otros dispositivos ZigBee, asumiendo que utilizan el mismo perfil en sus comunicaciones. En este aspecto 6LoWPAN ofrece interoperabilidad con otros dispositivos inalámbricos 802.15.4 al igual que ZigBee, pero también con otros dispositivos de distintas redes IP basadas en tecnologías como Ethernet o Wi-Fi. Y es que pasar de una red ZigBee a una red distinta requiere de una pasarela a nivel de aplicación para llevar a cabo el cambio de red. En el caso de 6LoWPAN esta pasarela no trabaja a nivel de aplicación, sino a nivel de red. Traduciendo las cabeceras comprimidas de 6LoWPAN a IPv6.

Otro factor a comparar entre ZigBee y 6LoWPAN es el tamaño de los paquetes, sus cabeceras y el tamaño de la implementación de la pila para poder funcionar con la tecnología en cuestión. En el caso de 6LoWPAN no es

necesario añadir información de enrutamiento en la capa de red. Se hace a través de otros protocolos como RPL (*IPv6 Routing Protocol for Low-Power and Lossy Networks*). Esto hace que haya más espacio para carga útil en los paquetes 6LoWPAN. Y respecto al tamaño de la pila, en el caso de ZigBee, una pila con todas las funcionalidades ocupa alrededor de 90KB. En el caso de 6LoWPAN, la pila ocupa solamente unos 30KB.

En el aspecto de seguridad ambos están a la par, pues se valen del cifrado AES-128 incluido en el estándar 802.15.4. Aunque en el caso de 6LoWPAN existe la posibilidad de usar IPSec [10].

Respecto a la disponibilidad y el coste de ambos, tenemos que ZigBee está un paso por delante. La mayoría de compañías de la industria, como Texas Instruments, Freescale o Atmel, promueven y aportan chips 802.15.4. Estas compañías ofrecen incluso las pilas ZigBee de manera gratuita. El apoyo de ellas a 6LoWPAN no es todavía lo suficientemente grande. Sin embargo existe una pila 6LoWPAN de código abierto y compañías como Archrock y Sensinode están lanzando sus propias pilas 6LoWPAN.

Por tanto 6LoWPAN es muy atractivo, ya que está basado en IP que es el protocolo estándar en Internet. Sin embargo ZigBee es más popular y está más extendido entre los principales fabricantes y productos de la industria de las redes de sensores. Además la ZigBee Alliance ha lanzado recientemente ZigBee IP. Pero aún no hay muchas implementaciones de este nuevo protocolo, que viene a reunir las virtudes de ambos.

Finalmente después de hacer un balance nos decidimos por 6LoWPAN por lo anteriormente expuesto. Pero sobre todo principalmente por estar basado en IPv6.

3.1.2 Bluetooth Low-Energy (BLE)

Una importante tecnología de comunicación para redes de área personal es Bluetooth. Es además la tecnología más implementada a día de hoy en el IoT. Sobre todo en el ámbito de los wearables. La nueva versión de Bluetooth, Bluetooth 4.0 Low-Energy (BLE) se adapta a dispositivos de bajo consumo. El hecho de que prácticamente todos los teléfonos móviles lo implementen ha hecho que su difusión sea muy grande.

A partir de la versión 4.2 llevará un perfil de funcionamiento para dar soporte a IP. Lo hará a través de 6LoWPAN, y de esta forma podremos comunicarnos con dispositivos Bluetooth directamente a través de Internet. Hasta ahora la forma de hacerlo era con una pasarela a nivel de aplicación, por ejemplo un teléfono móvil, que hiciera de intermediario entre el dispositivo sensor y otro equipo IP. Se espera que para 2018 más del 90% de los teléfonos móviles soporten el estándar BLE [11].

Un aspecto importante para las redes de sensores es que BLE sólo soporta un salto. Es decir, que si el dispositivo origen no está en rango para transmitir y alcanzar al destino, no es posible la comunicación. Si se usa 6LoWPAN sobre 802.15.4 sí es posible su transmisión pasando a través de otros sensores de la red.

No obstante aún no hay muchos dispositivos sensores que implementen BLE. Y por ello, y la posibilidad de transmitir salto a salto dentro de la red, consideramos que la opción de 6LoWPAN sobre 802.15.4 era más atractiva para este trabajo.

3.1.3 Z-Wave

Z-Wave es una tecnología de comunicaciones de bajo consumo pensada sobre todo para el ámbito de la automatización del hogar. Está optimizado para aportar bajo retardo en paquetes de pequeño tamaño y tasas de hasta 100 kbit/s. Da soporte para redes completamente malladas sin necesidad de un nodo coordinador. Resulta muy escalable, pudiendo tener hasta 232 dispositivos en una misma red.

Su protocolo resulta más sencillo que otros como ZigBee, Bluetooth ó 6LoWPAN. Ello hace que desarrollar para él sea simple y rápido. Pero el único fabricante de chips Z-Wave es Sigma Designs, por tanto su disponibilidad resulta muy limitada. Y su uso no está muy extendido. Además de no interoperar directamente con Internet como sí hace 6LoWPAN, que sólo necesita una pasarela a nivel de red.

[10] Enabling IPsec as optional security level for 6LoWPAN on Contiki. Pablo Puñal Pereira. 2013

[11] Three Ways Bluetooth® Smart Technology Enables Innovation for the Internet of Things. (http://blog.bluetooth.com/three-ways-bluetooth-smart-technology-enables-innovation-for-the-internet-of-things/)

3.2 802.15.4

El estándar 802.15.4 define tanto el nivél físico (*PHY*) como el control de acceso al medio (*MAC*). Y se usa para la transmisión de datos en redes inalámbricas de área personal. Tiene unas tasas de transmisión de datos muy bajas, pero consume muy poca potencia y es especialmente robusto en redes con muchas pérdidas. Esta especificación ha ido ganando mucha popularidad y se ha convertido en el núcleo de tecnologías de comunicaciones inalámbricas como ZigBee y 6LoWPAN.

3.2.1 Funcionamiento y opciones

La especificación original es de 2003 e introdujo dos tipos de capas físicas. Una operando para tasas de transferencia de datos de entre 20 kbit/s a 40 kbit/s y otra que llega hasta los 250 kbit/s.

También se especifican dos tipos de dispositivos que usan 802.15.4: *Full Function Device* (FFD) y *Reduced Function Device* (RFD). Los primeros implementan la pila del protocolo completamente, de forma que pueden actuar como coordinador PAN (coordinador de la red de área personal) y hablar con todos los nodos de la red. El coordinador PAN es el principal controlador de la PAN. Proporciona sincronización entre los distintos nodos de la red a través de mensajes periódicos. Los dispositivos RFD por el contrario tienen una implementación muy simple del protocolo y está limitado a ser un nodo extremo en topologías complejas.

Las dos topologías básicas disponibles para las redes 802.15.4 son las de topología en estrella y las P2P (peer-to-peer). En la topología en estrella todos los nodos se comunican entre ellos a través de un coordinador PAN central. En el caso de la topología P2P todos los nodos pueden comunicarse directamente entre ellos. El resto de topologías son combinación de estas dos.

El estándar 802.15.4 especifica también dos modos de operación en la capa de acceso al medio: con ranuras temporales para la transmisión y sin ellas.

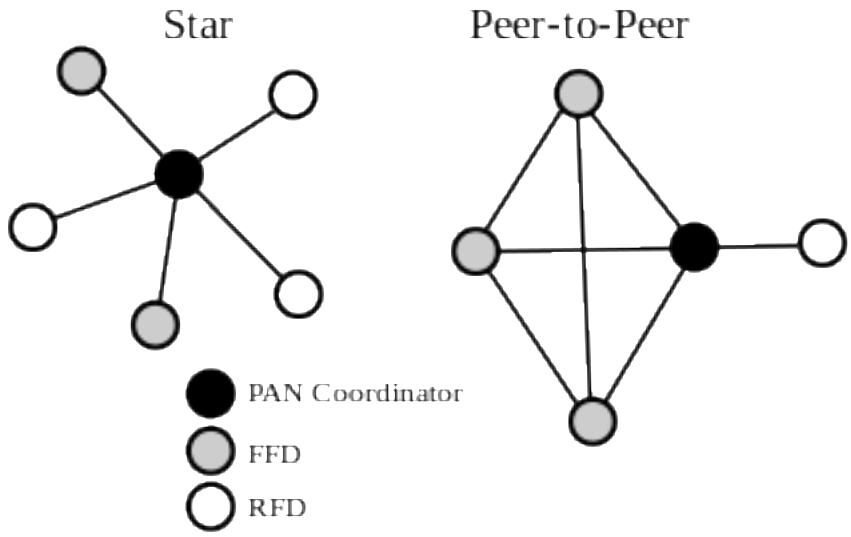

Figura 3-1. Ejemplo de topología en estrella y punto a punto en redes 802.15.4. Fuente: Wikipedia

3.2.2 Medidas de seguridad

Por defecto las comunicaciones 802.15.4 no proporcionan mecanismos de seguridad. Sin embargo, la especificación describe el uso de:

- **Solo cifrado**: Ofrece cifrado del contenido del paquete usando el algoritmo AES en el modo CTR. Este modo consiste en un cifrado por bloques donde además se tiene en cuenta un contador de bloques.
- **Solo autenticación**: Usa un MAC (*Message authentication code*) de 32, 64 ó 128 bits. Que consiste en realizar un hash al paquete. Luego se cifra el hash con una clave que sólo conocen los nodos legítimos de la red y se añade a la trama. De esta forma si el receptor de la trama realiza la función hash al paquete y no coincide con la que ha descifrado, puede saber que el paquete ha sido modificado y por tanto no lo tomaría como auténtico.
- **Cifrado y autenticación**: Además de añadir a la trama el código MAC se cifra el contenido del paquete. De forma que un nodo no legítimo no sólo no podría enviar mensajes, tampoco los podría leer.

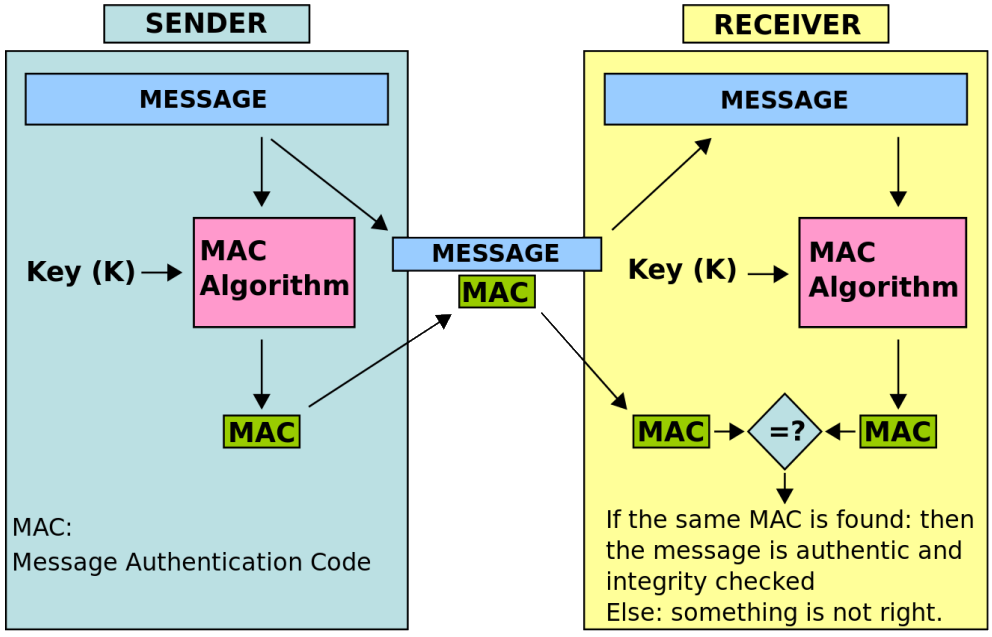

Figura 3-2. Ejemplo de algoritmo MAC. Fuente: Wikipedia

El coste de implementar estas medidas de seguridad tiene una contrapartida. Y es que requieren del uso de espacio extra en la carga útil de la trama. Esta carga útil es ya de por sí muy limitada. Por ejemplo en el caso del modo AES-CCM-128 se añaden 21 octetos a la cabecera 802.15.4. De esta forma la cabecera MAC del protocolo 802.15.4 pasaría a ocupar 46 bytes. Dejando sólo 81 octetos de espacio para las capas superiores. En la figura 3-2 podemos ver el formato de la trama 802.15.4.

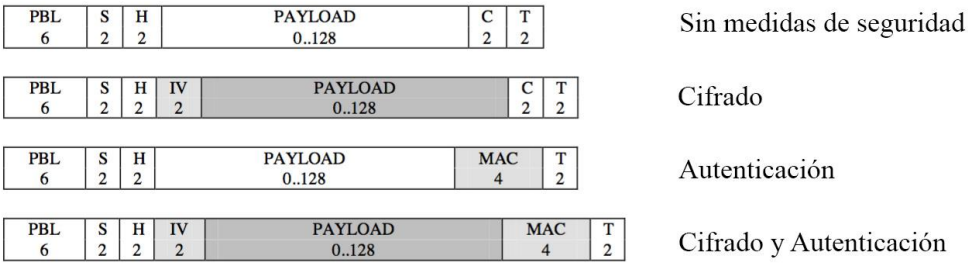

Figura 3-3. Cabecera 802.15.4 con el uso de las distintas medidas de autenticación y seguridad

Estas serán las medidas de autenticación y seguridad que probaremos en nuestro entorno de pruebas. La especificación no describe cómo llevar a cabo la gestión de las claves. En nuestro caso será una clave precompartida por todos los nodos legítimos de la red.

3.3 6LoWPAN

Al principio las redes de sensores estaban pensadas para recopilar información y enviarla a una estación base. Pero ahora el rol de los nodos ha cambiado en las WSN y se busca que sea un dispositivo independiente y pueda ser accedido desde Internet. De esta forma cualquier dispositivo que use IP podría hacer uso de él. Por eso se pensó que estaría bien poder usar IP en redes de sensores.

Sin embargo existen varios factores que impiden su uso directamente en redes 802.15.4. En la tabla 3-1 podemos ver las diferencias entre las redes 802.15.4 y las redes que usan IPv6.

Tabla 3–1. Comparación entre las redes 802.15.4 y las redes típicas IPv6

	Redes 802.15.4	Redes IPv6 típicas
Tamaño de los paquetes	Máximo 127 bytes.	Al menos una MTU de 1280 bytes.
Ancho de banda	Normalmente 250 kbit/s.	54 mbit/s (802.11g) ó 100 mbit/s (ethernet).
Potencia	Baja potencia de procesamiento y poca energía disponible. La mayoría de dispositivos funcionan con pilas.	No están limitados en este sentido. La mayoría de dispositivos están conectados a la red eléctrica.

3.3.1 Funcionamiento y opciones

Un requisito clave para el transporte de IPv6 es una MTU de al menos 1280 bytes por paquete [12]. Pero el estándar 802.15.4 fija el tamaño de los paquetes en 127 bytes. A raíz de este hecho nace 6LoWPAN. El IETF lo desarrolló en la RFC4944 [13]. Y se trata de una capa de adaptación para permitir la transmisión de datagramas IPv6 sobre redes inalámbricas 802.15.4.

Para el uso de 6LoWPAN será necesario trabajar en el modo no ranurado de la capa MAC de las redes 802.15.4. En la especificación se dice también cómo se deben fragmentar y reensamblar los paquetes IPv6, además de cómo se comprimen las cabeceras. Las cabeceras han sido actualizadas en la RFC 6282 [14]. Todos los datagramas encapsulados en tramas 802.15.4 deberán llevar una cabecera de encapsulación. En ella se indica el tipo de cabecera que va después (una cabecera sin comprimir o comprimida con IPv6 Header Compression definida en la RFC 6282).

En la especificación original había dos mecanismos de compresión de cabeceras, LOWPAN_HC1 y LOWPAN_HC2. Se basaban en omitir información que fuera posible calcular a partir del contexto en el que se estuviera. Lo malo de estos mecanismos es que valían de muy poco cuando era necesario el uso de tablas de enrutamiento. En el caso de LOWPAN_HC1, se requería la dirección IPv6 origen y destino entera (128 bits cada una).

A partir de la RFC 6282 se definió un nuevo formato de codificación de la cabecera: LOWPAN_IPHC. Para una compresión de las direcciones locales únicas, globales y multicast IPv6. Además también se definió LOWPAN_NHC para la compresión de las siguientes cabeceras. En el mejor caso en el que se use UDP sobre IPv6, se puede reducir a un total de 4 octetos.

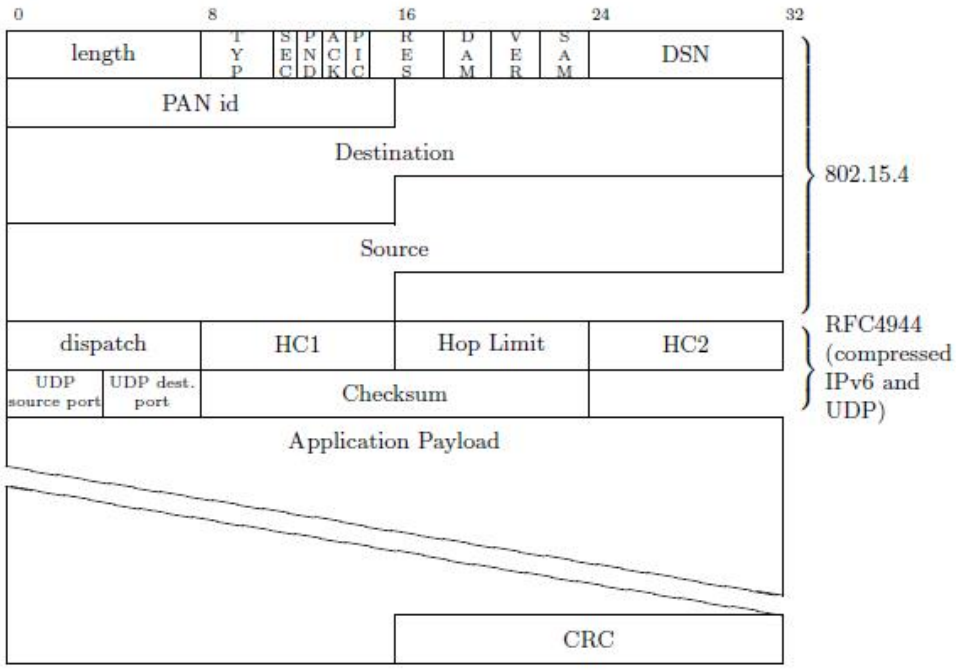

Figura 3-4. Cabecera 6LoWPAN sobre 802.15.4. Fuente: Atlassian

[12] RFC 2460 (https://www.ietf.org/rfc/rfc2460.txt)

[13] RFC 4944 (https://tools.ietf.org/html/rfc4944)

[14] RFC 6282 (https://tools.ietf.org/html/rfc6282)

3.3.2 Medidas de seguridad

En el apartado 3.2.2 se vieron medidas de autenticación y seguridad para redes basadas en 802.15.4. Estas medidas funcionaban a nivel de enlace. Y por tanto al salir de la red 802.15.4 no valdrían.

Existen soluciones para abordar la seguridad extremo a extremo en 6LoWPAN. IPsec (*Internet Protocol Security*) es un conjunto de protocolos que proporcionan cifrado y autenticación extremo a extremo. Se cifra y/o autentica cada paquete IP de la sesión de comunicación. Opera en la capa de red al contrario que otros protocolos como TLS ó SSH. De esta forma no es necesario modificar las aplicaciones para que IPsec se integre en ellas.

IPsec ofrece dos protocolos para proporcionar seguridad al tráfico de la red. Las cabeceras de autenticación (AH - *Authentication Header*) y el protocolo de encapsulamiento seguro (ESP - *Encapsulating Security Protocol*). El protocolo AH provée integridad en los datos y permite autenticar a su origen. Además ofrece mecanismos contra la repetición de paquetes. ESP además de todo esto cifra el contenido de los paquetes. Como en la mayoría de casos ESP proporciona la seguridad que se requiere, es obligatorio incluirlo a la hora de implementar IPSec. AH es opcional.

Ambos protocolos se pueden usar en dos modos: transporte y túnel. En modo transporte la cabecera IPsec. En modo transporte la cabecera IPsec se inserta después de la cabecera IP y los protocolos sólo proporcionan protección para la carga útil. Sólo se autentica o cifra el contenido del paquete IP. En el caso del modo túnel, se cifra o autentica todo el paquete IP. Luego se encapsula dentro de un nuevo paquete IP con una nueva cabecera IP.

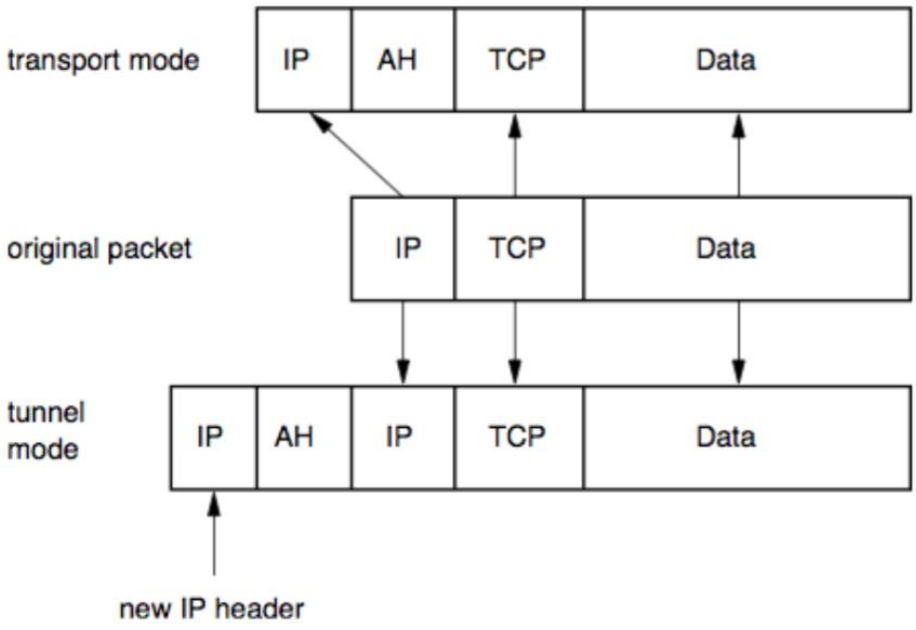

Figura 3-5. Modos IPsec

En IPsec existe el concepto de asociaciones de seguridad (SA - *Security Associations*). Una asociación de seguridad es una asociación entre dos extremos que usan IPsec que proporciona algoritmos y otros parámetros de seguridad, como por ejemplo las claves de cifrado. Estas asociaciones se pueden establecer manualmente, pero no resulta muy escalable. Se suele usar para establecerlas el protocolo IKE (*Internet Key Exchange*) ó IKEv2.

Otras medidas de seguridad para redes que usen 6LoWPAN sería el uso de SSL/TLS. Pero esto sería a nivel de aplicación y conlleva un gasto de recursos que no son asumibles por casi ningún tipo de sensores de bajo

consumo. Además, su uso depende de TCP. Y la pila TCP también resulta pesada para este tipo de sensores, además de implicar el establecimiento y mantenimiento de conexiones. Para securizar aplicaciones que usan UDP existe el protocolo DTLS (*Datagram Transport Layer Security*). Este protocolo es el equivalente a TLS para UDP. Pero igualmente, su implementación resulta muy pesada para incluirla en la memoria de los dispositivos.

En las pruebas del trabajo usaremos las medidas de seguridad a nivel de enlace vistas para redes 802.15.4. Por ser más simples. En cualquier caso el entorno de pruebas que desarrollemos servirá también para probar con algunas modificaciones IPsec. En el caso de SSL/TLS y DTLS sí se necesitarán más modificaciones a nivel de aplicación, pero igualmente se podría partir de lo desarrollado en este trabajo.

