INTERNET DE LAS COSAS

Antonio Liñán Colina, Alvaro Vives, Antoine Bagula, Marco Zennaro and Ermanno Pietrosemoli

INTERNET DE LAS COSAS

Editado por

Marco Zennaro y Ermanno Pietrosemoli

Autores

Antonio Liñán Colina
Alvaro Vives
Antoine Bagula
Marco Zennaro
Ermanno Pietrosemoli

Traducción
Lourdes González Valera

Octubre 2015

Descargo de responsabilidad

Derechos de autor

Agradecimientos

La traducción al español del libro “Internet of Things” fue financiada por el capítulo latinoamericano de la Internet Society (ISOC). Nuestros más sinceros agradecimientos.

Internet de las Cosas (IoT)

Partiendo de una compleja red que conecta millones de dispositivos y personas en una infraestructura de multi-tecnología, multi-protocolo y multi-plataforma, la visión principal de Internet de las cosas (**IoT**) es la creación de un mundo inteligente donde lo real, lo digital y lo virtual converjan para crear un entorno inteligente que proporcione más inteligencia a la energía, la salud, el transporte, las ciudades, la industria, los edificios y muchas otras áreas de la vida diaria.

La expectativa es la de interconectar millones de islas de redes inteligentes que habiliten el acceso a la información no sólo en cualquier momento y dondequiera, sino también usando cualquier cosa y por parte de cualquier persona, idealmente a través de cualquier ruta, red, y cualquier servicio. Esto se logrará si todos los objetos que manipulamos diariamente se dotan de sensores capaces de detectarlos, identificarlos o ubicar su posición, con una dirección IP que los convierta en objetos inteligentes capaces de comunicarse no sólo con otros objetos inteligentes, sino con seres humanos, en la expectativa de alcanzar ciertas áreas que serían inaccesibles sin los avances hechos por las tecnologías de sensores, identificación y posicionamiento.

Estos objetos inteligentes pueden ser globalmente identificados, interpelados, y al mismo tiempo pueden descubrir e interactuar con entidades externas a través de seres humanos, computadores u otros objetos inteligentes. Los objetos inteligentes, a su vez, pueden adquirir inteligencia por medio de la toma o habilitación de decisiones relacionadas con el entorno y aprovechar los canales de comunicación disponibles para dar información sobre sí mismos a la vez que acceden a la información acumulada por otros objetos inteligentes.

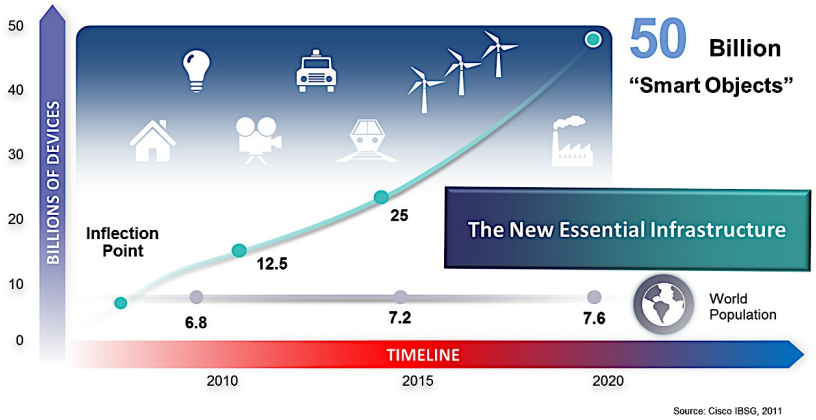

Figura 1. Dispositivos conectados por Internet y la evolución futura (Fuente: Cisco, 2011)

Como se muestra en la figura 1, **IoT** es la nueva estructura que, según las predicciones, en el 2020 conectará 50 millardos de objetos inteligentes cuando la población mundial alcance los 7,6 millardos. Como sugiere la ITU (Unión Internacional de Telecomunicaciones), esta estructura fundamental se construirá alrededor de una arquitectura multicapas en la cual los objetos inteligentes se usarán para prestar diferentes servicios a través de las cuatro capas principales

representadas en la figura 2: una capa del dispositivo, una capa de red, una capa de soporte y una de aplicación.

La capa del dispositivo contiene los dispositivos (sensores, actuadores, dispositivos RFID) y pasarelas (*gateways*) que se usan para recolectar las lecturas del sensor para su procesamiento posterior, mientras que la capa de red proporciona el transporte necesario y las capacidades de red para enrutar los datos de IoT a los sitios de procesamiento. La capa de soporte es una capa intermedia (*middleware*) que sirve para esconder la complejidad de las capas inferiores a la capa de aplicación y para dar servicios específicos y genéricos tales como almacenamiento bajo formas diferentes (sistemas de manejo de base de datos y/o sistemas de computación en la nube), así como otros servicios, por ejemplo traducción.

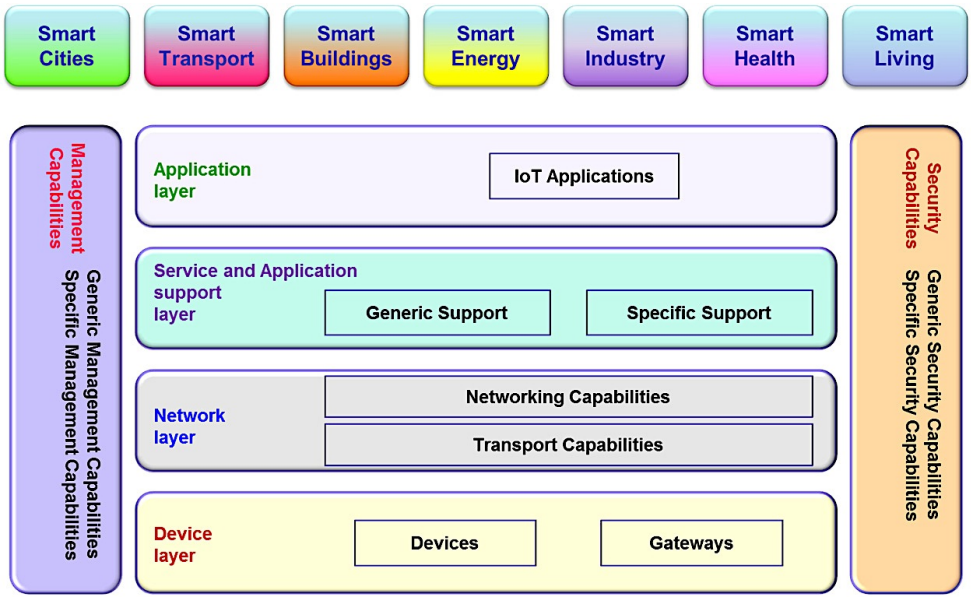

Figura 2. Arquitectura de Capas de IoT (Fuente: ITU-T)

Como se describe en la Figura 3, la IoT puede concebirse como una estructura que ofrece servicios de aplicación habilitados por varias tecnologías. Sus servicios de aplicación permiten la dotación de inteligencia a ciudades, servicios de transporte, edificios, manejo de la energía, de la industria, y de la salud, apoyándose en diferentes tecnologías como sensores, nanoelectrónica, redes de sensores inalámbricos (*wireless sensor network*: **WSN**), identificación de radio frecuencia (**RFID**), localización, almacenamiento y nube. Los sistemas y aplicaciones de la IoT están diseñados para proporcionar seguridad, privacidad, protección, integridad, confianza, fiabilidad, transparencia, anonimato, y están sujetos a limitaciones éticas.