4 CONTIKI OS

Los sistemas operativos para dispositivos de redes de sensores inalámbricas han de ser más sencillos que los de propósito general para adaptarse y funcionar bajo las restricciones de capacidad de procesamiento, memoria, batería, etc. Contiki OS es un sistema operativo pequeño y de código abierto. Fue desarrollado por el Swedish Institute of Computer Science. Y está pensado para ser usado en dispositivos embebidos de redes de sensores inalámbricas con muy poca capacidad en memoria. Se puede portar a distintas plataformas de manera relativamente sencilla.

Contiki OS está escrito en C y necesita desde unos pocos bytes a varios kilobytes de memoria RAM, dependiendo de la configuración escogida (por ejemplo se puede escoger no usar la pila TCP para ahorrar memoria). Una configuración típica de Contiki ocuparía unos 40kB de memoria ROM y 2kB de memoria RAM. Está disponible para su descarga de manera libre bajo licencia BSD [15] y funciona en una gran cantidad de plataformas.

4.1 Estructura del sistema

El sistema operativo Contiki OS se compone de la siguiente estructura:

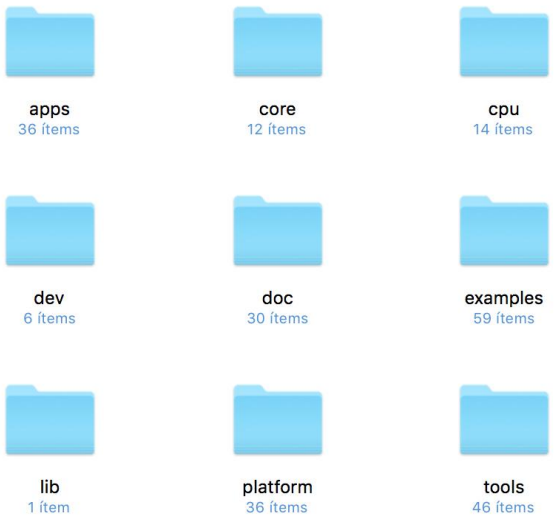

Figura 4-1. Ejemplo de proceso de envío y entrega de un evento de forma síncrona

[15] Contiki OS (http://www.contiki-os.org/)

Las carpetas más importantes son:

- **apps**: Aplicaciones que trae el sistema operativo. Usaremos en este trabajo algunas como `servreg-hack`, `powertracer` ó `collect-view`.
- **core**: Aquí se encuentra el núcleo del sistema operativo. Los módulos de radio, temporizadores, funciones de lectura/escritura, etc.
- **cpu**: Este directorio contiene las las implementaciones para cada uno de los microcontroladores que soportan el sistema operativo.
- **dev**: Incluye las librerías para el manejo de chips radiotransmisores externos.
- **doc**: Se trata de la documentación generada por `doxygen`.
- **examples**: En este directorio se tienen muchos ejemplos de aplicaciones para el sistema operativo. Resulta muy útil para aprender el funcionamiento del sistema y sus posibilidades. Nos hemos valido de algunos ejemplos para desarrollar el entorno de pruebas.
- **platform**: Contiene las librerías específicas para cada plataforma. En nuestro caso para la plataforma del dispositivo TMote Sky. Y se encuentran en otro directorio dentro de `platform` que se llama `sky`.
- **tools**: Aquí se encuentran las herramientas para cargar los programas a los dispositivos, depurar y simular. En nuestro caso usaremos el simulador Cooja y la herramienta Collect View para el entorno de pruebas.

4.2 Procesos

Contiki OS es un sistema con un kernel basado en eventos. Por encima de ese kernel, las aplicaciones son cargadas dinámicamente en tiempo de ejecución. El sistema es multitarea. Cada programa debe empezar al menos un proceso y sólo un proceso puede ejecutarse al mismo tiempo en CPU. Es responsabilidad de cada proceso pausar o parar la ejecución y prevenir que el sistema quede en punto muerto.

La comunicación entre procesos se hace a través de eventos. Cuando un proceso necesita avisar a otro de alguna novedad o suceso, añade un evento a la cola de eventos. El kernel va tomando eventos de la cola y se los hace llegar al proceso al que van dirigidos. También podría ir dirigido a todos los procesos en ejecución, en caso de que el evento vaya destinado a difusión.

El código en Contiki puede ejecutarse en dos tipos de contextos: preferente o cooperativo. El código cooperativo se ejecuta secuencialmente con respecto al resto de código cooperativo. Por el contrario, el código con preferencia para temporalmente la ejecución del código cooperativo.

Como antes se dijo, todos los programas de Contiki son procesos. Y los procesos son partes de código que el sistema operativo ejecuta cada cierto tiempo. Los procesos empiezan normalmente en el arranque del sistema. También puede ser que un módulo que contiene a un proceso lo cargue en el sistema. Los procesos siempre funcionan en el contexto cooperativo. El contexto preferente se reserva para las interrupciones en los drivers de los dispositivos y para tareas en tiempo real que tienen programadas un tiempo límite. Ya hablaremos de este tipo de tareas cuando hablemos de los temporizadores.

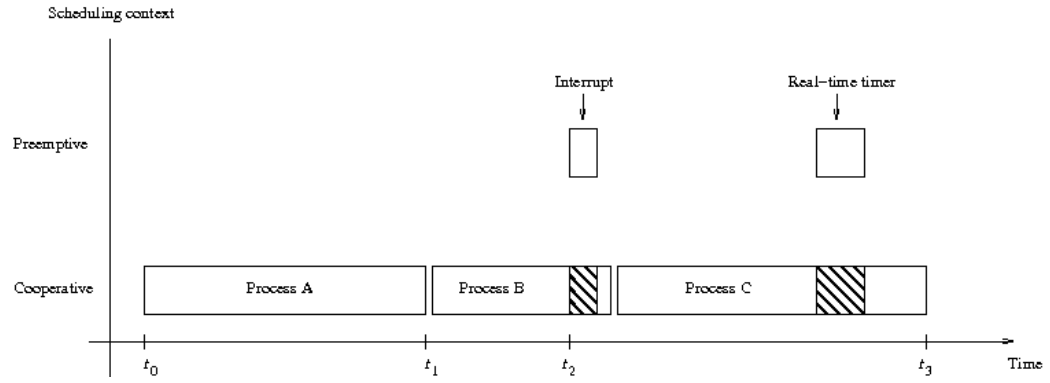

Figura 4-2. Ejemplo de ejecución de procesos en contextos preferente y cooperativo en Contiki

4.3 Protothreads

En Contiki existen los protothreads. Son hilos de procesos ligeros y no necesitan llevar una pila asociada. De esta forma funcionan de manera parecida a los hilos, pero sin incurrir en el gasto que supone el uso de hilos. Su uso está pensado para dispositivos que dispongan de muy poca memoria, como los dispositivos que se usan en las WSN. Su propósito es permitir un control secuencial del flujo sin recurrir a complejas máquinas de estado o sistemas multihilos complejos.

Un protothread permite estructurar el código de manera que el sistema pueda ejecutar otras actividades cuando el código está esperando que algo ocurra. De esta forma los protothreads permiten bloquear condicionalmente dentro de las funciones de la aplicación. Gracias a esto la estructura supera la limitación de la programación basada únicamente en eventos, en la que el bloqueo de las funciones hay que implementarlo dividiendo en dos partes la función de forma manual. Una parte que funciona antes de que se pause la ejecución y otra que se ejecuta tras el bloqueo. Esto hace muy difícil el uso de bucles `while` o sentencias condicionales `if`. Por eso los protothreads al no requerir una pila separada hacen el proceso de programación y funcionamiento más sencillos.

Y es que almacenar múltiples pilas consume grandes cantidades de memoria. Pero en el caso de los protothreads sólo se necesitan entre 2 y 12 bytes. Además no es un código exclusivo de cada máquina. Los protothreads están escritos en C.

Las macros que se usan en los procesos de Contiki para el manejo de protothreads son los siguientes:

```
PROCESS_BEGIN(); // Declara el inicio del protothread asociado al proceso.
PROCESS_END(); // Declara el final del protothread asociado al proceso.
PROCESS_EXIT(); // Sale del proceso.
PROCESS_WAIT_EVENT(); // Espera algún evento.
PROCESS_WAIT_EVENT_UNTIL(); // Espera a un evento, pero con una condición.
PROCESS_YIELD(); // Espera a algún evento. Es equivalente a PROCESS_WAIT_EVENT()
PROCESS_WAIT_UNTIL(); // Espera dada alguna condición. Puede que no cese el proceso.
PROCESS_PAUSE(); // Cesa temporalmente el proceso.
```

4.4 Eventos

En Contiki un proceso se ejecuta cuando recibe un evento. Hay dos tipos de eventos: síncronos y asíncronos. Los eventos síncronos se apilan en la cola de eventos del kernel y se entregan al proceso que espera recibirlo, pasado un tiempo. En el caso de los eventos síncronos, el evento se entrega inmediatamente después de enviarse.

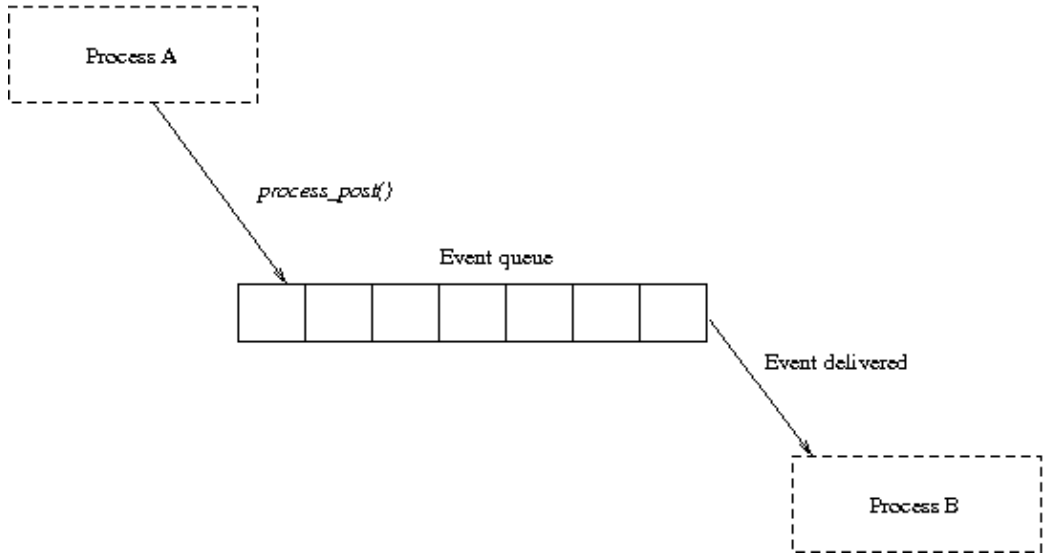

Figura 4-3. Ejemplo de proceso de envío y entrega de un evento de forma asíncrona

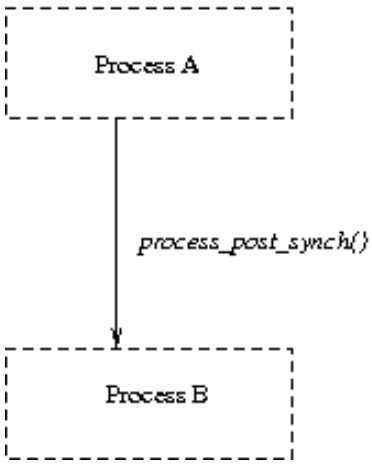

Figura 4-4. Ejemplo de proceso de envío y entrega de un evento de forma síncrona

Vistos los procesos, protothreads y eventos, un ejemplo de proceso en Contiki sería el siguiente:

```
#include "contiki.h"

PROCESS(Proceso_ejemplo, "Proceso de ejemplo");
AUTOSTART_PROCESSES(&Proceso_ejemplo);

PROCESS_THREAD(Proceso_ejemplo, ev, data)
{
```

```
  PROCESS_BEGIN();

  while(1) {
    PROCESS_WAIT_EVENT();
    printf("Se ha recibido el evento número %d\n", ev);
  }

  PROCESS_END();
}
```

4.5 Multithreading

Existe otra forma de usar programación concurrente en Contiki. Pero al contrario que con los protothreads, es dependiente de la arquitectura del procesador que se use. El código de la librería que implementa esta tecnología (mt library) se divide en dos partes: una independiente de la arquitectura del procesador y otra que depende de su arquitectura. Ha de cambiarse de contexto entre los distintos hilos y para ello se tiene una pila privada por cada hilo y un contador de programa que se guarda al cambiar de contexto.

La figura 4-5 muestra la máquina de estados de los hilos. Un hilo puede tener tres estados distintos durante su tiempo de uso:

- **Ready**: En este estado el hilo está listo para ejecutarse. Que se hace a través de la función `mt_exec()` del hilo. Este estado se puede reestablecer una vez que el control se cede a otro hilo usando `mt_yield()`.
- **Running**: Este estado indica que el hilo se está ejecutando.
- **Exited**: Es el estado final de un hilo. Significa que ya no se puede ejecutar más.

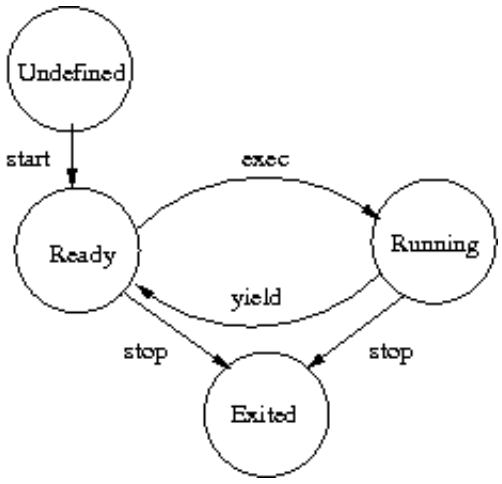

Figura 4-5. Máquina de estados de un hilo

Los hilos no pueden recibir eventos de Contiki y por lo general no se suelen usar. Pero Contiki los ofrece por si alguien los necesita en un programa o plataforma específica.

4.6 Temporizadores

Contiki OS también propociona una serie de librerías para el uso de temporizadores, que pueden usar tanto las aplicaciones, como el propio sistema operativo. Estas librerías contienen funciones para comprobar si un período de tiempo ha pasado, despertando al sistema desde el modo de bajo consumo en los tiempos programados. También permite programar tareas en tiempo real. Los temporizadores también sirven para dejar que el sistema trabaje con otras tareas por un período de tiempo antes de reanudar la ejecución.

4.7 Entradas y salidas

El tipo de dispositivos I/O que es usen depende altamente de la plataforma en cuestión. Contiki OS proporciona dos interfaces básicas que la mayoría de plataformas comparten: comunicación por puerto serie y LEDs.

La salida por puerto serie está soportada con la librería estándar C para imprimir (`printf` o `putchar`). Aquí un ejemplo para sacar algo por la salida del puerto serie:

```
int printf(const char *formato, ...); // Imprime una cadena conforme a un formato.
int putchar(int c); // Imprime un único carácter.
```

La entrada por puerto serie en cambio es a través de un mecanismo específico de Contiki. Será necesario primero llamar a la función `serial_line_init()` y así el proceso `serial_line_process` y el buffer puedan ser inicializados antes de empezar a recibir.

Cuando hay un carácter para ser leído del puerto serie se genera una interrupción. Y se van almacenando los datos en el buffer hasta que llega el carácter de nueva línea. Cuando esto ocurre se manda un evento a difusión a todos los procesos.

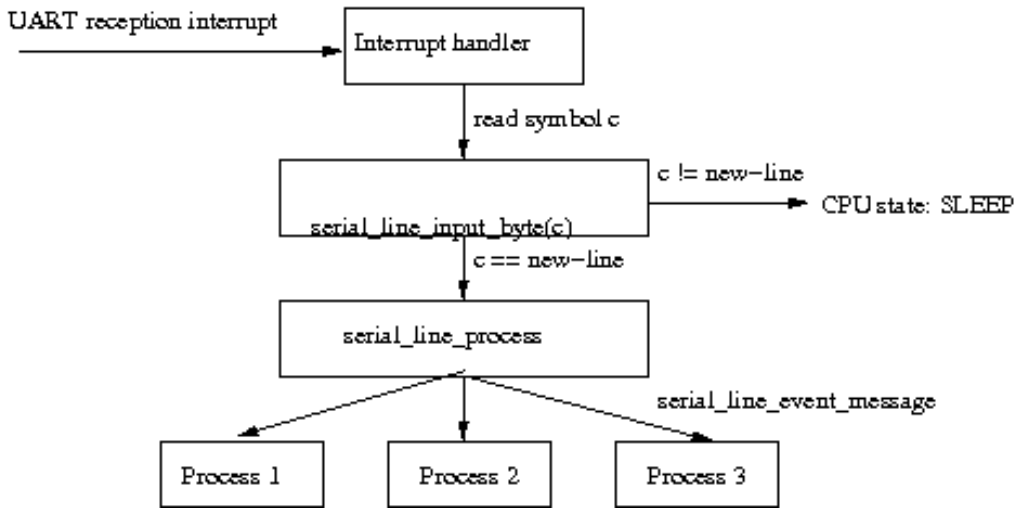

Figura 4-6. Máquina de estados de un hilo

Los LEDs no los usaremos en nuestro caso, ya que podemos ver la salida del puerto serie por pantalla en el simulador. Pero en entornos reales se usan bastante para depurar el funcionamiento de los programas en los sensores.

Para su uso es necesario inicializar la librería llamando a leds_init(). Luego podemos usar las funciones que

incluye la librería para modificar el estado del LED:

```
void leds_init(void); // Inicializa el driver de los LEDs.
unsigned char leds_get(void); // Obtiene el estado de un LED.
void leds_on(unsigned char ledv); // Enciende un conjunto de LEDs.
void leds_off(unsigned char ledv); // Apaga un conjunto de LEDs.
void leds_toggle(unsigned char ledv); // Invierte un conjunto de LEDs.
void leds_invert(unsigned char ledv); // Invierte un conjunto de LEDs.
```

Cada LED tiene un identificador, aquí los distintos identificadores para los LEDs verde, amarillo, rojo y todos:

```
#define LEDS_GREEN                  1
#define LEDS_YELLOW                 2
#define LEDS_RED                    4
#define LEDS_ALL                    7
```

4.8 Radio Duty Cycling

El consumo energético es muy importante para los sensores inalámbricos, para lograr perdurar bastante tiempo antes de tener que cambiarles las pilas o baterías. Para conseguirlo no basta con contar con un transmisor de bajo consumo. El hardware no es suficiente para reducir el consumo de permanecer constantemente a la escucha de nuevo tráfico de red.

En el caso del sensor que se va a usar en las pruebas, el gasto de estar siempre a la escucha es de 60 mW, y un poco mayor al transmitir. Esto hace que las pilas del sensor se agoten en cuestión de días.

Para evitarlo el transmisor debe apagarse todo lo posible. El problema es que cuando está apagado no le es posible enviar o recibir nuevos mensajes. Así que el transmisor deberá buscar una forma de recivir los mensajes pero mantener la radio apagada entre recepción y transmisión de mensajes.

El propósito de un protocolo de ahorro energético en el ciclo de trabajo es mantener la radio apagada entre los puntos donde las radios transmiten o reciben. De forma que si los nodos están sincronizados, el ciclo de trabajo puede programar los momentos en los que se debe transmitir y estar a la escucha. Si no existe sincronización, los nodos deben buscar otra manera de proporcionar momentos para comunicarse.

En el ámbito de las redes de sensores, los protocolos de ciclo de trabajo se llaman también protocolos de control de acceso al medio (MAC).

En Contiki OS se ha separado el ciclo de trabajo del control de acceso al medio. Y por ello se tiene la capa RDC antes de la capa MAC. Se tienen 3 métodos para gestionar el ciclo de trabajo de la radio de comunicaciones 802.15.4: ContikiMAC, X-MAC y LPP.

El más usado es ContikiMAC. Para que ContikiMAC funcione se deben satisfacer las siguientes restricciones. La primera, que el intervalo entre paquetes de transmisión (t_i) debe ser menor que el intervalo en el que se comprueba el uso del canal (t_c). De esta forma nos aseguramos que si no es en la primera comprobación de canal, sea en la segunda cuando el nodo receptor reciba el comienzo de la transmisión. Si el intervalo de comprobación (t_c) es menor que t_i, dos comprobaciones de canal no podrían asegurar que se haya transmitido un paquete. A la inversa, el tiempo de retardo (t_r) dos veces y el tiempo de comprobación del canal deben ser menor que el tiempo de transmisión del paquete más corto posible (t_s).

$$t_r + t_c + t_r < t_s \tag{4–1}$$

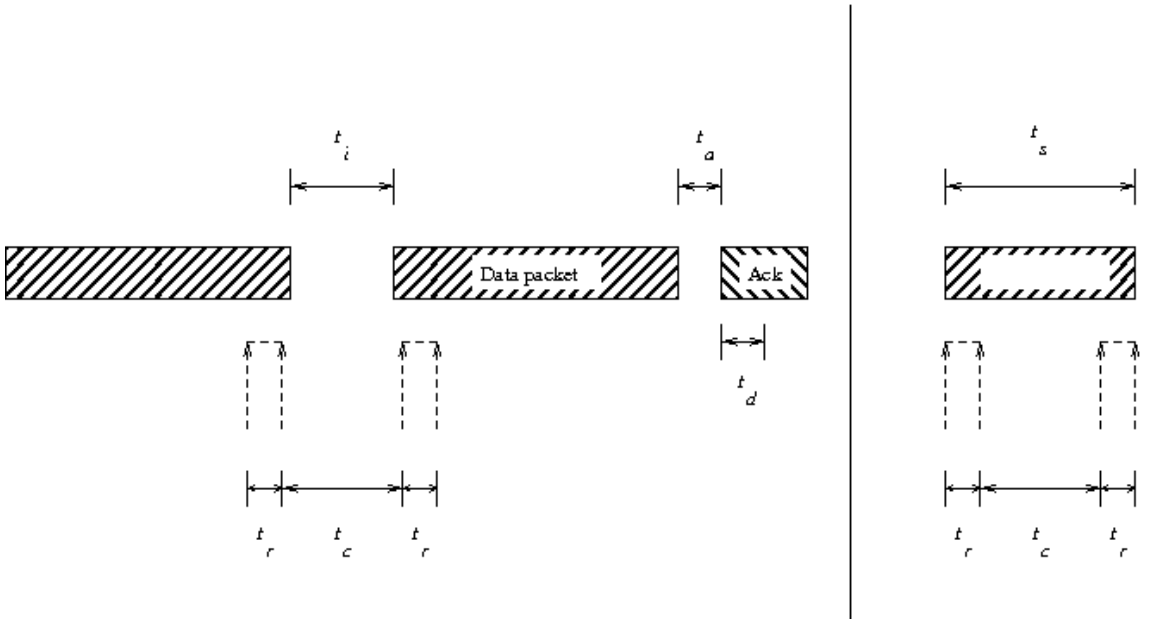

Figura 4-7. Ejemplo de transmisión de paquetes

Cuando se detecta una transmisión entrante, ContikiMAC mantiene la radio encendida para recibir el paquete entero. Cuando se recibe el paquete entero, se transmite un asentimiento a nivel de la capa de enlace.

El tiempo que tarda el asentimiento en transmitirse (t_a) y el tiempo de retraso hasta que se detecta el asentimiento (t_d) establecen la cota inferior para el intervalo de comprobación (t_c).

De forma que las restricciones de tiempo de ContikiMAC se pueden resumir así:

$$t_a + t_d < t_i \tag{4–2}$$

$$t_i < t_c \tag{4–3}$$

$$t_c + 2t_r < t_s \tag{4–4}$$

El problema de implementar los protocolos de ciclo de trabajo y control de acceso al medio de Contiki es que aumentan el tamaño del programa. En mi caso al utilizarlos, el programa no entraba en la ROM del dispositivo. Por ello tuve que desactivar ContikiMAC y cualquier protocolo MAC ó RDC. De esta forma en nuestras pruebas siempre estará encendida la radio. De todas formas, esto no afectará a nuestras pruebas como ya se explicará posteriormente.

4.9 Pila de red

Una de las características más importantes de Contiki OS es el soporte de los protocolos IP. Y eso lo hace ser uno de los primeros sistemas operativos para sistemas embebidos que proporcionan soporte para IPv6. Las pilas de comunicaciones IP que trae Contiki funcionan tanto con IPv4 como con IPv6.

uIP, que será la que usemos en este trabajo, cuenta con el sello "IPv6 Ready", que garantiza que la implementación del protocolo es correcta y cumple con el estándar. Esta pila fue lanzada en 2001 por el Swedish Institute of Computer Science (SICS) y está licenciada bajo licencia BSD. Está pensanda para microcontroladores de 8 y 16 bits. La otra pila de comunicaciones es Rime. No está basada en IP y proporciona funciones para las comunicaciones radio. Es por eso que elegimos uIP como pila de comunicaciones para este trabajo.

uIP es de código abierto. A pesar de ser pequeño y simple, uIP no requiere que alguno de los extremos de la comunicación tengan complejas pilas. Sino que permite la comunicación entre extremos que usan pilas ligeras a la vez. El código sólo requiere unos pocos kilobytes de memoria de programa y el uso de la RAM se puede reducir a unos pocos cientos de bytes.

Para habilitar el uso de IPv6 en la aplicación que desarrollemos, debemos incluir en su fichero `project-conf.h` la siguiente línea:

```
#define UIP_CONF_IPV6  1
```

Luego podemos configurar otros parámetros como número máximo de vecinos en la tabla de vecindades. O el máximo de rutas que almacenaremos en la tabla de enrutamiento:

```
#ifndef NBR_TABLE_CONF_MAX_NEIGHBORS
#define NBR_TABLE_CONF_MAX_NEIGHBORS        20
#endif
#ifndef UIP_CONF_MAX_ROUTES
#define UIP_CONF_MAX_ROUTES                 20
#endif
```

Otras posibles configuraciones son el tamaño del buffer para paquetes entrantes, si hacemos cola por cada vecino durante la resolución de direcciones, si miramos que los paquetes IPv6 sean consistentes, si usamos fragmentación en los paquetes IPv6:

```
#ifndef UIP_CONF_BUFFER_SIZE
#define UIP_CONF_BUFFER_SIZE              1300
#endif

#define UIP_CONF_IPV6_QUEUE_PKT             0
#define UIP_CONF_IPV6_CHECKS                1
#define UIP_CONF_IPV6_REASSEMBLY            0
#define UIP_CONF_MAX_LISTENPORTS            8
```

5 DESARROLLO DEL ENTORNO DE PRUEBAS

Primero se establecerán las pruebas a realizar. Qué cosas se van a evaluar y cómo las vamos a medir. Y luego se explicará paso por paso cómo se desarrollaron. Como ya se ha visto en el capítulo anterior, el lenguaje de programación para el desarrollo en Contiki es C. Así que fue éste el lenguaje que usé para desarrollar los programas que llevarán los sensores.

5.1 Pruebas

Se evaluarán los distintos mecanismos de seguridad y autenticación para redes 802.15.4 con sensores que funcionan con el sistema operativo Contiki. En las pruebas valoraremos tres aspectos de cada uno de los mecanismos, repitiendo siempre las pruebas en el mismo escenario:

- **Consumo energético**: Aquí nos fijaremos en el impacto que tiene cada mecanismo en la batería de los dispositivos. Éste resulta un aspecto crítico, ya que las aplicaciones en las que se piensan usar estas soluciones de seguridad han de ser de consumo muy bajo para que la vida de la pila o batería dure lo máximo posible.
- **Tamaño del programa**: La memoria Flash y RAM de los dispositivos resulta muy limitada. En el caso del Tmote Sky tenemos 48k de memoria Flash y 10k de RAM. Por tanto mediremos cuánto se incrementa el tamaño del programa con las distintas medidas de autenticación y seguridad.
- **Retardo en las comunicaciones**: El hecho de aplicar estas medidas conllevará un trabajo extra por parte de los sensores al enviar o recibir datos. Tendrán que cifrar, añadir códigos de autenticación al mensaje, descifrar y autenticar. Por tanto mediremos los distintos retardos.

5.2 Entorno de pruebas

Ahora que ya se han definido los distintos aspectos a evaluar en las pruebas, pasaré a definir el entorno en el que se llevarán a cabo las pruebas. En mi caso me decanté por una solución sencilla en la que sólo hubiera dos dispositivos, de manera que no hubiera otros que interfieran en lo que queremos medir. Estos dos dispositivos serán Tmote Sky. Uno de ellos se encargará de enviar paquetes UDP unicast al otro. Mientras tanto el otro dispositivo estará a la escucha y recibirá los paquetes y los registrará. De esta forma serán necesarios dos programas. Uno para el dispositivo emisor (`udp-sender.c`) y otro para el receptor (`udp-sink.c`).

Por otro lado, aunque las pruebas se vayan a realizar para el caso de dos dispositivos. El desarrollo del entorno de pruebas ha de ser escalable. De esta forma en vez de establecerse una dirección destino para los paquetes unicast, habrá una fase previa en la que el dispositivo receptor anuncie por multicast su servicio como sumidero de paquetes unicast y el o los dispositivos emisores busquen el servicio y descubran así a qué

dirección IP destino tendrán que enviar sus mensajes unicast. Esto se hará a través de una aplicación que trae Contiki que se llama `servreg-hack`.

Esto no será suficiente para poder medir el consumo energético. De forma que paralelamente al proceso de envío en el caso del emisor y recepción de paquetes unicast en el caso del receptor, se tendrá un proceso que irá recopilando datos sobre el consumo del dispositivo emisor y los enviará al dispositivo receptor. El dispositivo receptor podrá mostrarlos a través del puerto serie a la herramienta "Collect View" del simulador Cooja como se verá en el siguiente capítulo. En la figura 5-1 se tiene un diagrama paso de mensajes para intentar explicar más claramente cómo será el funcionamiento del escenario de pruebas.

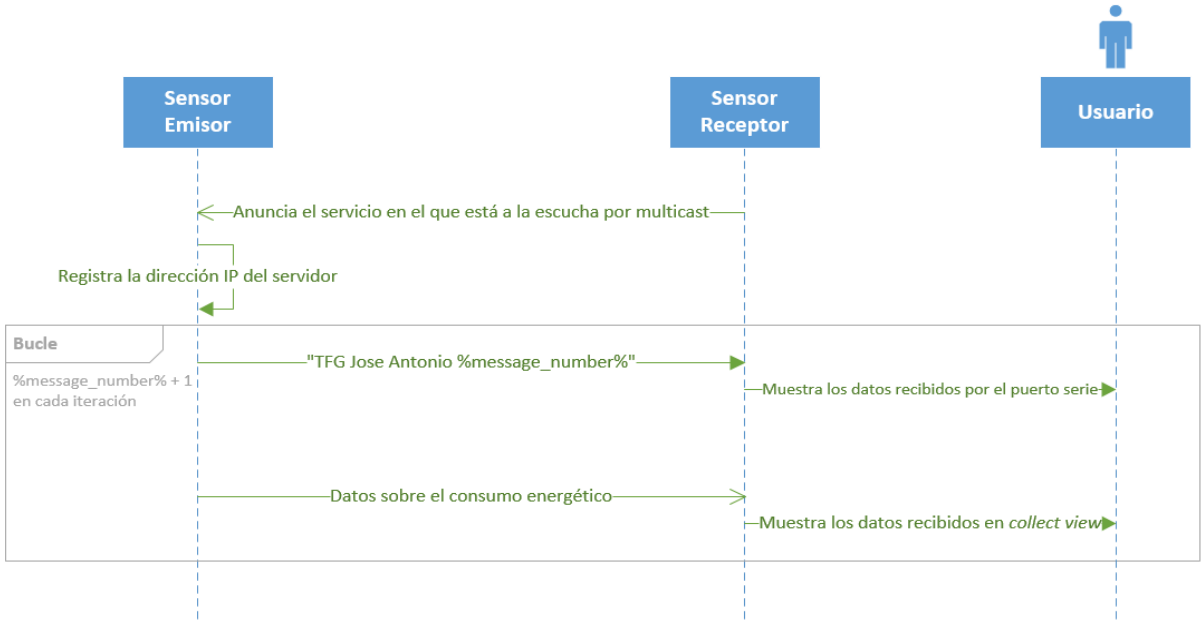

Figura 5-1. Diagrama paso de mensajes del escenario de pruebas

5.3 udp-sender.c

Este fichero contendrá el programa que vamos a usar en el dispositivo emisor. Primero se ha de configurar el dispositivo para que esté en la misma red que el receptor. Despues será necesario por un lado enviar los mensajes UDP unicast. Pero por otro lado también se necesitará recolectar los datos del consumo energético y registrar de algún modo el retardo en las comunicaciones.

5.3.1 Configuración del dispositivo

En este apartado explicaré cómo es la fase de configuración del dispositivo. En el caso del proceso `unicast_sender_process` que se encarga del envío de mensajes unicast al dispositivo receptor, lo primero será empezar la aplicación `servreg-hack` que será el mecanismo que usemos para descubrir su dirección. De esta forma nuestro dispositivo registrará los anuncios de servicios que hagan otros equipos que usen esta aplicación. Esta aplicación también se usará en el receptor. Los anuncios de servicios se hacen por multicast. Se comienza a registrar los distintos anuncios de la siguiente forma:

```
349   /*
350   * Starts the servreg_hack app that is used to discover the
    sink
351   * service server address
352   */
353   servreg_hack_init();
```

Después estableceremos la dirección IPv6 de nuestro dispositivo y la del dispositivo receptor. La del dispositivo receptor es la que usaremos para enviar las estadísticas para verlas en la aplicación “Collection View”. Como hemos dicho antes, la del servidor receptor de paquetes unicast se obtendrá por medio de la aplicación `servreg-hack`. Que en este caso coinciden y son la misma, pero podrían no serlo en el caso de que el dispositivo recolector de estadísticas no fuera el mismo que el receptor de paquetes unicast. Para establecer entonces las direcciones usamos la función `set_global_address()`:

```
355   /*
356   * sets the IPv6 addresses of the device and the Collection
    View
357   * server
358   */
359   set_global_address();
```

Esta función por dentro va haciendo lo siguiente:

```
260 /*
261 * set_global_address: sets the IPv6 addresses of the device
262 * and the Collection View server
263 */
264 static void set_global_address(void)
265 {
266   uip_ipaddr_t ipaddr;
267
268
269   /* Build the IPv6 Address */
270   uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0,
    0);
271   /* Set the last 64 bits based on the MAC Address */
272   uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
273   /* Add the unicast address to the list */
274   uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
275
276   /* Set the Collection View server address */
277   uip_ip6addr(&server_ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0,
    0, 0, 0, 1);
278
279 }
```

Primero construye una dirección IPv6 a partir de ocho enteros de 16 bits con la función `uip_ip6addr()`. En nuestro caso tomaremos el que Contiki asigna por defecto, que es `0xfd00`. Luego establece el valor de los últimos 64 bits de la dirección a partir de la MAC del dispositivo con la función

`uip_ds6_set_addr_iid()`. Una vez montada la dirección la añadimos a la lista de direcciones IPv6 del dispositivo con `uip_ds6_addr_add()` el valor 0 hace referencia al tiempo que esta dirección será valida. Seleccionamos 0 para que siga válida hasta que no se indique lo contrario. Y el valor `ADDR_AUTOCONF` es para indicar la naturaleza de la dirección. En este caso se ha obtenido por autoconfiguración. Este valor se usa en el caso de que haya varias direcciones asignadas a un dispositivo. Contiki elegirá la dirección a usar al enviar un paquete entre otras cosas en función de cómo se han obtenido las direcciones.

En el proceso de envío de paquetes unicast `unicast_sender_process` lo siguiente sería registrar el socket UDP que queremos usar para posteriormente enviar los paquetes unicast al receptor. El conjunto de funciones `simple_udp` nos facilitan bastante la comunicación. Este módulo provée una API más simple que la que Contiki tiene por defecto para el manejo de paquetes UDP. En concreto la función `simple_udp_register()` nos permite aceptar y establecer con qué puerto enviaremos los paquetes según los siguientes parámetros:

- Conexión: Es el puntero a la estructura `simple_udp_connection`.
- Puerto local: Se refiere al puerto local UDP en el que se espera recibir los paquetes UDP entrantes. En este caso usaremos el puerto `UDP_PORT` que hemos definido como `1234`.
- Dirección remota: Se trata de la dirección desde la que se espera recibir los paquetes. Será NULL para indicar que aceptaremos paquetes desde cualquier dirección IPv6.
- Puerto remoto: El puerto desde el que se espera recibir los paquetes entrantes.
- Función callback: Un puntero a la función que se llamará cuando se reciban los paquetes.

La función de callback no era estrictamente necesaria en el escenario final. Pero como se pretendía montar un entorno de pruebas de autenticación y seguridad, dejamos esta función por si se quisiera hacer la prueba de forma que el receptor también interactúe con el emisor.

En cualquier caso se deja la función callback receiver() por si se quisiera implementar otra funcionalidad, en nuestro escenario sólo se imprimirían los datos recibidos por el emisor. Pero se podría usar por ejemplo para recibir asentimientos del receptor y contabilizarlos para un entorno en el que haya muchas pérdidas:

```
319  /*
320  * receiver: callback function to handle incoming udp packets
     from
321  * the sink server. It is not really used in this scenario
322  */
323  static void receiver(struct simple_udp_connection *c,
324           const uip_ipaddr_t *sender_addr,
325           uint16_t sender_port,
326           const uip_ipaddr_t *receiver_addr,
327           uint16_t receiver_port,
328           const uint8_t *data,
329           uint16_t datalen)
330  {
331    printf("Data received on port %d from port %d with length
     %d\n",
332           receiver_port, sender_port, datalen);
333  }
```

A partir de ahí tendremos configurado el proceso `unicast_sender_process` aunque estableceremos un tiempo que esperaremos antes de empezar a enviar los mensajes al receptor.

```
369   /* We set the timer to SEND_INTERVAL */
370   etimer_set(&periodic_timer, SEND_INTERVAL);
```

Este tiempo `SEND_INTERVAL` será en nuestro caso 10 segundos. Y además será el mismo tiempo que después emplearemos para decidir cuándo volvemos a enviar otro mensaje. Para definir los 10 segundos definí la variable `SEND_INTERVAL` de la siguiente forma:

```
71  /*
72  * Time interval to send another message
73  */
74  #define SEND_INTERVAL         (10 * CLOCK_SECOND)
```

En el proceso collection_view_client_process la parte de configuración es breve, pues sólo estableceremos el socket UDP con el servidor "Collection View" del receptor:

```
296   /* new connection with remote host */
297   client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
298   udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

Los puertos UDP usados para la comunicación serán el `5688` en el caso del servidor y el puerto `8775` en el caso del cliente. A partir de entonces la comunicación la gestionará la app de `powertrace` y `collect-view`. Hasta aquí la fase de configuración del programa del emisor `udp-sender.c`.

5.3.2 Envío de la información

Una vez ya hemos configurado el dispositivo emisor, pasaremos a describir cómo funciona el envío de la información al receptor. En el proceso `unicast_sender_process` de envío de paquetes UDP unicast al receptor habrá un bucle que se encargue de ir enviando los paquetes cada cierto intervalo de tiempo:

```
372   while(1) {
373
374     /*
375     * Wait for the interval to start sending the message, but
    before
376     * reset it again so that in the next interval we cand send
    another
377     * message
378     */
379     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
380     etimer_reset(&periodic_timer);
381
382     /* It looks for the server address */
383     addr = servreg_hack_lookup(SERVICE_ID);
384     if(addr != NULL) {
385       static unsigned int message_number;
386       char buf[24];
387
```

```
388         #if DEBUG_SENDER == 1
389         printf("Sending unicast to ");
390         uip_debug_ipaddr_print(addr);
391         printf("\n");
392         #endif
393 
394         /* The message to send with the sequence number */
395         sprintf(buf, "TFG Jose Antonio %d", message_number);
396 
397         /* If we reach the max sequence number we reset it */
398         if (message_number == MAX_MESSAGE_NUMBER) {
399           message_number = 0;
400         }
401         else {
402           message_number++;
403         }
404 
405         /* It sends it to the sink server */
406         simple_udp_sendto(&unicast_connection, buf, strlen(buf) +
    1, addr);
407 
408       } else {
409         #if DEBUG_SENDER == 1
410         printf("Service %d not found\n", SERVICE_ID);
411         #endif
412       }
413     }
414 
415     PROCESS_END();
416 }
```

Lo primero es esperar a que acabe el temporizador. Este temporizador ya lo establecimos a 10 segundos en la fase de configuración. Una vez acabado lo reseteamos para que cuando vuelvan a pasar 10 segundos volvamos a enviar:

```
374       /*
375       * Wait for the interval to start sending the message, but
    before
376       * reset it again so that in the next interval we cand send
    another
377       * message
378       */
379       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
380       etimer_reset(&periodic_timer);
```

Lo siguiente será averiguar la dirección del receptor a través de nuestro registro de servicios de la app `servreg-hack`. Buscamos por el `SERVICE_ID` que hemos asignado al servicio del sumidero de paquetes. En nuestro caso `SERVICE_ID` es `190`. Y así obtenemos la dirección:

```
382       /* It looks for the server address */
383       addr = servreg_hack_lookup(SERVICE_ID);
```

Después se escribe en el buffer de salida la cadena que queremos enviar. En nuestro caso será "TFG Jose Antonio" más un número de secuencia :

```
394         /* The message to send with the sequence number */
395         sprintf(buf, "TFG Jose Antonio %d", message_number);
```

Para saber el número máximo de número de secuencia compilé un programa en un sensor TMote Sky que incluía una línea con:

```
405         printf("Size of unsigned int %d\n", sizeof(unsigned int));
```

La salida del sensor fue la de la figura 5-2.

```
00:01.209  ID:3   Size of unsigned int 2
```

Figura 5-2. Tamaño de un `unsigned int` en el TMote Sky

Por tanto al ser 2 bytes, el límite será `65535`. Lo comprobamos con la siguiente línea:

```
404         unsigned int prueba = 0;
405         printf("unsigned int limit %u\n", prueba-1);
```

Que dan por resultado la salida del sensor que vemos en la figura 5-3.

```
00:01.212  ID:4   unsigned int limit 65535
```

Figura 5-3. Límite de tamaño de un `unsigned int` en el TMote Sky

Es importante tener controlada esta situación para evitar casos de *overflow*. De esta forma el tamaño máximo de la cadena de texto que vamos a enviar tendrá 24 caractéres. 23 caracteres para el texto en el caso del máximo número de secuencia y 1 carácter para el final de la cadena `\0`. De esta forma cada vez que escribimos el mensaje comprobamos justo después que no se haya llegado al número máximo de mensajes. Si hemos llegado reseteamos el número de secuencia, si no pues aumentamos el número de secuencia.

```
397         /* If we reach the max sequence number we reset it */
```

```
398         if (message_number == MAX_MESSAGE_NUMBER) {
399           message_number = 0;
400         }
401         else {
402           message_number++;
403         }
```

Una vez hecho esto, se envía por UDP al dispositivo receptor a través de la función `simple_udp_sendto()` con el socket UDP que registramos en la fase de configuración.

```
405     /* It sends it to the sink server */
406     simple_udp_sendto(&unicast_connection, buf, strlen(buf) + 1,
    addr);
```

Y una vez enviado volveríamos al comienzo del bucle donde esperaríamos a que expirara el temporizador y volveríamos a realizar todo este proceso hasta enviar el siguiente paquete. Esto es todo lo que se hace en el proceso `unicast_sender_process`.

En cambio en el proceso collection_view_client_process sólo esperaremos recibir paquetes entrantes, aunque en nuestro caso los ignoraremos.

```
307   while(1) {
308     PROCESS_YIELD();
309     if(ev == tcpip_event) {
310       tcpip_handler();
311     }
312   }
```

La función `tcpip_handler()` no hace nada:

```
143 static void tcpip_handler(void)
144 {
145   if(uip_newdata()) {
146     /* The sender ignores incoming data */
147   }
148 }
```

Son las apps de Contiki `powertrace` y `collect-view` las que se encargan de recoger la información del consumo y enviarla al receptor. Nosotros hemos definido la función `collect_common_send()` que se requiere para el funcionamiento de la app `collect-view`. En esta función se recopila la información para enviar al sensor.

```
152 /*
153 * collect_common_send: Send the statistics to the Collect View
    server
154 */
155 void collect_common_send(void)
```

```
156 {
157   static uint8_t seqno;
158   struct {
159     uint8_t seqno;
160     uint8_t for_alignment;
161     struct collect_view_data_msg msg;
162   } msg;
163 
164   uint16_t parent_etx;
165   uint16_t rtmetric;
166   uint16_t num_neighbors;
167   uint16_t beacon_interval;
168   rpl_parent_t *preferred_parent;
169   linkaddr_t parent;
170   rpl_dag_t *dag;
171 
172   if(client_conn == NULL) {
173     /* Not setup yet */
174     return;
175   }
176   memset(&msg, 0, sizeof(msg));
177   seqno++;
178   if(seqno == 0) {
179     /* Wrap to 128 to identify restarts */
180     seqno = 128;
181   }
182   msg.seqno = seqno;
183 
184   linkaddr_copy(&parent, &linkaddr_null);
185   parent_etx = 0;
186 
187   /* Let's suppose we have only one instance */
188   dag = rpl_get_any_dag();
189   if(dag != NULL) {
190     preferred_parent = dag->preferred_parent;
191     if(preferred_parent != NULL) {
192       uip_ds6_nbr_t *nbr;
193       nbr =
    uip_ds6_nbr_lookup(rpl_get_parent_ipaddr(preferred_parent));
194       if(nbr != NULL) {
195         /* Use parts of the IPv6 address as the parent address,
    in reversed byte order. */
196         parent.u8[LINKADDR_SIZE - 1] = nbr-
    >ipaddr.u8[sizeof(uip_ipaddr_t) - 2];
197         parent.u8[LINKADDR_SIZE - 2] = nbr-
    >ipaddr.u8[sizeof(uip_ipaddr_t) - 1];
198         parent_etx = rpl_get_parent_rank((uip_lladdr_t *)
    uip_ds6_nbr_get_ll(nbr)) / 2;
199       }
200     }
201     rtmetric = dag->rank;
202     beacon_interval = (uint16_t) ((2L << dag->instance-
    >dio_intcurrent) / 1000);
203     num_neighbors = uip_ds6_nbr_num();
204   } else {
205     rtmetric = 0;
```

```
206       beacon_interval = 0;
207       num_neighbors = 0;
208     }
209
210     collect_view_construct_message(&msg.msg, &parent,
211                                    parent_etx, rtmetric,
212                                    num_neighbors,
    beacon_interval);
213
214     uip_udp_packet_sendto(client_conn, &msg, sizeof(msg),
215                           &server_ipaddr,
    UIP_HTONS(UDP_SERVER_PORT));
216   }
```

Esta función la obtuve de uno de los ejemplos de funcionamiento de Contiki (`rpl-collect`). Hay que destacar la función `collect_view_construct_message()`. Con ella construimos el mensaje, y lo enviamos con `uip_udp_packet_sendto()` pasando el socket UDP (`client_conn`), la dirección del servidor (`server_ipaddr`) y el puerto en el que escucha como parámetro (`UDP_SERVER_PORT`) además del mensaje.

```
214     uip_udp_packet_sendto(client_conn, &msg, sizeof(msg),
215                           &server_ipaddr, HTONS(UDP_SERVER_PORT));
```

En nuestro caso el puerto UDP del servidor será el `5688`. Y esto es todo en la parte del programa del sensor emisor.

5.4 udp-sink.c

Este fichero contendrá el programa que vamos a usar en el dispositivo receptor.

5.4.1 Configuración del dispositivo

Lo primero será declarar los procesos y hacer que comiencen, además del proceso collect_common_process(), que es parte de la app Collect-View, y vale tanto para el sensor recolector como para el sensor emisor:

```
93  /*
94  *
95  * collection_view_server_process: will receive the statistics of
    the
96  * sender sensor
97  *
98  * unicast_receiver_process: will receive the unicast packets to
    the sink
99  *
```

```
100 */
101 PROCESS(unicast_receiver_process, "Unicast receiver example
    process");
102 PROCESS(collection_view_server_process, "UDP server process");
103 AUTOSTART_PROCESSES(&unicast_receiver_process,
    &collection_view_server_process, &collect_common_process);
```

En el proceso receptor de mensajes, se ha de configurar el dispositivo para que esté en la misma red que el emisor. Después será necesario por un lado iniciar el servicio `servreg-hack` para anunciar que tenemos un servicio de recepción de mensajes UDP unicast.

```
293   servreg_hack_init();
294
295   ipaddr = set_global_address();
296
297   servreg_hack_register(SERVICE_ID, ipaddr);
```

El identificador del servicio `SERVICE_ID` será el que vimos en la parte del emisor, `190`. Y en la función `set_global_address()` asignamos la dirección IPv6 del dispositivo de la misma forma que ya hicimos en el caso del emisor, a partir de su dirección MAC. Aunque esta vez no asignaremos la dirección del servidor, pues somos nosotros:

```
196 /*
197 * set_global_address: sets the IPv6 addresses of the device
198 */
199 static uip_ipaddr_t * set_global_address(void)
200 {
201   static uip_ipaddr_t ipaddr;
202
203   // Build the IPv6 Address
204   uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0,
    0);
205   // Set the last 64 bits based on the MAC Address
206   uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
207   // We add the unicast address to the list
208   uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
209
210   return &ipaddr;
211 }
```

De esta forma se empezará a anunciar el servicio por multicast gracias a la app `servreg-hack`. En la figura 5-5 podemos ver el contenido de uno de los paquetes de anuncio. En él se puede ver el `SERVICE_ID 190` que en hexadecimal es `0xBE`.

Source	Destination	Protocol	Length	Info
fe80::212:7401:1:101	ff02::1	UDP	50	61616 → 61616 Len=26

Figura 5-4. Información del paquete de anuncio de servicio multicast de la app servreg-hack

```
0000  41 d8 02 cd ab ff ff 01  01 01 00 01 74 12 00 7e   A....... ....t..~
0010  3b 01 f3 00 76 9e 01 00  fd 00 00 00 00 00 00 00   ;...v... ........
0020  02 12 74 01 00 01 01 01  be 00 00 01 58 52 78 cb   ..t..... ....XRx.
0030  85 cc dd 60 f5 2f                                  ...`./
```

Figura 5-5. Contenido de un paquete multicast de la app servreg-hack que anuncia el servicio con `SERVICE_ID 190`

Después registramos el socket de escucha para escuchar las peticiones que se hagan en el puerto `UDP_PORT` (1234). Le pasamos como parámetro la dirección de la estructura que tiene los datos de la comunicación UDP unicast. La dirección remota la registraremos como NULL para que se acepten paquetes de cualquier dirección. De esta forma si en vez de uno, tuviéramos varios dispositivos, el servidor receptor seguiría recibiendo los paquetes y registrándolos. Para registrar los paquetes que llegan, se pasa como parámetro la función callback que se llamará cada vez que llegue un paquete en el puerto 1234 paquete.

```
299   simple_udp_register(&unicast_connection, UDP_PORT,
300                       NULL, UDP_PORT, receiver);
```

Por otro lado, en el proceso de recolección de los datos del consumo energético por parte de la aplicación `Collect-View` tenemos que configurar la dirección y puerto en los que escucharemos. Además de configurar el dispositivo como `root` para el protocolo RPL.

```
231 PROCESS_THREAD(collection_view_server_process, ev, data)
232 {
233   uip_ipaddr_t ipaddr;
234   struct uip_ds6_addr *root_if;
235
236   PROCESS_BEGIN();
237
238   PROCESS_PAUSE();
239
240   #if DEBUG_SINK == 1
241   PRINTF("UDP server started\n");
242   #endif
243
244 #if UIP_CONF_ROUTER
```

```
245   uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0,
      1);
246   /* uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr); */
247   uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
248   root_if = uip_ds6_addr_lookup(&ipaddr);
249   if(root_if != NULL) {
250     rpl_dag_t *dag;
251     dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t
      *)&ipaddr);
252     uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0,
      0, 0);
253     rpl_set_prefix(dag, &ipaddr, 64);
254     PRINTF("created a new RPL dag\n");
255   } else {
256     PRINTF("failed to create a new RPL DAG\n");
257   }
258 #endif /* UIP_CONF_ROUTER */
259
260   print_local_addresses();
261
262   server_conn = udp_new(NULL,
      UIP_HTONS(UDP_COLLECT_CLIENT_PORT), NULL);
263   udp_bind(server_conn, UIP_HTONS(UDP_COLLECT_SERVER_PORT));
264
265   #if DEBUG_SINK == 1
266   PRINTF("Created a server connection with remote address ");
267   PRINT6ADDR(&server_conn->ripaddr);
268   PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn-
      >lport),
269          UIP_HTONS(server_conn->rport));
270   #endif
```

El puerto UDP en el que el servidor `Collect-View` escuchará será el 5688. Y el puerto UDP remoto del que espera recibir los datos 8775.

```
76  #define UDP_COLLECT_CLIENT_PORT 8775
77  #define UDP_COLLECT_SERVER_PORT 5688
```

```
262   server_conn = udp_new(NULL,
      UIP_HTONS(UDP_COLLECT_CLIENT_PORT), NULL);
263   udp_bind(server_conn, UIP_HTONS(UDP_COLLECT_SERVER_PORT));
```

También habrá que conectar la app `Collect-View` a la salida del puerto serie para poder visualizar las estadísticas en la herramienta del mismo nombre `Collect-View`. Esta herramienta se ejecuta tanto en escenarios reales como en los nuestros de prueba en un PC.

```
128 /*
129 * collect_common_net_init: sets the serial line input for
    Collection View
130 */
```

```
131 void collect_common_net_init(void)
132 {
133 #if CONTIKI_TARGET_Z1
134   uart0_set_input(serial_line_input_byte);
135 #else
136   uart1_set_input(serial_line_input_byte);
137 #endif
138   serial_line_init();
139 }
```

5.4.2 Recepción de información

En el proceso principal tenemos al proceso esperando eventos, pues ya establecimos antes la función callback que se ejecutará cuando se reciba un paquete entrante en el puerto 1234.

```
302   while(1) {
303     PROCESS_WAIT_EVENT();
304   }
305   PROCESS_END();
306 }
```

La función callback `receiver()` registra los paquetes según van llegando. Va imprimiendo por la salida estándar, que en nuestro caso será el puerto serie, la dirección y el puerto desde los que se reciben los datos, además de la longitud de los datos del mensaje y el mensaje en sí. El formato es: `"Data received from IP_ADDR on port 1234 from port 1234 with length MSG_LENGTH: 'TFG Jose Antonio SEQ_NUM" "`. A continuación vemos el código que corresponde a esta función.

```
215 static void receiver(struct simple_udp_connection *c,
216         const uip_ipaddr_t *sender_addr,
217         uint16_t sender_port,
218         const uip_ipaddr_t *receiver_addr,
219         uint16_t receiver_port,
220         const uint8_t *data,
221         uint16_t datalen)
222 {
223   printf("Data received from ");
224   uip_debug_ipaddr_print(sender_addr);
225   printf(" on port %d from port %d with length %d: '%s'\n",
226         receiver_port, sender_port, datalen, data);
227 }
```

En el caso del proceso de recepción de la información del consumo energético del sensor emisor tenemos la función callback `tcpip_handler()`, que se encargará de tratar los paquetes que lleguen al servidor `Collect-View`. El proceso estará constantemente a la espera de nuevos eventos y llamará a la función en caso de que llegue un paquete. La llamamos de la siguiente forma:

```
272   while(1) {
```

```
273       PROCESS_YIELD();
274       if(ev == tcpip_event) {
275         tcpip_handler();
276       } else if (ev == sensors_event && data == &button_sensor) {
277         PRINTF("Initiaing global repair\n");
278         rpl_repair_root(RPL_DEFAULT_INSTANCE);
279       }
280     }
281
282     PROCESS_END();
283   }
```

Y en la función se entregará la información a la app de `Collect-View` a través de la función `collect_common_recv()`.

```
143   /*
144   * tcpip_handler: handles the incoming packet from the sender
      sensor
145   * and gathers the information for the Collection View app
146   */
147   static void tcpip_handler(void)
148   {
149     uint8_t *appdata;
150     linkaddr_t sender;
151     uint8_t seqno;
152     uint8_t hops;
153
154     if(uip_newdata()) {
155       appdata = (uint8_t *)uip_appdata;
156       sender.u8[0] = UIP_IP_BUF->srcipaddr.u8[15];
157       sender.u8[1] = UIP_IP_BUF->srcipaddr.u8[14];
158       seqno = *appdata;
159       hops = uip_ds6_if.cur_hop_limit - UIP_IP_BUF->ttl + 1;
160       collect_common_recv(&sender, seqno, hops,
161                           appdata + 2, uip_datalen() - 2);
162     }
163   }
```

Y esto es todo por parte del sensor receptor.

5.5 project-conf.h

El fichero `project-conf.h` se usa para configurar las diferentes opciones de Contiki. Podemos decidir qué módulos vamos a usar y cuáles no. De esta forma se permite ahorrar memoria ROM y RAM que de otra forma íbamos a gastar sin usarla.

Lo primero que haremos será reducir el máximo número de vecinos y rutas, pues en nuestro escenario sólo tendremos 2 dispositivos.

```
35  #undef NBR_TABLE_CONF_MAX_NEIGHBORS
```

```
36 #undef UIP_CONF_MAX_ROUTES
37
38 #ifdef TEST_MORE_ROUTES
39 /* configure number of neighbors and routes */
40 #define NBR_TABLE_CONF_MAX_NEIGHBORS     2
41 #define UIP_CONF_MAX_ROUTES   2
42 #endif /* TEST_MORE_ROUTES */
```

Después seleccionamos los drivers `nullrdc` y `nullmac` para las capas de *Radio Duty Cycling* y control de acceso al medio (MAC) respectivamente. Con esto reduciremos mucho la memoria del programa, que en el dispositivo TMote-Sky resulta bastante escasa.

```
45 #undef NETSTACK_CONF_RDC
46 #define NETSTACK_CONF_RDC     nullrdc_driver
47
48 #undef NETSTACK_CONF_MAC
49 #define NETSTACK_CONF_MAC nullmac_driver
```

A continuación establecemos la que será nuestra clave a la hora de cifrar tanto los paquetes enteros como los códigos de autenticación de los mensajes.

```
51 #undef NONCORESEC_KEY
52 #define NONCORESEC_KEY { 0x00 , 0x01 , 0x02 , 0x03 , \
53                          0x04 , 0x05 , 0x06 , 0x07 , \
54                          0x08 , 0x09 , 0x0A , 0x0B , \
55                          0x0C , 0x0D , 0x0E , 0x0F }
```

Y establecemos el nivel de seguridad que usaremos en las comunicaciones. Podemos distinguir entre comunicaciones unicast o difusión. El valor de `ADAPTIVESEC_CONF_UNICAST_SEC_LVL` y `ADAPTIVESEC_CONF_BROADCAST_SEC_LVL` puede configurarse del 1 al 7. Cada valor corresponde a las medidas de autenticación y/o cifrado de la siguiente tabla.

```
57 #undef ADAPTIVESEC_CONF_UNICAST_SEC_LVL
58 #define ADAPTIVESEC_CONF_UNICAST_SEC_LVL 1
59 #undef ADAPTIVESEC_CONF_BROADCAST_SEC_LVL
60 #define ADAPTIVESEC_CONF_BROADCAST_SEC_LVL 1
61 #undef LLSEC802154_CONF_USES_AUX_HEADER
62 #define LLSEC802154_CONF_USES_AUX_HEADER 0
63 #undef NETSTACK_CONF_LLSEC
64 #define NETSTACK_CONF_LLSEC noncoresec_driver
65
66 #include "net/llsec/adaptivesec/noncoresec-autoconf.h"
```

Tabla 5–1. Mecanismos de autenticación y cifrado a usar en el escenario de pruebas

Nombre	Descripción
0 - No-Security	Sin cifrado de datos ni autenticación
1 - AES-CBC-MAC-32	Sin cifrado de datos. Autenticación con MAC de 32 bits
2 - AES-CBC-MAC-64	Sin cifrado de datos. Autenticación con MAC de 64 bits
3 - AES-CBC-MAC-128	Sin cifrado de datos. Autenticación con MAC de 128 bits
4 - AES-CTR	Cifrado de datos. Sin autenticación
5 - AES-CCM-32	Cifrado de datos. Autenticación con MAC de 32 bits
6 - AES-CCM-64	Cifrado de datos. Autenticación con MAC de 64 bits
7 - AES-CCM-128	Cifrado de datos. Autenticación con MAC de 128 bits

Los siguientes parámetros a configurar fueron los referentes al protocolo RPL, que en nuestro caso será bastante sencilla su tarea. Pues sólo hay dos dispositivos en el escenario, de modo que el protocolo no tendrá muchas iteraciones. Le definimos que su tiempo de validez será infinito poniendo a 1 el valor de `RPL_CONF_DEFAULT_ROUTE_INFINITE_LIFETIME`.

```
69  /* Define as minutes */
70  #define RPL_CONF_DEFAULT_LIFETIME_UNIT   60
71
72  /* 10 minutes lifetime of routes */
73  #define RPL_CONF_DEFAULT_LIFETIME        10
74
75  #define RPL_CONF_DEFAULT_ROUTE_INFINITE_LIFETIME 1
```

Y para ahorrar un poco de memoria ROM desactivamos la pila TCP y los paquetes fragmentados 6LoWPAN, pues no los vamos a usar. Y también configuramos que los nombres descriptivos de los procesos no vayan en la memoria de programa.

```
77  /* Save some ROM */
78  #undef UIP_CONF_TCP
79  #define UIP_CONF_TCP                    0
80
81  #undef PROCESS_CONF_NO_PROCESS_NAMES
82  #define PROCESS_CONF_NO_PROCESS_NAMES 1
83
84  #undef SICSLOWPAN_CONF_FRAG
85  #define SICSLOWPAN_CONF_FRAG            0
```

Y esto es todo del fichero de configuración del proyecto.

5.6 Makefile

En el fichero Makefile incluímos las instrucciones para compilar el proyecto. Lo primero es indicar el directorio en el que se encuentra la raíz del sistema operativo Contiki:

```
1  CONTIKI=../../../..
```

Después nombraremos las apps de Contiki que usaremos en nuestro proyecto. En nuestro caso `powertrace`, `collect-view`, `servreg-hack` y `reboot-app`.

```
2  APPS =  powertrace collect-view servreg-hack reboot-app
```

Es necesario especificar los ficheros del proyecto y también los ficheros necesarios para el funcionamiento de ambos. En nuestro caso necesitaremos `collect-common.c`. Y los ficheros que queremos compilar `udp-sender` y `udp-sink`.

```
3  CONTIKI_PROJECT = udp-sender udp-sink
4  PROJECT_SOURCEFILES += collect-common.c
```

También se especifica el fichero de configuración del proyecto:

```
6  CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
```

Los objetivos del Makefile se corresponderán a la variable `CONTIKI_PROJECT`:

```
12  all: $(CONTIKI_PROJECT)
```

Por último se indicará que se va a hacer uso de IPv6 y el fichero Makefile por defecto de Contiki:

```
14  CONTIKI_WITH_IPV6 = 1
15  include $(CONTIKI)/Makefile.include
```

6 PRUEBAS Y RESULTADOS

Después de haber programado el software que ejecutarán los dispositivos sensores en el entorno de pruebas, pasé a montar el escenario de pruebas. Trabajé sobre la plataforma Instant Contiki 3.0, que es una máquina virtual Ubuntu para VMWare que trae consigo todo el entorno de desarrollo para Contiki (herramientas de desarrollo, compiladores y simuladores). El simulador de redes de sensores COOJA nos permitirá modificar el entorno de pruebas a nuestro antojo. Con él es posible simular los distintos sensores de la red modificando sus posiciones o el ruido del entorno entre otras muchas cosas. Además nos dará la posibilidad de medir el gasto energético que supone cada una de las soluciones de seguridad probadas.

6.1 Cooja

El simulador de redes de sensores COOJA permite simular sensores que ejecuten el sistema operativo Contiki. El sistema está controlado y analizado por COOJA. Para conseguirlo compila el sistema operativo Contiki para la plataforma nativa del sensor como una librería. Y esa librería la carga en Java usando *Java Native Interfaces* (JNI). Se pueden compilar y cargar muchas librerías Contiki diferentes en la misma simulación de COOJA. De esta forma se tienen distintos tipos de sensores en el a misma red, cada uno llevando a cabo su función. COOJA controla y analiza los sistemas operativos Contiki a través de varias funciones. Por ejemplo, el simulador informa al sistema operativo Contiki que se haga cargo de un evento o puede mirar toda la memoria del dispositivo por si fuera necesario analizarla. Con este enfoque COOJA tiene control absoluto de los sistemas simulados.

Desafortunadamente el uso de JNI tiene también sus desventajas. Y es por eso que se depende de herramientas externas como compiladores, enlazadores y sus argumentos en tiempo de ejecución. COOJA está disponible para Linux, Cygwin/Windows y MacOS.

6.2 Montaje del escenario

Para probar el rendimiento de los distintos mecanismos de cifrado y autenticación montamos el siguiente escenario con dos sensores:

Figura 6-1. Diagrama del escenario de pruebas

En el caso del dispositivo receptor tendremos un sensor ejecutando el programa udp-sink.c que creamos en el anterior capítulo. Este programa consta principalmente de dos procesos que se ejecutan en paralelo. Uno de ellos será un servicio que escucha en un puerto UDP y espera recibir paquetes UDP unicast del receptor. El otro será un servicio que recibirá estadísticas del otro dispositivo como por ejemplo el consumo energético. De esta forma podremos evaluar el gasto que conllevan las distintas medidas adoptadas.

El dispositivo emisor ejecutará el programa udp-sender.c que también creamos en el anterior capítulo. Este programa también ejecutará dos procesos simultáneos. Uno se encargará de enviar paquetes UDP con la cadena "TFG Jose Antonio número_de_secuencia" donde número de secuencia se irá incrementando con cada paquete. Por otra parte tendremos un proceso que se encargará de enviar al receptor, que en este caso hace de colector, la información acerca del consumo energético que ha registrado.

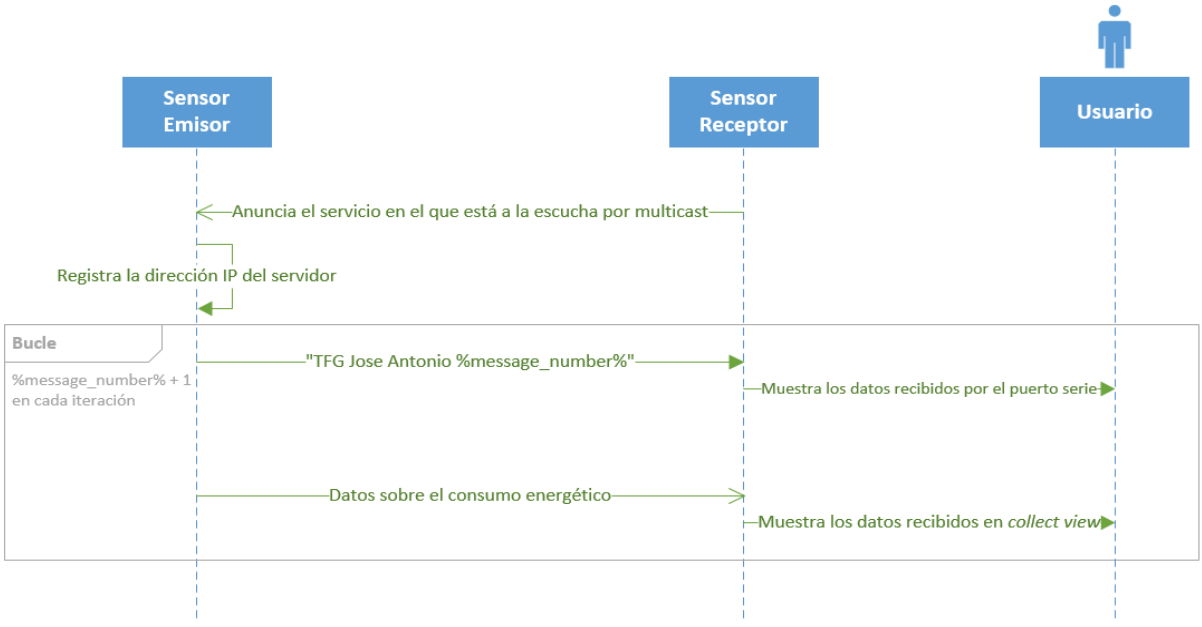

Figura 6-2. Diagrama paso de mensajes del escenario de pruebas

6.2.1 Parámetros de la simulación

Cada fichero de simulación se llamará "TFG-Nombre_del_mecanismo_utilizado" donde Nombre_del_mecanismo_utilizado será el nombre del mecanismo que vayamos a utilizar en el escenario.

Tabla 6–1. Mecanismos de autenticación y cifrado a usar en el escenario de pruebas

Nombre	Descripción
0 - No-Security	Sin cifrado de datos ni autenticación
1 - AES-CBC-MAC-32	Sin cifrado de datos. Autenticación con MAC de 32 bits
2 - AES-CBC-MAC-64	Sin cifrado de datos. Autenticación con MAC de 64 bits
3 - AES-CBC-MAC-128	Sin cifrado de datos. Autenticación con MAC de 128 bits
4 - AES-CTR	Cifrado de datos. Sin autenticación
5 - AES-CCM-32	Cifrado de datos. Autenticación con MAC de 32 bits
6 - AES-CCM-64	Cifrado de datos. Autenticación con MAC de 64 bits
7 - AES-CCM-128	Cifrado de datos. Autenticación con MAC de 128 bits

Una vez dentro del simulador seleccionaremos los siguientes parámetros:

- Radio medium: nos permite seleccionar el medio a través del que los dispositivos transmitirán. Seleccioné *Unit Disk Graph Medium (UDGM): Distance Loss*. De modo que tendrá perdidas según la distancia a la que se encuentren los dispositivos transceptores 802.15.4.
- Mote startup delay (ms): El tiempo que tardarán en arrancar los dispositivos sensores después del comienzo de la simulación. En mi caso escogí un segundo.
- Random seed: Es el número inicializador de números aleatorios. Para todas nuestras simulaciones usaremos el mismo (123456), de manera que sean reproducibles.

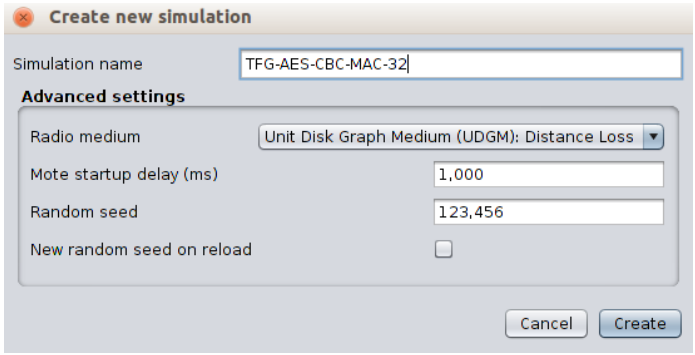

Figura 6-3. Parámetros para la simulación en Cooja

Luego una vez dentro del simulador añadimos los sensores. En mi caso el Sky Mote. Y elegí el fichero udp-sink.c para el sensor receptor. Acto seguido pasé a compilarlo. Luego seleccioné "*Create*" para añadirlo a la simulación.

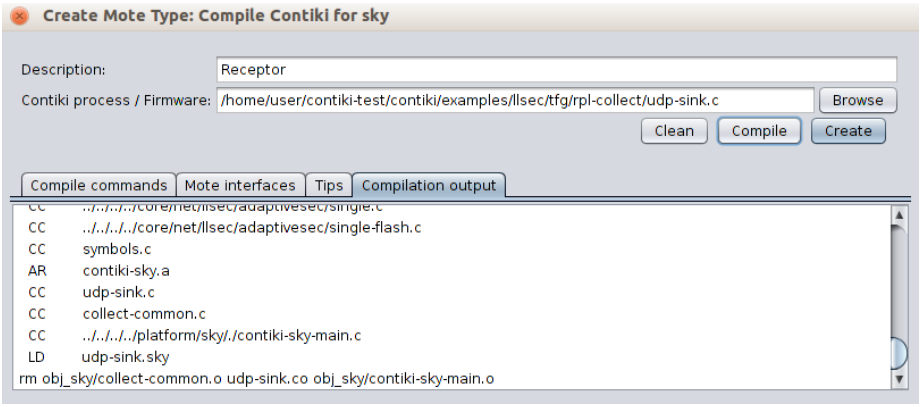

Figura 6-4. Creación de un sensor en Cooja

Lo siguiente fue seleccionar el número de sensores a desplegar (uno en mi caso) y fijar su posición. En el escenario ambos sensores estarán separados 10m y por ello las posiciones serán las siguientes:

- Emisor: X: 45 ; Y: 50 ; Z: 0
- Receptor: X: 55 ; Y: 50 ; Z: 0

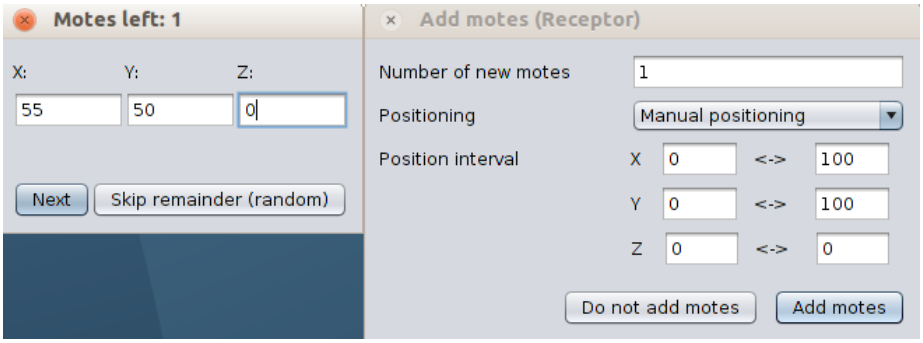

Figura 6-5. Posición del sensor en Cooja

Después seguí los mismos pasos para crear el dispositivo emisor. Sólo que ahora cambié el fichero a compilar por udp-sender.c y la posición correspondiente al emisor. Y finalmente Nos quedará el escenario de la figura 6-6.

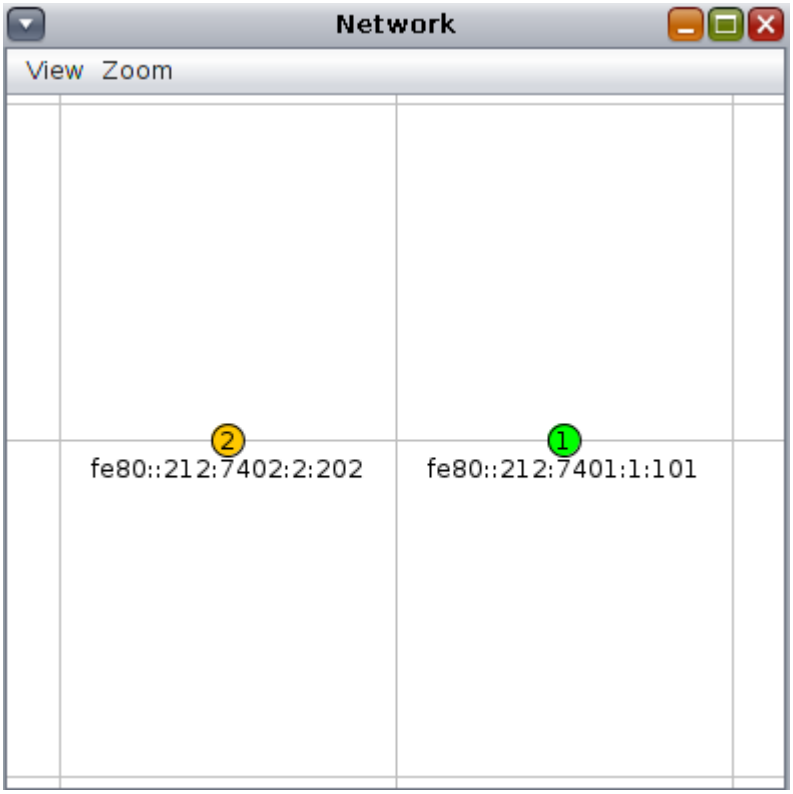

Figura 6-6. Vista del escenario a simular en Cooja

Antes de comenzar la simulación usé la herramienta "Radio messages" de Cooja. Esta herramienta permite exportar los mensajes de las simulaciones a un fichero PCAP para luego poder analizar las comunicaciones.

Por otra parte también usé un script que permitía fijar el tiempo límite a la simulación. En mi caso serían 600.000 ms, que corresponden a 10 minutos. De esta forma todas las medidas se evaluarían en el mismo tiempo.

```
/*
 * Example Contiki test script (JavaScript).
 * A Contiki test script acts on mote output, such as via printf()'s.
 * The script may operate on the following variables:
 *  Mote mote, int id, String msg
 */

TIMEOUT(600000);

while (true) {
  log.log(time + ":" + id + ":" + msg + "\n");
  YIELD();
}
```

Además de fijar el tiempo límite de la simulación, el script también permite mostrar todos los mensajes generados por el programa a través de funciones como `printf ()`.

Lo siguiente fue activar el uso del script. Para ello hice uso de la herramienta "Simulation script editor" que se incluye en Cooja. Y en la pestaña "Run" seleccioné "Activate".

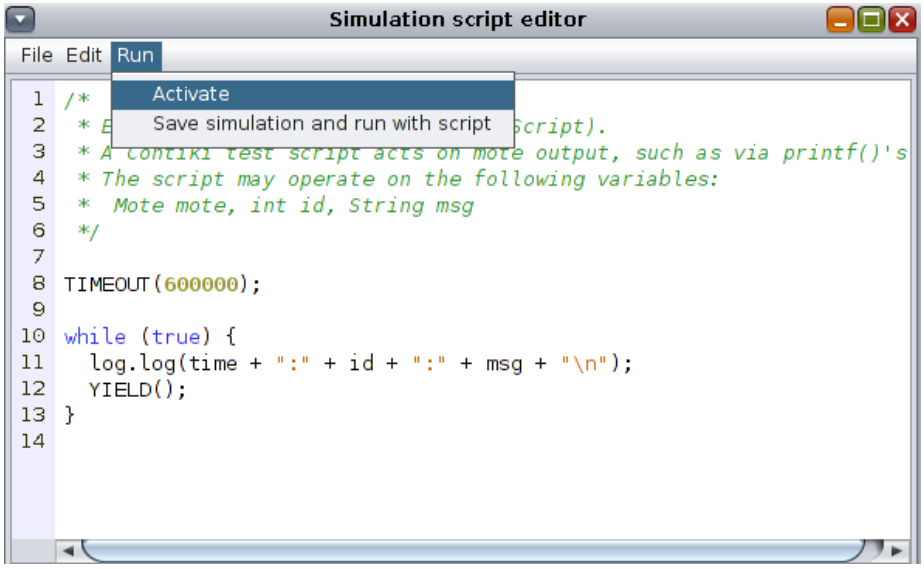

Figura 6-7. Editor de scripts para las simulaciones en Cooja

Lo siguiente fue abrir la herramienta “Collect View” en el dispositivo receptor (1). Esta herramienta permite visualizar el consumo energético en el sensor emisor. De esta forma pude medir el consumo tanto en CPU como en la transmisión que conlleva cada mecanismo de autenticación y seguridad. La configuraremos con los siguientes parámetros:

- Report interval: Este valor determina el intervalo de tiempo en el que el emisor emitirá sus informes. En mi caso será 10 segundos.
- Report randomness: Sirve para introducir un factor aleatorio a la hora de decidir cuándo se envía el informe. En mi caso 0 segundos. Queremos que se envíe exactamente cada 10 segundos en todos los casos.
- Hop-by-hop retransmissions: Permite decidir el número máximo de saltos que puede dar el informe en la red de sensores. Como en nuestro caso no habrá saltos en la red, ya que tenemos dos sensores solamente, lo dejamos a su valor por defecto. Será por tanto 31.
- Number of reports: Establece el número de informes. Lo dejaremos a 0, de esta forma no habrá un límite de informes. Éstos se seguirán enviando hasta el final de la simulación.

Después envié el comando a los nodos con el botón “Send command to nodes” y comencé a recolectar los datos pulsando el botón “Start Collect”.

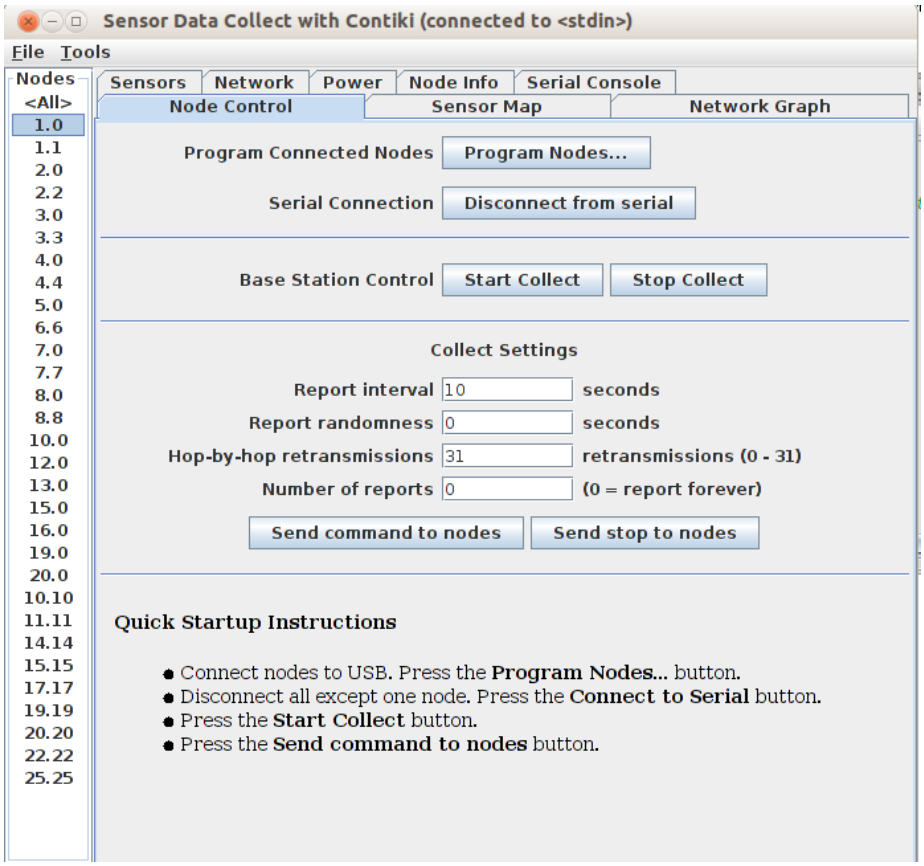

Figura 6-8. Parámetros de Collect View en Cooja

6.3 Simulación

Una vez hecho lo anterior, ya tenía la simulación lista. Pulsé en el botón "Start" de la ventana "Simulation control" y ya se ejecutaría la simulación hasta llegar a los 10 minutos. Durante la simulación se pueden ver las salidas de ambos sensores en la ventana "Simulation script editor". Para ir cambiando de mecanismo de autenticación y/o cifrado iba cambiando el parámetro ADAPTIVESEC_CONF_UNICAST_SEC_LVL y el ADAPTIVESEC_CONF_BROADCAST_SEC_LVL seleccionando los datos según la tabla 6-1. Ya no se muestran en la ventana "Mote output", esto es debido al uso del script para fijar el tiempo límite. Para evaluar los distintos mecanismos de autenticación y seguridad seguí varios criterios: consumo energético, tamaño del programa y retardo añadido a las comunicaciones.

6.3.1 Consumo energético

El resultado en el caso del algoritmo AES-CBC-MAC-32 fue el de la figura 6-10. Podemos acercar la imagen seleccionando la zona de la gráfica que queramos ver con más detalle o también pulsando el botón derecho del ratón y seleccionando "Zoom in". Si dejamos el ratón encima de alguna de las distintas zonas podemos ver los valores exactos que corresponden al consumo de CPU, transmisión y recepción o del modo de bajo consumo (LPM). Veremos que el consumo del módulo de comunicaciones en escucha ("Radio listen") es muy elevado en comparación al resto. Esto se debe a que, como ya vimos en el capítulo del sistema operativo Contiki, el ciclo de trabajo ("RDC: Radio duty cycle") del receptor radio está fijado a 100%. De modo que siempre está a la escucha. Esto se hizo así para ahorrar memoria de programa (ROM), ya que al incluir otros drivers de Contiki para gestionar el RDC nuestro programa no entraba en la memoria del dispositivo. De esta forma

conseguí el espacio necesario. Al fin y al cabo ese gasto es irrelevante para el estudio que se pretende realizar. Nos interesa por tanto comparar el consumo de CPU y transmisión radio.

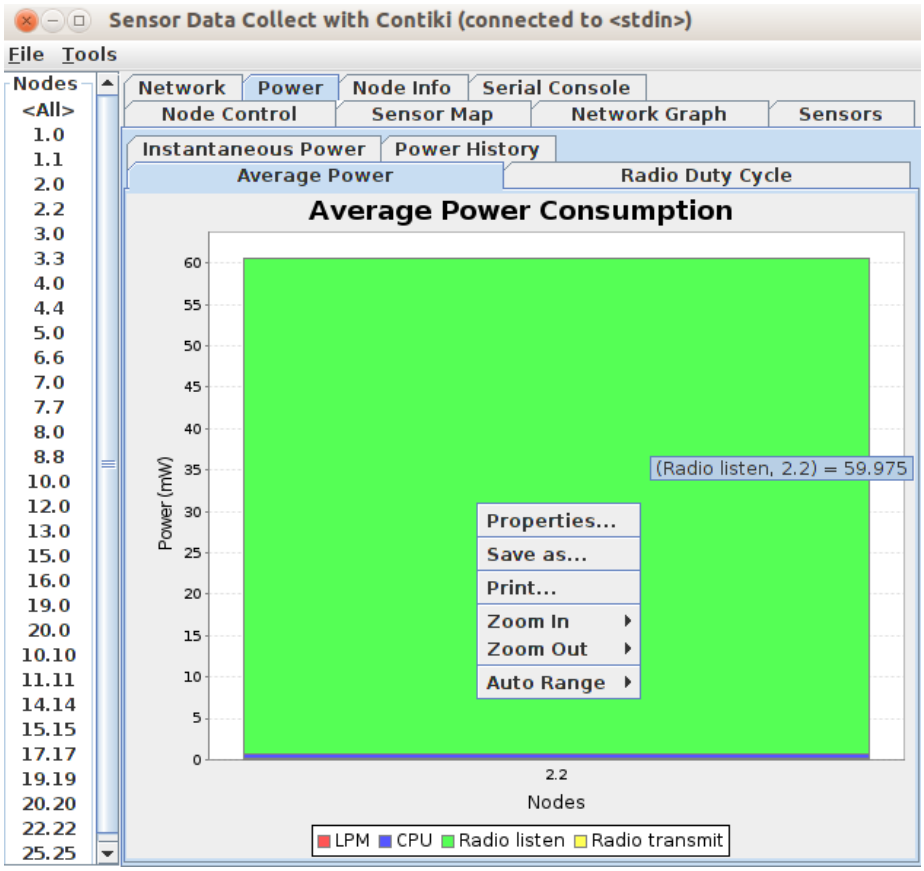

Figura 6-9. Consumo energético medio en en Cooja

6.3.2 Tamaño del programa

Además podemos mirar el tamaño del programa que hemos cargado en el sensor. De esta forma podemos evaluar el uso que hace en memoria cada mecanismo. Al compilar mi programa con Cooja y crear los sensores se generó el programa que habría que cargar en el sensor real. Usaremos este fichero para averiguar el tamaño usado. La salida de la utilidad `size` da la siguiente información acerca del programa que vamos a cargar en el sensor TMote Sky:

- **Text**: Es el tamaño del código que irá en la flash ROM.
- **Data**: Se trata del tamaño usado para datos inicializados. Estas variables que se guardan en RAM y se inicializan con valores almacenados en la ROM.
- **BSS** (*Block Started by Symbol*): Contiene todos los datos no inicializados o inicializados a cero que irán en la RAM.
- **Dec**: Corresponde al número total de bytes de "Text", "Data" y "BSS".
- **Hex**: Es exactamente lo mismo que "Dec" pero en valor hexadecimal.

De modo que el tamaño en ROM del programa es: $Text + Data = ROM\ usada$

```
user@instant-contiki:~/contiki-test/contiki/examples/llsec/tfg/rpl-collect$ size udp-sink.sky
   text    data     bss     dec     hex filename
  46093     278    6408   52779    ce2b udp-sink.sky
user@instant-contiki:~/contiki-test/contiki/examples/llsec/tfg/rpl-collect$ size udp-sender.sky
   text    data     bss     dec     hex filename
  47677     280    6462   54419    d493 udp-sender.sky
user@instant-contiki:~/contiki-test/contiki/examples/llsec/tfg/rpl-collect$
```

Figura 6-10. Tamaño de los programas para el sensor TMote Sky

Para hacer la comparativa me fijé en las variaciones de estos tamaños según el mecanismo de seguridad y autenticación escogido.

6.3.3 Retardo en las comunicaciones

Al acabar la simulación, gracias a la herramienta "Radio messages", se guardan todas las comunicaciones hechas de modo que podemos analizarlas. Pero si queremos medir el retardo que existe al cifrar y añadir el código de autenticación del mensaje por parte del emisor, y descifrar y autenticar los mensajes por parte del receptor, no podemos hacerlo a través del fichero PCAP. Esto es debido a que cuando la herramienta "Radio messages" capta los mensajes, éstos ya van cifrados y/o autenticados. Por tanto se desconoce el tiempo que le ha llevado al dispositivo cifrar los datos y añadir el código de autenticación. Pero es que tampoco han sido descifrados ni autenticados por el dispositivo receptor. Por tanto tuve que buscar otra forma de medir este retardo.

La manera finalmente empleada fue la que ya se vio en el anterior capítulo: Antes de enviar el mensaje usamos la función `printf ()` para anunciar que se va a enviar. Así se mostrará en la ventana "Simulation script editor". Y al haberlo recibido también se muestra por la salida del dispositivo receptor. De esta forma se tiene un intervalo de tiempo desde antes de proceder al envío hasta justo después de recibirlo. Será este el tiempo a comparar entre los distintos mecanismos de seguridad y autenticación.

En la figura 6-11 se puede apreciar el tiempo en milisegundos a la izquierda, el identificador del dispositivo (1-receptor ; 2-emisor). Y los datos de la transmisión y recepción.

Para hacer la comparativa restaremos el número de milisegundos del envío por parte del emisor al que corresponde a la recepción por parte del receptor.

```
20544626:2:Sending unicast to fd00::212:7401:1:101
20570190:1:Data received from fd00::212:7402:2:202 on port 1234 from port 1234 with length 19: 'TFG Jose Antonio 1'
30544535:2:Sending unicast to fd00::212:7401:1:101
30570144:1:Data received from fd00::212:7402:2:202 on port 1234 from port 1234 with length 19: 'TFG Jose Antonio 2'
40544478:2:Sending unicast to fd00::212:7401:1:101
40570101:1:Data received from fd00::212:7402:2:202 on port 1234 from port 1234 with length 19: 'TFG Jose Antonio 3'
50544607:2:Sending unicast to fd00::212:7401:1:101
50570185:1:Data received from fd00::212:7402:2:202 on port 1234 from port 1234 with length 19: 'TFG Jose Antonio 4'
```

Figura 6-11. Salida estándar de ambos dispositivos

6.4 Resultados

A continuación se pasará a mostrar y comentar los resultados obtenidos durante las pruebas en los distintos escenarios. Como ya se ha dicho, las medidas que se han tomado pretenden medir tres aspectos de las distintas medidas: consumo energético, aumento de la memoria usada y retardo añadido en las comunicaciones entre sensores.

6.4.1 Consumo energético

A través de la herramienta *Collect-View* que trae Contiki, se recogieron los resultados al usar los distintos tipos de medidas de seguridad. El método seguido para realizar las mediciones es el que se explicó en el apartado anterior. Los resultados corresponden al sensor emisor.

Tabla 6–2. Potencia media consumida por los distintos mecanismos de autenticación y/o cifrado en el dispositivo emisor

Mecanismo de autenticación y/o seguridad	Radio transmit (mW)	CPU (mW)	LPM (mW)	Radio Listen (mW)	Total (mW)
Sin seguridad	0,02	0,408	0,151	59,973	60,552
AES-CBC-MAC-32	0,021	0,493	0,149	59,975	60,638
AES-CBC-MAC-64	0,022	0,493	0,149	59,973	60,637
AES-CBC-MAC-128	0,024	0,487	0,149	59,972	60,632
AES-CTR	0,02	0,492	0,149	59,976	60,637
AES-CCM-32	0,021	0,495	0,149	59,975	60,64
AES-CCM-64	0,022	0,495	0,149	59,974	60,64
AES-CCM-128	0,024	0,489	0,149	59,972	60,634

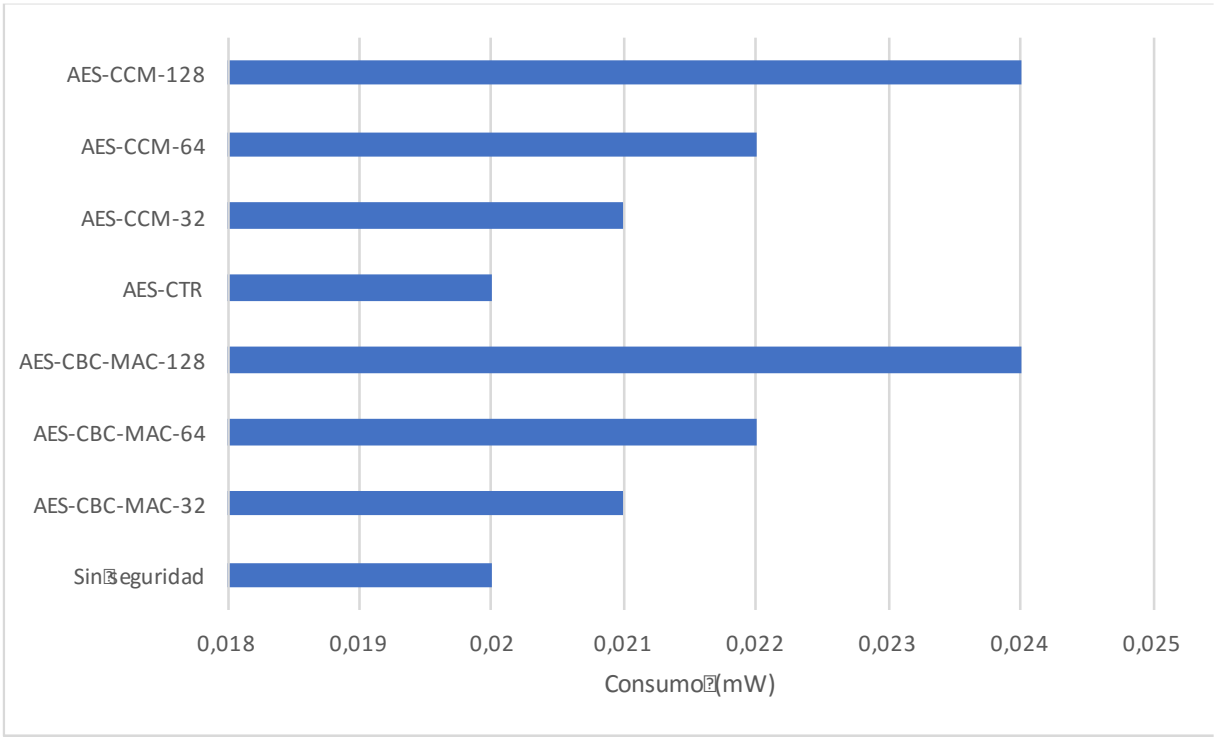

Figura 6-12. Gráfico de la potencia media usada en la CPU del sensor emisor

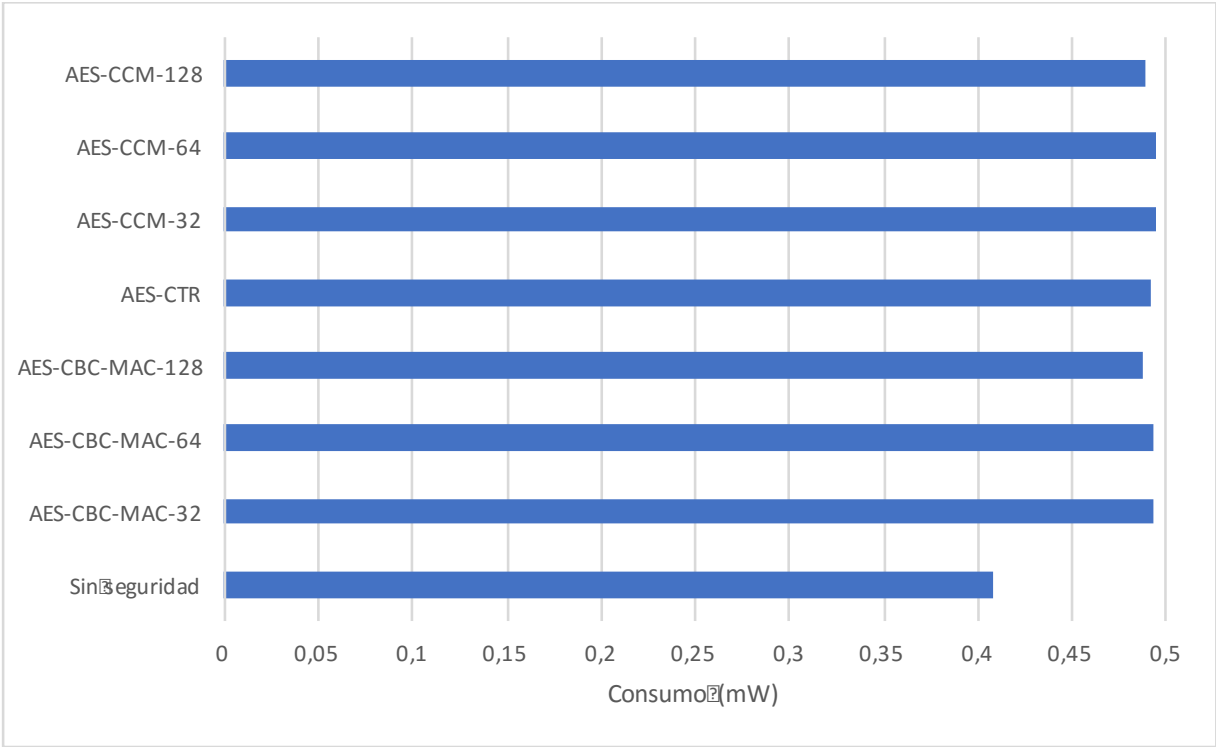

Figura 6-13. Gráfico de la potencia media usada en la transmisión por radio del sensor emisor

Al no implementar ningún protocolo de *Radio Duty Cycling*, ni de control de acceso al medio (*MAC*), la radio permanece siempre encendida. Conviene usarlo para conseguir una mayor vida de la pila o batería en

aplicaciones reales. Pero como ya se explicó anteriormente, al aplicar las medidas de autenticación y/o cifrado y además recolectar los datos del consumo energético con la app `Collect-View`, el tamaño del programa no dejaba lugar a la inclusión de este tipo de protocolos para la gestión de la radio. De todas formas, en las medidas que vamos a probar no resulta especialmente importante el consumo de la radio en escucha.

Aunque en las tablas se aporten todos los datos, nos fijaremos sobre todo en el consumo energético en la CPU y en las transmisiones radio. El modo de bajo consumo LPM no es relevante, al igual que la radio en escucha. El consumo medio de las distintas medidas de autenticación y cifrado de la CPU es de 0,492mW. En el caso de la potencia media de transmisión, de 0,022mW.

Para ilustrar y pasar a comentar las diferencias, pasaremos a mostrar las diferencias de potencia respecto a no aplicar ninguna medida en el escenario de pruebas en la tabla 6-3.

Tabla 6–3. Diferencia de potencia media de los distintos mecanismos de autenticación y/o cifrado en el dispositivo emisor

Mecanismo de autenticación y/o seguridad	Radio transmit (mW)	CPU (mW)	LPM (mW)	Radio Listen (mW)	Total (mW)
Sin seguridad	0	0	0	0	0
AES-CBC-MAC-32	0,001	0,085	-0,002	0,002	0,086
AES-CBC-MAC-64	0,002	0,085	-0,002	0	0,085
AES-CBC-MAC-128	0,004	0,079	-0,002	-0,001	0,08
AES-CTR	0	0,084	-0,002	0,003	0,085
AES-CCM-32	0,001	0,087	-0,002	0,002	0,088
AES-CCM-64	0,002	0,087	-0,002	0,001	0,088
AES-CCM-128	0,004	0,081	-0,002	-0,001	0,082

Nos centraremos en el consumo medio de potencia de la CPU y la transmisión por radio. En el caso de la potencia media consumida por la CPU se puede ver que los algoritmos de 128 bits requieren menos potencia que el resto. Esto es porque el cifrado de AES de 128 bits está implementado nativamente en el hardware del sensor T-Mote Sky. Aunque también estos algoritmos resultan ser los que más potencia necesitan para transmitir, pues necesitan añadir un código de autenticación MAC más grande al final del mensaje.

El TMote-Sky se suele alimentar de 2 pilas AA que tienen una capacidad cada una de 2800mWh. Calculando como en la ecuación 6-1, a partir de los datos del consumo de cada una de las medidas de autenticación y/o cifrado se ha elaborado la tabla 6-4. En ella se recoge las diferencias de tiempo de uso que pueden aportar las pilas con cada una de las medidas.

$$\frac{2800 \times 2\ (mWh)}{Potencia\ media\ (mW)} = Duración\ total\ estimada \tag{6–1}$$

Tabla 6–4. Diferencia de potencia media de los distintos mecanismos de autenticación y/o cifrado en el dispositivo emisor

Mecanismo de autenticación y/o seguridad	Duración total estimada (días / horas:minutos:sengundos)	Diferencia de duración (horas:minutos:segundos)
Sin seguridad	3 / 20:28:57	0:00:00
AES-CBC-MAC-32	3 / 20:21:05	-0:07:52
AES-CBC-MAC-64	3 / 20:21:10	-0:07:47
AES-CBC-MAC-128	3 / 20:21:38	-0:07:19
AES-CTR	3 / 20:21:10	-0:07:47
AES-CCM-32	3 / 20:20:54	-0:08:03
AES-CCM-64	3 / 20:20:54	-0:08:03
AES-CCM-128	3 / 20:21:27	-0:07:30

De nuevo hay que recordar que los datos de la duración estimada están influenciados por la energía que consume tener la radio a la escucha siempre que no se está transmitiendo. En una aplicación real, donde se use un protocolo de Duty Cycling y de control de acceso al medio, la duración total será mucho mayor que eso. Si no tendríamos que estar constantemente cambiando las pilas a los sensores.

Se tiene que el tiempo medio de diferencia al aplicar las medidas es de -0:07:46. Teniendo en cuenta que la duración de las pilas en aplicaciones reales sin tener la radio encendida será mucho mayor, resulta razonable en el aspecto del consumo energético aplicar medidas de autenticación y/o cifrado.

6.4.2 Tamaño del programa

Como se vio en apartados anteriores, para obtener el tamaño del programa se usó la herramienta `size`. Pasándole como parámetro el nombre del fichero generado por el compilador para el sensor. Este fichero en nuestro caso tiene la extensión `.sky`.

Los resultados en el caso del tamaño del programa del receptor fueron los siguientes:

Tabla 6–5. Tamaño del programa del sensor receptor

Mecanismo de autenticación y/o seguridad	Text	Data	BSS	Dec
Sin seguridad	43753	258	5780	49791
AES-CBC-MAC-32	47657	280	6450	54387
AES-CBC-MAC-64	47657	280	6450	54387
AES-CBC-MAC-128	47665	280	6450	54395
AES-CTR	47619	280	6450	54349
AES-CCM-32	47659	280	6450	54389
AES-CCM-64	47659	280	6450	54389
AES-CCM-128	47667	280	6450	54397

Recordemos que como vimos en el apartado anterior, el tamaño del programa en ROM es el de *Text+Data*. Y el campo *BSS* corresponde a la RAM reservada al principio del programa. En la figura 6-14 se puede ver más clara la diferencia de tamaños entre los distintos mecanismos de autenticación y seguridad.

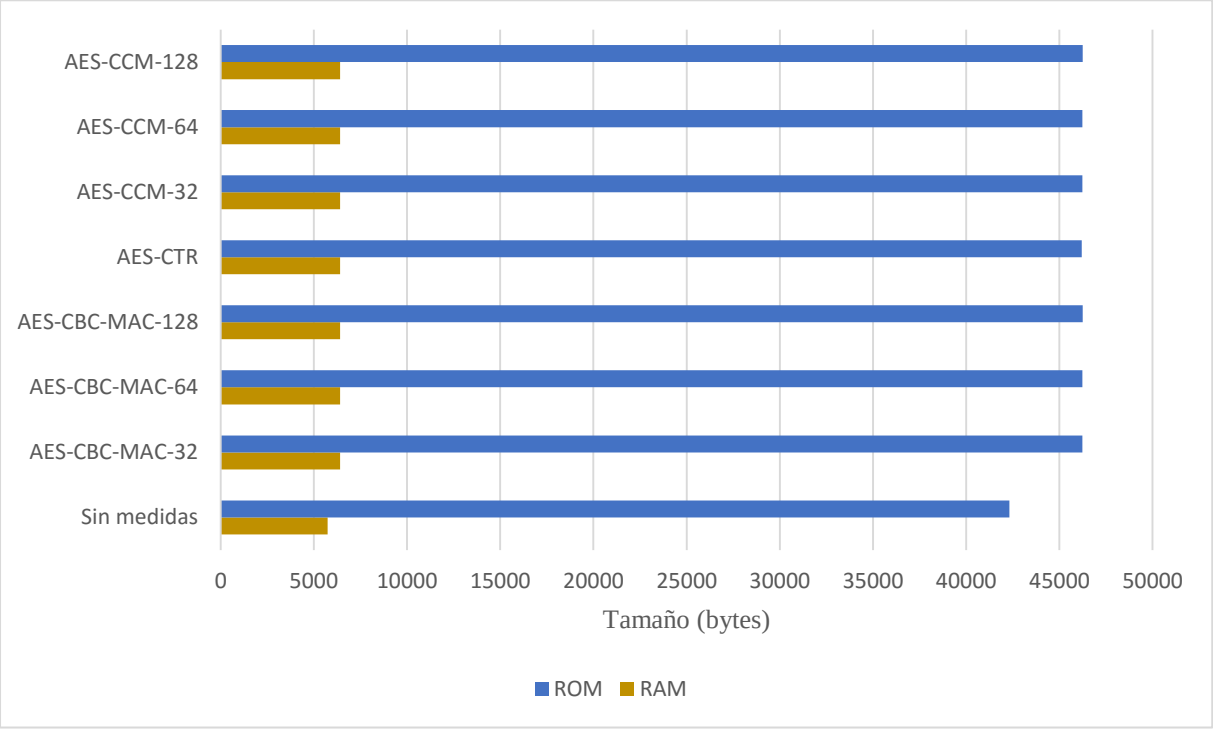

Figura 6-14. Gráfica del tamaño que ocupa en ROM y en RAM el programa del sensor receptor

Ahora pasaremos a mostrar las diferencias de tamaño entre las distintas medidas de autenticación y/o cifrado y el uso de ninguna medida, para que pueda verse más claro.

Tabla 6–6. Diferencia de tamaño del programa del sensor receptor con respecto a no usar ninguna medida de autenticación y seguridad

Mecanismo de autenticación y/o seguridad	Text	Data	BSS	Dec
AES-CBC-MAC-32	3904	22	670	4596
AES-CBC-MAC-64	3904	22	670	4596
AES-CBC-MAC-128	3912	22	670	4604
AES-CTR	3866	22	670	4558
AES-CCM-32	3906	22	670	4598
AES-CCM-64	3906	22	670	4598
AES-CCM-128	3914	22	670	4606

Se puede apreciar que en el caso de los algoritmos de autenticación AES-CBC-MAC todos tienen prácticamente el mismo tamaño. Sólo varían 8 bytes en el caso de AES-CBC-MAC-128.

Todos los mecanismos tienen el mismo tamaño en *Data* y *BSS*. Es decir, ninguno de ellos aumenta el consumo de RAM al inicio del programa más que otro. E inicializan la misma cantidad de datos en RAM (22 bytes) que corresponden al apartado *Data*.

El algoritmo de cifrado AES-CTR es la más liviana de todas las medidas. De modo que puede ser una buena opción si disponemos de poca ROM. Con él nos aseguramos que nuestras comunicaciones no pueden ser leídas en claro.

En cuanto a los mecanismos que proporcionan cifrado y autenticación de las tramas, el algoritmo AES-CCM-128 tiene ligeramente más tamaño que el resto de mecanismos de cifrado y autenticación.

En la figura 6-15 se puede apreciar visualmente la diferencia de tamaños en RAM con respecto a no usar ninguna medida.

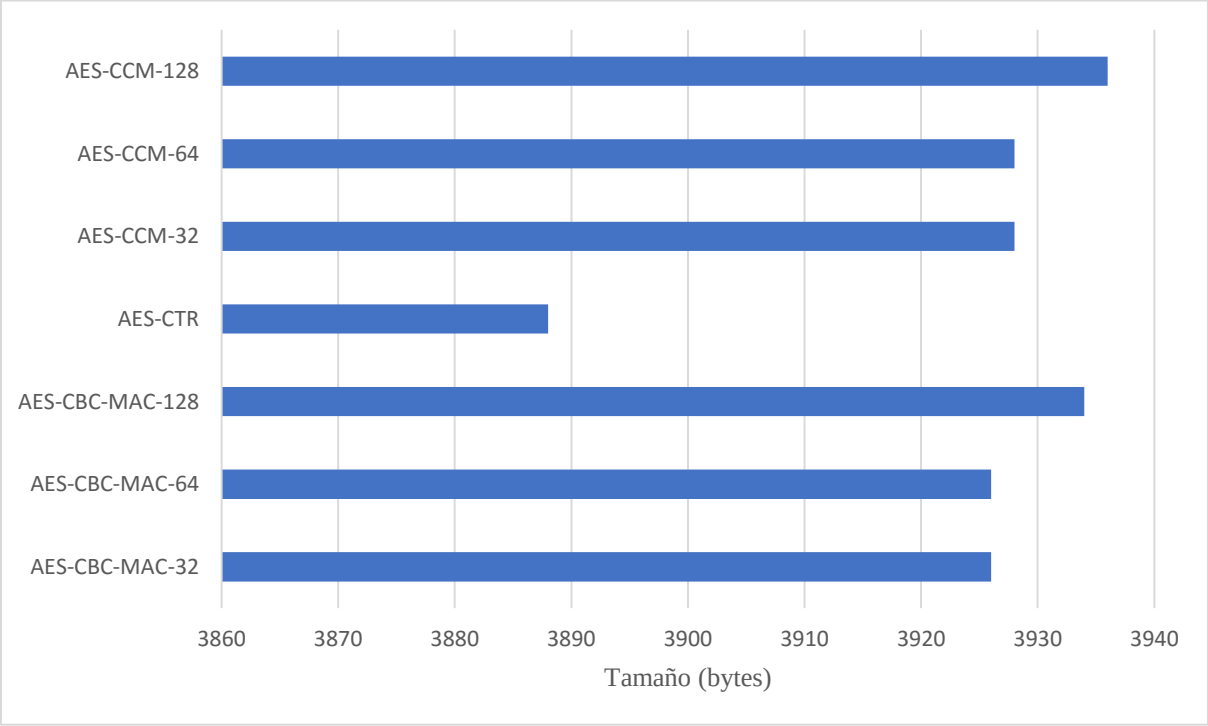

Figura 6-15. Gráfica de la diferencia de tamaños en ROM de cada medida de autenticación y/o cifrado con respecto a no usar ninguna medida

En el caso del programa del sensor emisor, los resultados mantuvieron las mismas diferencias de tamaño al aplicar las medidas de seguridad y/o cifrado. A continuación se muestra la tabla 6-7 con los valores de tamaño del programa del sensor emisor.

Tabla 6–7. Tamaño del programa del sensor receptor

Mecanismo de autenticación y/o seguridad	Text	Data	BSS	Dec
Sin medidas	43753	258	5780	49791
AES-CBC-MAC-32	47657	280	6450	54387
AES-CBC-MAC-64	47657	280	6450	54387
AES-CBC-MAC-128	47665	280	6450	54395
AES-CTR	47619	280	6450	54349
AES-CCM-32	47659	280	6450	54389
AES-CCM-64	47659	280	6450	54389
AES-CCM-128	47667	280	6450	54397

En este caso para mostrar el tamaño del programa, sólo lo haremos del uso de la ROM. Pues ya hemos visto que en el caso de la RAM, es el mismo en todas las medidas aplicadas.

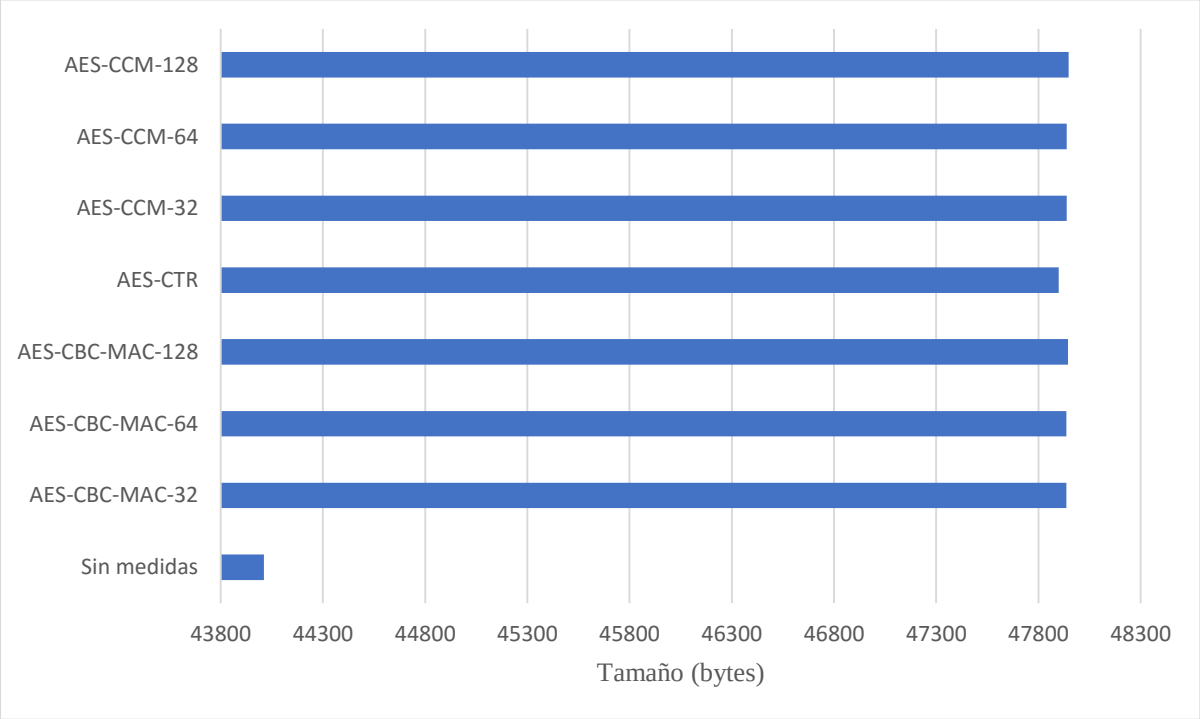

Figura 6-16. Gráfica del tamaño que ocupa en ROM el programa del sensor emisor

Las diferencias de tamaño en el emisor son las mismas que en el caso del receptor. Por tanto no se aporta la tabla de diferencias ni la gráfica.

6.4.3 Retardo en las comunicaciones

El retardo en las comunicaciones puede ser crítico en ciertas aplicaciones. Sobre todo en aquellas que trabajen en tiempo real con los datos de los sensores. Por ello se llevaron a cabo mediciones de estos retardos durante las pruebas.

El procedimiento seguido para obtener los tiempos de las mediciones es el que se vio en el apartado anterior. Se tomaron para las medidas los tiempos de envío y recepción de los paquetes 1, 2 y 3. En la tabla 6-8 se muestran los tiempos obtenidos (los valores de tiempo están expresados en segundos).

Tabla 6–8. Tiempos de envío y recepción de los mensajes en cada una de las medidas

Mecanismo de autenticación y/o seguridad	Paquete enviado 1	Paquete recibido 1	Paquete enviado 2	Paquete recibido 2	Paquete enviado 3	Paquete recibido 3
Sin medidas	30,531865	30,550859	40,531844	40,550839	50,531825	50,55082
AES-CBC-MAC-32	30,543794	30,569324	40,543737	40,569283	50,543832	50,569385
AES-CBC-MAC-64	30,543794	30,56955	40,543737	40,56952	50,543832	50,569602
AES-CBC-MAC-128	30,543794	30,570079	40,543737	40,570037	50,543832	50,570134
AES-CTR	30,543794	30,573285	40,543737	40,573242	50,543832	50,573322
AES-CCM-32	30,543794	30,573398	40,543737	40,573359	50,543832	50,573447
AES-CCM-64	30,543794	30,57359	40,543737	40,573564	50,543832	50,573618
AES-CCM-128	30,543794	30,574054	40,543737	40,574016	50,543832	50,574112

El retardo se muestra en la tabla 6-9. Y en la figura 6-17 también se puede ver con más claridad la diferencia de retardo que suponen las distintas medidas de autenticación y/o cifrado que se han probado.

Tabla 6–9. Retardos medidos, retardo medio y diferencia de retardo en cada una de las medidas

Mecanismo de autenticación y/o seguridad	Retardo 1 (ms)	Retardo 2 (ms)	Retardo 3 (ms)	Retardo medio (ms)	Diferencia de retardo (ms)	Diferencia de retardo (%)
Sin medidas	18,994	18,995	18,995	18,99466667	0	0,00%
AES-CBC-MAC-32	25,53	25,546	25,553	25,543	6,548333333	+25,64%
AES-CBC-MAC-64	25,756	25,783	25,77	25,76966667	6,775	+26,29%
AES-CBC-MAC-128	26,285	26,3	26,302	26,29566667	7,301	+27,77%
AES-CTR	29,491	29,505	29,49	29,49533333	10,50066667	+35,60%
AES-CCM-32	29,604	29,622	29,615	29,61366667	10,619	+35,86%
AES-CCM-64	29,796	29,827	29,786	29,803	10,80833333	+36,27%
AES-CCM-128	30,26	30,279	30,28	30,273	11,27833333	+37,26%

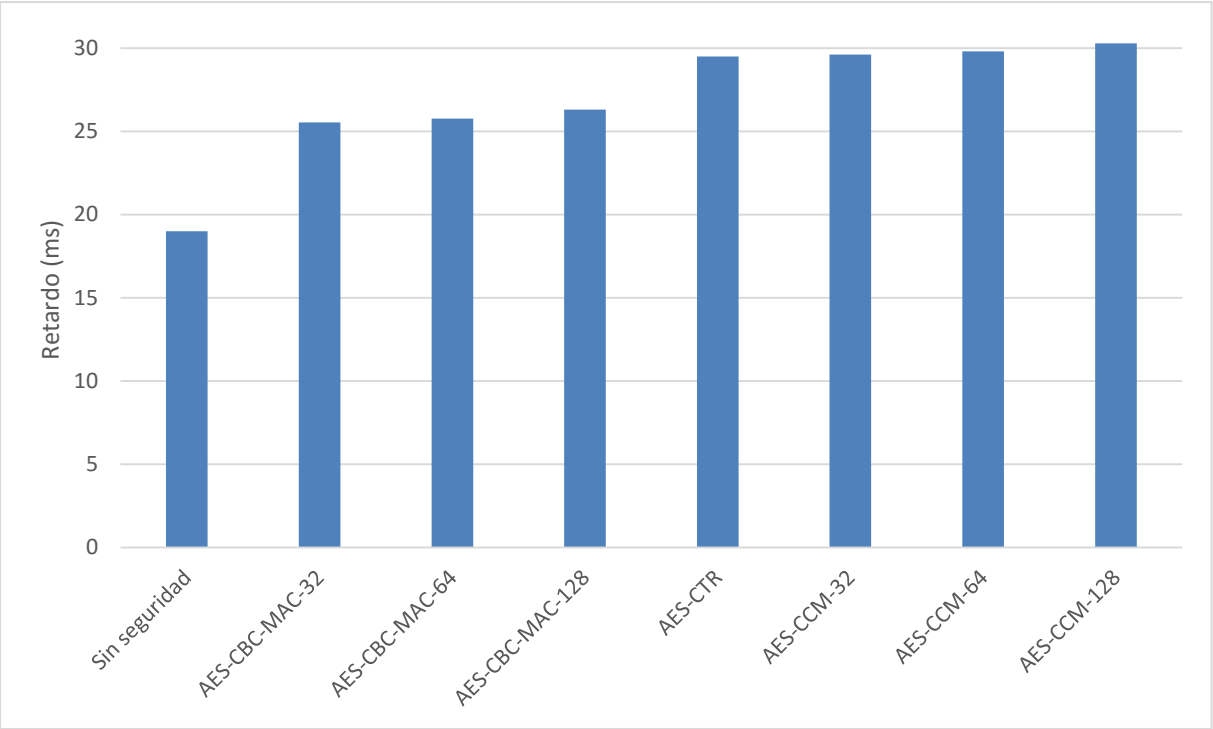

Figura 6-17. Gráfica de la diferencia de retardo medio entre las distintas medidas

El retardo medio añadido a las comunicaciones a causa de las medidas de autenticación y/o cifrado es un 32,10%. Las medidas que implican cifrado de los datos del mensaje suponen un mayor retardo.

6.5 Conclusiones

Después de haber llevado a cabo las pruebas de los distintos mecanismos de autenticación y/o cifrado en el entorno de pruebas, vamos a extraer una serie de conclusiones.

En cuanto al consumo energético, parece bastante asumible el uso de alguna medida de autenticación y/o seguridad. En las pruebas realizadas, el uso de la medida de seguridad más robusta AES-CCM-128, sólo restaba unos 7 minutos y 30 segundos a la duración de la batería, de un total de más de 3 días y 20 horas.

Con respecto al tamaño de programa, habría que evaluar la situación para cada aplicación. Pues la memoria de programa es muy limitada. En el caso del TMote-Sky sólo 48k de memoria Flash. Así que habrá que ver si hay hueco para aproximadamente 4k de memoria de programa y unos 600 bytes de memoria RAM.

En cuanto al retardo añadido a las comunicaciones, habría que entrar a valorar si es aceptable en la aplicación que vayan a desempeñar los sensores. Pero por lo general un retardo añadido de unos 6 a 11 ms suele ser admisible a cambio de la seguridad que se obtiene.

Por tanto, es necesario ver la naturaleza de la aplicación y valorar todos los aspectos como los evaluados en apartados anteriores.

6.6 Líneas futuras

Este trabajo de fin de grado se ha limitado a probar medidas de seguridad en la capa de enlace (802.15.4). Esto no nos proporciona ningún grado de seguridad en las comunicaciones IPv6 a través de Internet.

Por ello, un posible trabajo futuro sería implementar sobre este mismo entorno de pruebas otras medidas de seguridad que sí proporcionen seguridad a los sensores en Internet. Existen implementaciones de IPsec para 6LoWPAN en Contiki. Con ello se conseguiría asegurar las comunicaciones extremo a extremo a través de Internet. El nodo receptor de nuestro entorno de pruebas pasaría a hacer de pasarela a Internet. Y habrá otro equipo en una red externa IPv6 que recibirá los paquetes IPv6 y los registrará.

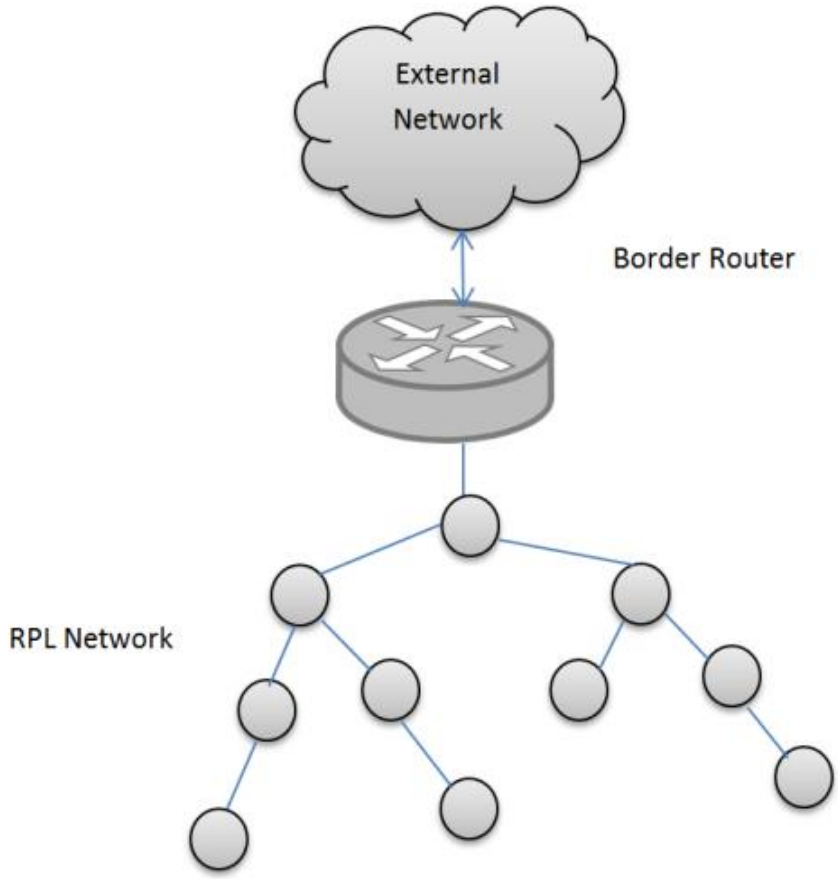

Figura 6-18. Escenario de una WSN conectada a Internet a través de una pasarela

Otra posible prueba de seguridad sobre el entorno sería añadir soporte para TLS/DTLS en la aplicación del emisor y el receptor. De esta forma conseguiríamos seguridad a nivel de aplicación. Se presupone que serán unas medidas de seguridad poco ligeras para lo que viene siendo el software de estas plataformas. Pero por eso habría que realizar las pruebas y ver si es viable su implantación. Existen implementaciones de TLS/DTLS para Contiki.

Y dejando de lado la seguridad en Internet e intentando mejorar la seguridad dentro de la WSN, se podría mejorar el escenario planteado. En este trabajo se ha usado una clave precompartida por los nodos (PSK) para el cifrado de los mensajes y los códigos MAC. Si en vez de ello se plantease un escenario con una infraestructura de clave

pública, los nodos se podrían autenticar con identidad propia. De esta forma si la clave de un nodo es vulnerada, no se vería vulnerada toda la red, sólo el nodo en particular. A la hora de escribir estas líneas, existe un desarrollador de la comunidad Contiki trabajando en una solución de este tipo. Podría probarse qué tal funciona el sistema de intercambio de claves en el entorno que se ha desarrollado en este trabajo.

Anexo A: Código de las pruebas

udp-sender.c:

```
/*
* Copyright (c) 2016, Jose Antonio Caballero Martos.
* Universidad de Sevilla
*
* Copyright (c) 2010, Swedish Institute of Computer Science.
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or
without
* modification, are permitted provided that the following
conditions
* are met:
* 1. Redistributions of source code must retain the above
copyright
*    notice, this list of conditions and the following
disclaimer.
* 2. Redistributions in binary form must reproduce the above
copyright
*    notice, this list of conditions and the following
disclaimer in the
*    documentation and/or other materials provided with the
distribution.
* 3. Neither the name of the Institute nor the names of its
contributors
*    may be used to endorse or promote products derived from
this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS
``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR
CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
```

```
26  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
    INTERRUPTION)
27  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
    CONTRACT, STRICT
28  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
    IN ANY WAY
29  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
    POSSIBILITY OF
30  * SUCH DAMAGE.
31  *
32  * This file is part of the Contiki operating system.
33  *
34  */
35
36  #include "contiki.h"
37  #include "sys/ctimer.h"
38  #include "sys/etimer.h"
39  #include "net/ip/uip.h"
40  #include "net/ipv6/uip-ds6.h"
41  #include "net/ip/uip-udp-packet.h"
42  #include "simple-udp.h"
43  #include "net/rpl/rpl.h"
44  #include "dev/serial-line.h"
45  #include "lib/random.h"
46
47  #if CONTIKI_TARGET_Z1
48  #include "dev/uart0.h"
49  #else
50  #include "dev/uart1.h"
51  #endif
52  #include "collect-common.h"
53  #include "collect-view.h"
54
55  #include <stdio.h>
56  #include <string.h>
57
58  #define UDP_CLIENT_PORT 8775
59  #define UDP_SERVER_PORT 5688
60
61  #define DEBUG DEBUG_PRINT
62  #include "net/ip/uip-debug.h"
63
64  #include "reboot-app.h"
65  #include "servreg-hack.h"
66
67
68  #define UDP_PORT 1234
69  #define SERVICE_ID 190
70
71  /*
72  * Time interval to send another message
73  */
74  #define SEND_INTERVAL        (10 * CLOCK_SECOND)
75
76  /*
77  * Set this to 1 in order to show the printf() output
78  * Set to another value to save ROM
```

```
 79  */
 80  #define DEBUG_SENDER 1
 81
 82  /*
 83  * Last sequence number for the messages
 84  */
 85  #define MAX_MESSAGE_NUMBER 65535
 86
 87  /* Unicast sender connections*/
 88  static struct simple_udp_connection unicast_connection;
 89  static uip_ipaddr_t *addr;
 90
 91  /* Collect View client connections */
 92  static struct uip_udp_conn *client_conn;
 93  static uip_ipaddr_t server_ipaddr;
 94
 95  /*---------------------------------------------------------------
     -------------*/
 96
 97  /*
 98  *
 99  * collection_view_client_process: will send the statistics to
     the
100  * Collection View server
101  *
102  * unicast_sender_process: will send the unicast packets to the
     sink
103  *
104  */
105 PROCESS(collection_view_client_process, "Collection View client
     process");
106 PROCESS(unicast_sender_process, "Unicast sender example
     process");
107 AUTOSTART_PROCESSES(&unicast_sender_process,
     &collection_view_client_process, &collect_common_process);
108
109 /*---------------------------------------------------------------
     -------------*/
110
111 void collect_common_set_sink (void)
112 {
113   /* A udp client can never become sink */
114 }
115
116 /*---------------------------------------------------------------
     -------------*/
117
118 /*
119 * collect_common_net_print: Prints the routing information
120 */
121 void collect_common_net_print (void)
122 {
123   rpl_dag_t *dag;
124   uip_ds6_route_t *r;
125
126   /* Let's suppose we have only one instance */
```

```
127   dag = rpl_get_any_dag();
128   if(dag->preferred_parent != NULL) {
129     PRINTF("Preferred parent: ");
130     PRINT6ADDR(rpl_get_parent_ipaddr(dag->preferred_parent));
131     PRINTF("\n");
132   }
133   for(r = uip_ds6_route_head();
134       r != NULL;
135       r = uip_ds6_route_next(r)) {
136     PRINT6ADDR(&r->ipaddr);
137   }
138   PRINTF("---\n");
139 }
140 
141 /*---------------------------------------------------------------
    -------------*/
142 
143 static void tcpip_handler(void)
144 {
145   if(uip_newdata()) {
146     /* The sender ignores incoming data */
147   }
148 }
149 
150 /*---------------------------------------------------------------
    -------------*/
151 
152 /*
153 * collect_common_send: Send the statistics to the Collect View
    server
154 */
155 void collect_common_send(void)
156 {
157   static uint8_t seqno;
158   struct {
159     uint8_t seqno;
160     uint8_t for_alignment;
161     struct collect_view_data_msg msg;
162   } msg;
163 
164   uint16_t parent_etx;
165   uint16_t rtmetric;
166   uint16_t num_neighbors;
167   uint16_t beacon_interval;
168   rpl_parent_t *preferred_parent;
169   linkaddr_t parent;
170   rpl_dag_t *dag;
171 
172   if(client_conn == NULL) {
173     /* Not setup yet */
174     return;
175   }
176   memset(&msg, 0, sizeof(msg));
177   seqno++;
178   if(seqno == 0) {
179     /* Wrap to 128 to identify restarts */
180     seqno = 128;
```

```
181   }
182   msg.seqno = seqno;
183
184   linkaddr_copy(&parent, &linkaddr_null);
185   parent_etx = 0;
186
187   /* Let's suppose we have only one instance */
188   dag = rpl_get_any_dag();
189   if(dag != NULL) {
190     preferred_parent = dag->preferred_parent;
191     if(preferred_parent != NULL) {
192       uip_ds6_nbr_t *nbr;
193       nbr =
uip_ds6_nbr_lookup(rpl_get_parent_ipaddr(preferred_parent));
194       if(nbr != NULL) {
195         /* Use parts of the IPv6 address as the parent address,
in reversed byte order. */
196         parent.u8[LINKADDR_SIZE - 1] = nbr-
>ipaddr.u8[sizeof(uip_ipaddr_t) - 2];
197         parent.u8[LINKADDR_SIZE - 2] = nbr-
>ipaddr.u8[sizeof(uip_ipaddr_t) - 1];
198         parent_etx = rpl_get_parent_rank((uip_lladdr_t *)
uip_ds6_nbr_get_ll(nbr)) / 2;
199       }
200     }
201     rtmetric = dag->rank;
202     beacon_interval = (uint16_t) ((2L << dag->instance-
>dio_intcurrent) / 1000);
203     num_neighbors = uip_ds6_nbr_num();
204   } else {
205     rtmetric = 0;
206     beacon_interval = 0;
207     num_neighbors = 0;
208   }
209
210   collect_view_construct_message(&msg.msg, &parent,
211                                  parent_etx, rtmetric,
212                                  num_neighbors,
beacon_interval);
213
214   uip_udp_packet_sendto(client_conn, &msg, sizeof(msg),
215                         &server_ipaddr,
UIP_HTONS(UDP_SERVER_PORT));
216 }
217
218 /*---------------------------------------------------------------
-------------*/
219
220 /*
221 * collect_common_net_init: sets the serial line input for
Collection View
222 */
223 void collect_common_net_init(void)
224 {
225 #if CONTIKI_TARGET_Z1
226   uart0_set_input(serial_line_input_byte);
```

```
#else
  uart1_set_input(serial_line_input_byte);
#endif
  serial_line_init();
}

/*---------------------------------------------------------------------------*/

/*
* print_local_addresses: prints the current IPv6 addresses of the device
*/
static void print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Client IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
        uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}

/*---------------------------------------------------------------------------*/

/*
* set_global_address: sets the IPv6 addresses of the device
* and the Collection View server
*/
static void set_global_address(void)
{
  uip_ipaddr_t ipaddr;


  /* Build the IPv6 Address */
  uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0, 0);
  /* Set the last 64 bits based on the MAC Address */
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  /* Add the unicast address to the list */
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

  /* Set the Collection View server address */
  uip_ip6addr(&server_ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0, 1);

```

```
279 }
280
281 /*---------------------------------------------------------------
    -------------*/
282
283 PROCESS_THREAD(collection_view_client_process, ev, data)
284 {
285   PROCESS_BEGIN();
286
287   PROCESS_PAUSE();
288
289   set_global_address();
290
291   #if DEBUG_SENDER == 1
292   PRINTF("UDP client process started\n");
293   print_local_addresses();
294   #endif
295
296   /* new connection with remote host */
297   client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
298   udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
299
300   #if DEBUG_SENDER == 1
301   PRINTF("Created a connection with the server ");
302   PRINT6ADDR(&client_conn->ripaddr);
303   PRINTF(" local/remote port %u/%u\n",
304         UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn-
    >rport));
305   #endif
306
307   while(1) {
308     PROCESS_YIELD();
309     if(ev == tcpip_event) {
310       tcpip_handler();
311     }
312   }
313
314   PROCESS_END();
315 }
316
317 /*---------------------------------------------------------------
    -------------*/
318
319 /*
320 * receiver: callback function to handle incoming udp packets
    from
321 * the sink server. It is not really used in this scenario
322 */
323 static void receiver(struct simple_udp_connection *c,
324          const uip_ipaddr_t *sender_addr,
325          uint16_t sender_port,
326          const uip_ipaddr_t *receiver_addr,
327          uint16_t receiver_port,
328          const uint8_t *data,
329          uint16_t datalen)
330 {
```

```
331   printf("Data received on port %d from port %d with length
      %d\n",
332           receiver_port, sender_port, datalen);
333   }
334
335   /*---------------------------------------------------------------
      -------------*/
336
337   /*
338   * unicast_sender_process: It sends the unicast packet to the
      sink
339   * server. But first it has to lookup the service with the
      servreg_hack
340   * app
341   */
342   PROCESS_THREAD(unicast_sender_process, ev, data)
343   {
344     static struct etimer periodic_timer;
345
346
347     PROCESS_BEGIN();
348
349     /*
350     * Starts the servreg_hack app that is used to discover the
      sink
351     * service server address
352     */
353     servreg_hack_init();
354
355     /*
356     * sets the IPv6 addresses of the device and the Collection
      View
357     * server
358     */
359     set_global_address();
360
361     /*
362     * This function registers the UDP connection with the sink
      server
363     * and attaches the receiver() function as a callback.
364     * UDP_PORT is used as local and remote port.
365     */
366     simple_udp_register(&unicast_connection, UDP_PORT,
367                         NULL, UDP_PORT, receiver);
368
369     /* We set the timer to SEND_INTERVAL */
370     etimer_set(&periodic_timer, SEND_INTERVAL);
371
372     while(1) {
373
374       /*
375       * Wait for the interval to start sending the message, but
      before
376       * reset it again so that in the next interval we cand send
      another
377       * message
378       */
```

```
379     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
380     etimer_reset(&periodic_timer);
381
382     /* It looks for the server address */
383     addr = servreg_hack_lookup(SERVICE_ID);
384     if(addr != NULL) {
385       static unsigned int message_number;
386       char buf[24];
387
388       #if DEBUG_SENDER == 1
389       printf("Sending unicast to ");
390       uip_debug_ipaddr_print(addr);
391       printf("\n");
392       #endif
393
394       /* The message to send with the sequence number */
395       sprintf(buf, "TFG Jose Antonio %d", message_number);
396
397       /* If we reach the max sequence number we reset it */
398       if (message_number == MAX_MESSAGE_NUMBER) {
399         message_number = 0;
400       }
401       else {
402         message_number++;
403       }
404
405       /* It sends it to the sink server */
406       simple_udp_sendto(&unicast_connection, buf, strlen(buf) +
    1, addr);
407
408     } else {
409       #if DEBUG_SENDER == 1
410       printf("Service %d not found\n", SERVICE_ID);
411       #endif
412     }
413   }
414
415   PROCESS_END();
416 }
417
418 /*---------------------------------------------------------------
    -------------*/
419
```

project-conf.h:

```
/*
* Copyright (c) 2016, Jose Antonio Caballero Martos.
* Universidad de Sevilla
*
* Copyright (c) 2015, Swedish Institute of Computer Science.
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. Neither the name of the Institute nor the names of its contributors
*    may be used to endorse or promote products derived from this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*/
```

```
32
33  #ifndef PROJECT_CONF_H_
34  #define PROJECT_CONF_H_
35
36
37
38  #undef NBR_TABLE_CONF_MAX_NEIGHBORS
39  #undef UIP_CONF_MAX_ROUTES
40
41  #ifdef TEST_MORE_ROUTES
42  /* configure number of neighbors and routes */
43  #define NBR_TABLE_CONF_MAX_NEIGHBORS     2
44  #define UIP_CONF_MAX_ROUTES    2
45  #endif /* TEST_MORE_ROUTES */
46
47
48  #undef NETSTACK_CONF_RDC
49  #define NETSTACK_CONF_RDC     nullrdc_driver
50
51  #undef NETSTACK_CONF_MAC
52  #define NETSTACK_CONF_MAC nullmac_driver
53
54  #undef NONCORESEC_KEY
55  #define NONCORESEC_KEY { 0x00 , 0x01 , 0x02 , 0x03 , \
56                           0x04 , 0x05 , 0x06 , 0x07 , \
57                           0x08 , 0x09 , 0x0A , 0x0B , \
58                           0x0C , 0x0D , 0x0E , 0x0F }
59
60  #undef ADAPTIVESEC_CONF_UNICAST_SEC_LVL
61  #define ADAPTIVESEC_CONF_UNICAST_SEC_LVL 1
62  #undef ADAPTIVESEC_CONF_BROADCAST_SEC_LVL
63  #define ADAPTIVESEC_CONF_BROADCAST_SEC_LVL 1
64  #undef LLSEC802154_CONF_USES_AUX_HEADER
65  #define LLSEC802154_CONF_USES_AUX_HEADER 0
66  #undef NETSTACK_CONF_LLSEC
67  #define NETSTACK_CONF_LLSEC noncoresec_driver
68
69  #include "net/llsec/adaptivesec/noncoresec-autoconf.h"
70
71
72  /* Define as minutes */
73  #define RPL_CONF_DEFAULT_LIFETIME_UNIT   60
74
75  /* 10 minutes lifetime of routes */
76  #define RPL_CONF_DEFAULT_LIFETIME        10
77
78  #define RPL_CONF_DEFAULT_ROUTE_INFINITE_LIFETIME 1
79
80  /* Save some ROM */
81  #undef UIP_CONF_TCP
82  #define UIP_CONF_TCP                  0
83
84  #undef PROCESS_CONF_NO_PROCESS_NAMES
85  #define PROCESS_CONF_NO_PROCESS_NAMES 1
86
87  #undef SICSLOWPAN_CONF_FRAG
```

```
88  #define SICSLOWPAN_CONF_FRAG           0
89
90  #endif /* PROJECT_CONF_H_ */
```

Makefile:

```
1   #  Copyright (c) 2016, Jose Antonio Caballero Martos.
2   # Universidad de Sevilla
3
4   CONTIKI=../../../..
5   APPS =  powertrace collect-view servreg-hack reboot-app
6   CONTIKI_PROJECT = udp-sender udp-sink
7   PROJECT_SOURCEFILES += collect-common.c
8
9   CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
10
11  ifdef PERIOD
12  CFLAGS=-DPERIOD=$(PERIOD)
13  endif
14
15  all: $(CONTIKI_PROJECT)
16
17  CONTIKI_WITH_IPV6 = 1
18  include $(CONTIKI)/Makefile.include
```

udp-sink:

```
/*
* Copyright (c) 2016, Jose Antonio Caballero Martos.
* Universidad de Sevilla
*
* Copyright (c) 2010, Swedish Institute of Computer Science.
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or
without
* modification, are permitted provided that the following
conditions
* are met:
* 1. Redistributions of source code must retain the above
copyright
*    notice, this list of conditions and the following
disclaimer.
* 2. Redistributions in binary form must reproduce the above
copyright
*    notice, this list of conditions and the following
disclaimer in the
*    documentation and/or other materials provided with the
distribution.
* 3. Neither the name of the Institute nor the names of its
contributors
*    may be used to endorse or promote products derived from
this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS
``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR
CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF
* SUCH DAMAGE.
*
* This file is part of the Contiki operating system.
*
```

```
34  */
35
36  #include "contiki.h"
37  #include "contiki-lib.h"
38  #include "contiki-net.h"
39  #include "net/ip/uip.h"
40  #include "net/rpl/rpl.h"
41  #include "net/linkaddr.h"
42
43  #include "net/netstack.h"
44  #include "dev/button-sensor.h"
45  #include "dev/serial-line.h"
46  #if CONTIKI_TARGET_Z1
47  #include "dev/uart0.h"
48  #else
49  #include "dev/uart1.h"
50  #endif
51  #include <stdio.h>
52  #include <stdlib.h>
53  #include <string.h>
54  #include <ctype.h>
55  #include "collect-common.h"
56  #include "collect-view.h"
57
58  #define DEBUG DEBUG_PRINT
59  #include "net/ip/uip-debug.h"
60
61  /* Contiki apps */
62  #include "reboot-app.h"
63  #include "servreg-hack.h"
64
65  /* Sink server UDP Port and Service ID */
66  #define UDP_PORT 1234
67  #define SERVICE_ID 190
68
69  #if 0
70  #define SEND_INTERVAL         (10 * CLOCK_SECOND)
71  #define SEND_TIME         SEND_INTERVAL
72  #endif
73
74  #define UIP_IP_BUF   ((struct uip_ip_hdr
    *)&uip_buf[UIP_LLH_LEN])
75
76  #define UDP_COLLECT_CLIENT_PORT 8775
77  #define UDP_COLLECT_SERVER_PORT 5688
78
79  /*
80  * Set this to 1 in order to show the printf() output
81  * Set to another value to save ROM
82  */
83  #define DEBUG_SINK 1
84
85  /* Unicast connections*/
86  static struct simple_udp_connection unicast_connection;
87
88  /* Collect View */
89  static struct uip_udp_conn *server_conn;
```

```
90
91  /*---------------------------------------------------------------
    -------------*/
92
93  /*
94  *
95  * collection_view_server_process: will receive the statistics of
    the
96  * sender sensor
97  *
98  * unicast_receiver_process: will receive the unicast packets to
    the sink
99  *
100 */
101 PROCESS(unicast_receiver_process, "Unicast receiver example
    process");
102 PROCESS(collection_view_server_process, "UDP server process");
103 AUTOSTART_PROCESSES(&unicast_receiver_process,
    &collection_view_server_process, &collect_common_process);
104
105 /*---------------------------------------------------------------
    -------------*/
106
107 /*
108 * These functions are common between the client and the server,
    but
109 * there is no real need to implement them in the sink server
110 */
111 void collect_common_set_sink(void)
112 {
113 }
114
115 void collect_common_net_print(void)
116 {
117   printf("Sink\n");
118 }
119
120
121 void collect_common_send(void)
122 {
123   /* Server never sends */
124 }
125
126 /*---------------------------------------------------------------
    -------------*/
127
128 /*
129 * collect_common_net_init: sets the serial line input for
    Collection View
130 */
131 void collect_common_net_init(void)
132 {
133 #if CONTIKI_TARGET_Z1
134   uart0_set_input(serial_line_input_byte);
135 #else
136   uart1_set_input(serial_line_input_byte);
```

```
137 #endif
138   serial_line_init();
139 }
140 
141 /*---------------------------------------------------------------
    -------------*/
142 
143 /*
144 * tcpip_handler: handles the incoming packet from the sender
    sensor
145 * and gathers the information for the Collection View app
146 */
147 static void tcpip_handler(void)
148 {
149   uint8_t *appdata;
150   linkaddr_t sender;
151   uint8_t seqno;
152   uint8_t hops;
153 
154   if(uip_newdata()) {
155     appdata = (uint8_t *)uip_appdata;
156     sender.u8[0] = UIP_IP_BUF->srcipaddr.u8[15];
157     sender.u8[1] = UIP_IP_BUF->srcipaddr.u8[14];
158     seqno = *appdata;
159     hops = uip_ds6_if.cur_hop_limit - UIP_IP_BUF->ttl + 1;
160     collect_common_recv(&sender, seqno, hops,
161                         appdata + 2, uip_datalen() - 2);
162   }
163 }
164 
165 
166 /*---------------------------------------------------------------
    -------------*/
167 
168 /*
169 * print_local_addresses: prints the current IPv6 addresses of
    the device
170 */
171 static void print_local_addresses(void)
172 {
173   #if DEBUG_SINK == 1
174 
175   int i;
176   uint8_t state;
177 
178   PRINTF("Server IPv6 addresses: ");
179   for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
180     state = uip_ds6_if.addr_list[i].state;
181     if(state == ADDR_TENTATIVE || state == ADDR_PREFERRED) {
182       PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
183       PRINTF("\n");
184       /* hack to make address "final" */
185       if (state == ADDR_TENTATIVE) {
186         uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
187       }
188     }
189   }
```

```
190
191 #endif
192 }
193
194 /*---------------------------------------------------------------
    -------------*/
195
196 /*
197 * set_global_address: sets the IPv6 addresses of the device
198 */
199 static uip_ipaddr_t * set_global_address(void)
200 {
201   static uip_ipaddr_t ipaddr;
202
203   // Build the IPv6 Address
204   uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0,
    0);
205   // Set the last 64 bits based on the MAC Address
206   uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
207   // We add the unicast address to the list
208   uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
209
210   return &ipaddr;
211 }
212
213 /*---------------------------------------------------------------
    -------------*/
214
215 static void receiver(struct simple_udp_connection *c,
216          const uip_ipaddr_t *sender_addr,
217          uint16_t sender_port,
218          const uip_ipaddr_t *receiver_addr,
219          uint16_t receiver_port,
220          const uint8_t *data,
221          uint16_t datalen)
222 {
223   printf("Data received from ");
224   uip_debug_ipaddr_print(sender_addr);
225   printf(" on port %d from port %d with length %d: '%s'\n",
226          receiver_port, sender_port, datalen, data);
227 }
228
229 /*---------------------------------------------------------------
    -------------*/
230
231 PROCESS_THREAD(collection_view_server_process, ev, data)
232 {
233   uip_ipaddr_t ipaddr;
234   struct uip_ds6_addr *root_if;
235
236   PROCESS_BEGIN();
237
238   PROCESS_PAUSE();
239
240   #if DEBUG_SINK == 1
241   PRINTF("UDP server started\n");
```

```
242   #endif
243
244 #if UIP_CONF_ROUTER
245   uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0, 0,
    1);
246   /* uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr); */
247   uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
248   root_if = uip_ds6_addr_lookup(&ipaddr);
249   if(root_if != NULL) {
250     rpl_dag_t *dag;
251     dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t
    *)&ipaddr);
252     uip_ip6addr(&ipaddr, UIP_DS6_DEFAULT_PREFIX, 0, 0, 0, 0, 0,
    0, 0);
253     rpl_set_prefix(dag, &ipaddr, 64);
254     PRINTF("created a new RPL dag\n");
255   } else {
256     PRINTF("failed to create a new RPL DAG\n");
257   }
258 #endif /* UIP_CONF_ROUTER */
259
260   print_local_addresses();
261
262   server_conn = udp_new(NULL,
    UIP_HTONS(UDP_COLLECT_CLIENT_PORT), NULL);
263   udp_bind(server_conn, UIP_HTONS(UDP_COLLECT_SERVER_PORT));
264
265   #if DEBUG_SINK == 1
266   PRINTF("Created a server connection with remote address ");
267   PRINT6ADDR(&server_conn->ripaddr);
268   PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn-
    >lport),
269          UIP_HTONS(server_conn->rport));
270   #endif
271
272   while(1) {
273     PROCESS_YIELD();
274     if(ev == tcpip_event) {
275       tcpip_handler();
276     } else if (ev == sensors_event && data == &button_sensor) {
277       PRINTF("Initiaing global repair\n");
278       rpl_repair_root(RPL_DEFAULT_INSTANCE);
279     }
280   }
281
282   PROCESS_END();
283 }
284
285 /*---------------------------------------------------------------
    -------------*/
286
287 PROCESS_THREAD(unicast_receiver_process, ev, data)
288 {
289   uip_ipaddr_t *ipaddr;
290
291   PROCESS_BEGIN();
292
```

```
293   servreg_hack_init();
294
295   ipaddr = set_global_address();
296
297   servreg_hack_register(SERVICE_ID, ipaddr);
298
299   simple_udp_register(&unicast_connection, UDP_PORT,
300                       NULL, UDP_PORT, receiver);
301
302   while(1) {
303     PROCESS_WAIT_EVENT();
304   }
305   PROCESS_END();
306 }
307
308 /*---------------------------------------------------------------------
    --------------*/
309
310
311
```

collect-common.c:

```
/*
* Copyright (c) 2010, Swedish Institute of Computer Science.
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or
without
* modification, are permitted provided that the following
conditions
* are met:
* 1. Redistributions of source code must retain the above
copyright
*    notice, this list of conditions and the following
disclaimer.
* 2. Redistributions in binary form must reproduce the above
copyright
*    notice, this list of conditions and the following
disclaimer in the
*    documentation and/or other materials provided with the
distribution.
* 3. Neither the name of the Institute nor the names of its
contributors
*    may be used to endorse or promote products derived from
this software
*    without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS
``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR
CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF
* SUCH DAMAGE.
*
*/

/**
* \file
```

```
33  *         Example of how the collect primitive works.
34  * \author
35  *         Adam Dunkels <adam@sics.se>
36  */
37
38  #include "contiki.h"
39  #include "lib/random.h"
40  #include "net/netstack.h"
41  #include "dev/serial-line.h"
42  #include "dev/leds.h"
43  #include "collect-common.h"
44
45  #include <stdio.h>
46  #include <string.h>
47  #include <ctype.h>
48
49  static unsigned long time_offset;
50  static int send_active = 1;
51
52  #ifndef PERIOD
53  #define PERIOD 60
54  #endif
55  #define RANDWAIT (PERIOD)
56
57  /*---------------------------------------------------------------
    -------------*/
58  PROCESS(collect_common_process, "collect common process");
59  AUTOSTART_PROCESSES(&collect_common_process);
60  /*---------------------------------------------------------------
    -------------*/
61  static unsigned long
62  get_time(void)
63  {
64    return clock_seconds() + time_offset;
65  }
66  /*---------------------------------------------------------------
    -------------*/
67  static unsigned long
68  strtolong(const char *data) {
69    unsigned long value = 0;
70    int i;
71    for(i = 0; i < 10 && isdigit(data[i]); i++) {
72      value = value * 10 + data[i] - '0';
73    }
74    return value;
75  }
76  /*---------------------------------------------------------------
    -------------*/
77  void
78  collect_common_set_send_active(int active)
79  {
80    send_active = active;
81  }
82  /*---------------------------------------------------------------
    -------------*/
83  void
```

```
84  collect_common_recv(const linkaddr_t *originator, uint8_t seqno,
    uint8_t hops,
85                      uint8_t *payload, uint16_t payload_len)
86  {
87    unsigned long time;
88    uint16_t data;
89    int i;
90
91    printf("%u", 8 + payload_len / 2);
92    /* Timestamp. Ignore time synch for now. */
93    time = get_time();
94    printf(" %lu %lu 0", ((time >> 16) & 0xffff), time & 0xffff);
95    /* Ignore latency for now */
96    printf(" %u %u %u %u",
97           originator->u8[0] + (originator->u8[1] << 8), seqno,
    hops, 0);
98    for(i = 0; i < payload_len / 2; i++) {
99      memcpy(&data, payload, sizeof(data));
100     payload += sizeof(data);
101     printf(" %u", data);
102   }
103   printf("\n");
104   leds_blink();
105 }
106 /*---------------------------------------------------------------
    -------------*/
107 PROCESS_THREAD(collect_common_process, ev, data)
108 {
109   static struct etimer period_timer, wait_timer;
110   PROCESS_BEGIN();
111
112   collect_common_net_init();
113
114   /* Send a packet every 60-62 seconds. */
115   etimer_set(&period_timer, CLOCK_SECOND * PERIOD);
116   while(1) {
117     PROCESS_WAIT_EVENT();
118     if(ev == serial_line_event_message) {
119       char *line;
120       line = (char *)data;
121       if(strncmp(line, "collect", 7) == 0 ||
122          strncmp(line, "gw", 2) == 0) {
123         collect_common_set_sink();
124       } else if(strncmp(line, "net", 3) == 0) {
125         collect_common_net_print();
126       } else if(strncmp(line, "time ", 5) == 0) {
127         unsigned long tmp;
128         line += 6;
129         while(*line == ' ') {
130           line++;
131         }
132         tmp = strtolong(line);
133         time_offset = clock_seconds() - tmp;
134         printf("Time offset set to %lu\n", time_offset);
135       } else if(strncmp(line, "mac ", 4) == 0) {
136         line +=4;
137         while(*line == ' ') {
```

```
138           line++;
139         }
140         if(*line == '0') {
141           NETSTACK_RDC.off(1);
142           printf("mac: turned MAC off (keeping radio on): %s\n",
143                  NETSTACK_RDC.name);
144         } else {
145           NETSTACK_RDC.on();
146           printf("mac: turned MAC on: %s\n", NETSTACK_RDC.name);
147         }
148
149       } else if(strncmp(line, "~K", 2) == 0 ||
150                 strncmp(line, "killall", 7) == 0) {
151         /* Ignore stop commands */
152       } else {
153         printf("unhandled command: %s\n", line);
154       }
155     }
156     if(ev == PROCESS_EVENT_TIMER) {
157       if(data == &period_timer) {
158         etimer_reset(&period_timer);
159         etimer_set(&wait_timer, random_rand() % (CLOCK_SECOND *
    RANDWAIT));
160       } else if(data == &wait_timer) {
161         if(send_active) {
162           /* Time to send the data */
163           collect_common_send();
164         }
165       }
166     }
167   }
168
169   PROCESS_END();
170 }
171 /*---------------------------------------------------------------------
    -------------*/
```

BIBLIOGRAFÍA

(Referencias comprobadas por última vez el 6 de julio de 2016. No se asegura que el contenido de las mismas siga siendo el mismo pasada esta fecha).

[1] Konrad-Felix Krentz et al. "*6LoWPAN security: adding compromise resilience to the 802.15.4 security sublayer*" Hasso Plattner Institute, septiembre 2013.

[2] Antonio Liñán Colina et al. "IoT in five days" Zolertia, junio 2016.

[3] Lighfoot et al. "*An energy efficient Link-Layer Security Protocol for wireless sensor networks*" Department of Electrical & Computer Engineering, Michigan State University, mayo 2007.

[4] Lander Casado et al. "*ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System*" Department of Computer Science and Engineering, Chalmers University of Technology.

[5] Boyle et al. "*Security Protocols for use with wireless sensor networks*" Department of Electronic and Computer Engineering, University of Limerick.

[6] Borgohain et al. "*Survey of Security and Privacy Issues of Internet of Things*" Department of Instrumentation Engineering, Assam Engineering College 2015.

[7] Ferreira et al. "*Proposal of a secure, deployable and transparent middleware for Internet of Things*" Electrical Engineering Department, University of Brasilia 2014.

[8] Kumar et al. "*A survey on Internet of Things: Security and Privacy Issues*" Department of Computer Engineering SVNIT Surat, marzo 2014.

[9] Zhao et al. "*A survey on the Internet of Things Security*" School of Inf. Science and Engineering, Guangxi University, diciembre 2013.

[10] Ahmed et al. "*An evaluation of security protocols on wireless sensor network*" Helsinki University of Technology.

[11] Konrad-Felix Krentz et al. "*Handling Reboots and Mobility in 802.15.4 Security*" Hasso Plattner Institute 2015.

[12] Shahid Raza et al. "*Securing Communication in 6LoWPAN with Compressed IPsec*" Swedish Institute of Computer Science and Lancaster University School of Computing and Communications.

[13] Vladislav Perelman "*Security in IPv6-enabled Wireless Sensor Networks: An Implementation of TLS/DTLS for the Contiki Operating System*" School of Engineering and Science, Jacobs University, junio 2012.

[14] Pablo Puñal Pereira "*Enabling IPsec as optional security level for 6LoWPAN on Contiki*" Lulea University of Technology.

[15] Christian Schumacher, "*6LoWPAN Security Considerations*" IETF, noviembre 2005.

[16] Satish Phakade Pawar "*Home Automation with 6LoWPAN*" ASM's IBMR, Chichwad.

[17] S. Park et al. "*IPv6 over Low Power WPAN Security Analysis*" Samsung Electronics.

[18] Socrates Varakliotis et al. "Universal Integration of the Internet of Things through an IPv6-based Service Oriented Architecture enabling heterogeneous components interoperability" University College London, enero 2011.

[19] Jen Sarto, "*ZigBee VS 6LoWPAN for Sensor Networks*" LSR: http://www.lsr.com/white-papers/zigbee-vs-6lowpan-for-sensor-networks

[20] Thomas Watteyne, "*6LoWPAN*" OpenWSN Atlassian: http://openwsn.atlassian.net/wiki/display/OW/6LoWPAN

[21] Jonathan Hui et al., "*Compression Format for IPv6 Datagrams in 6LoWPAN networks*" IETF San Francisco, marzo 2009.

[22] Raheem Beyah et al. "Computer and Network Security : Security in Ad-Hoc and Sensor Networks" World Scientific, septiembre 2009.

[23] "*Get Started with Contiki*" Contiki: http://www.contiki-os.org/start.html

[24] Pietro Gonizzi, "*Hands on Contiki OS and Cooja Simulator*" University of Parma, septiembre 2013.

[25] Thiemo Voigt, "*Contiki Cooja Crash Course*" Swedish Institute of Computer Science, julio 2009.

[26] "*IoT Emulation with Cooja*" Department of Computer Science, ISAT Laboratory, University of Western Cape, marzo 2015.

[27] "*Processes in Contiki*" Contiki Wiki: http://github.com/contiki-os/contiki/wiki/Processes

[28] "*Contiki MAC and Radio Duty Cycling protocols*" Contiki Wiki: http://github.com/contiki-os/contiki/wiki/Change-mac-or-radio-duty-cycling-protocols

[29] "*RPL Border Router in Contiki*" University of Southern California: http://anrg.usc.edu/contiki/index.php/RPL_Border_Router

[30] "*Contiki netstack*" University of Southern California: http://anrg.usc.edu/contiki/index.php/Network_Stack

[31] "*The Contiki netstack*" Thingsquare.

[32] "*Contiki Apps*" Zolertia: http://zolertia.sourceforge.net/wiki/index.php/Mainpage:Contiki_apps

[33] Jorge Grajal, Edmundo Monteiro, Jorge Sá Silva, "*Enabling network-layer security on IPv6 wireless sensor networks*": http://www.academia.edu/2805780/Enabling_network-layer_security_on_IPv6_wireless_sensor_networks

[34] "*Mitigate Risk, Simplify, Compliance, and Build Trust*" Cisco IoT System Security.

[35] Chema Alonso et al. "*Scope, scale and risk like never before: Securing the Internet of Things*" Telefónica y ElevenPaths, enero 2016.

[36] Kevin Ashton, "*That 'Internet of Things' Thing*" *artículo de RFID Journal*, 2009.

[37] Dave Evans, "*The Internet of Things. How the Next Evolution of the Internet Is Changing Everything*", Cisco, Abril 2011.

[38]"*How Many Things Are Currently Connected To The "Internet of Things*", Forbes, 7 enero 2013: http://www.forbes.com/sites/quora/2013/01/07/how-many-things-are-currently-connected-to-the-internet-of-things-iot/#2944fd7e6379

[39] "*Cisco IoT System Security: Mitigate Risk, Simplify, Compliance, and Build Trust. Cisco White Paper*" Global Market Insite (GMI), a division of Lightspeed Research, from a Cisco-sponsored study 2015.

[40] "*Securing the Internet of Things*" SANS Institute, Cisco 2015.

[41] "*¿Qué pasaría si Internet se quedara sin espacio? En realidad, es algo que ya está ocurriendo.*" Google: http://www.google.com/intl/es/ipv6/

[42] "*Estadísticas de adopción de IPv6*" Google: http://www.google.com/intl/es/ipv6/statistics.html

[43] Google's IPv6 Traffic Hits 5% Globally, 28% in Belgium, 12% in USA and Germany" Internet Society: http://www.internetsociety.org/deploy360/blog/2014/12/googles-ipv6-traffic-hits-5-globally-28-in-belgium-12-in-usa-and-germany/

[44] "*IoT Devices and Local Networks*" Micrium: https://www.micrium.com/iot/devices/

[45] "*Securing the Internet of Things Opportunity: Putting Cybersecurity at the Heart of the IoT*". Capgemini Consulting, Sogeti High Tech.

[46] Alnoor M. Peermohamed, "*Cyber Security: The Thorn That Can Cripple The IoT*" NextBigWhat: http://www.nextbigwhat.com/cyber-security-iot-297/

[47] Joe Hanson, "*The 10 Challenges of Securing IoT Communications*" Pubnub, 4 de mayo de 2015: https://www.pubnub.com/blog/2015-05-04-10-challenges-securing-iot-communications-iot-security/

[48] Alan Grau, "*What is Really Needed to Secure the Internet of Things?*" Iconlabs: http://www.iconlabs.com/prod/internet-secure-things-%E2%80%93-what-really-needed-secure-internet-things

[49] "*Encryption Libraries for Waspmote Sensor Networks*" Libelium : http://www.libelium.com/products/waspmote/encryption/

[50] S. Thomson et al. "*RFC 4862: IPv6 Stateless Address Autoconfiguration*" Cisco, septiembre 2007

GLOSARIO

WSN: Wireless Sensor Network.

IoT: Internet of Things.

IP: Internet Protocol.

IPv4/IPv6: Internet Protocol version 4/ Internet Protocol version 6

NAT: Network Address Translation.

DHCP: Dynamic Host Configuration Protocol.

DNS: Domain Name System.

M2M: Machine to machine.

CRC: Comprobación de Redundancia Cíclica.

DoS: Denial of Service.

6LoWPAN: IPv6 over Low power Wireless Personal Area Networks.

PAN: Personal Area Network.

FFD: Full Function Device.

RFD: Reduced Function Device.

MAC: Media Access Control.

P2P: peer-to-peer.

AES: Advanced Encryption Standard.

MAC/MIC: Message Authentication Code / Message Integrity Code.

CTR: Counter.

CCM: Counter with CBC-MAC.

TCP: Transmission Control Protocol.

UDP: User Datagram Protocol.

IPsec: Internet Protocol Security.

AH: Authentication Header.

ESP: Encapsulating Security Protocol.

SA: Security Associations.

SSL: Secure Sockets Layer.

TLS: Transport Layer Security.

RAM: Random Access Memory.

CPU: Central Processing Unit.

JNI: Java Native Interfaces.

RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks.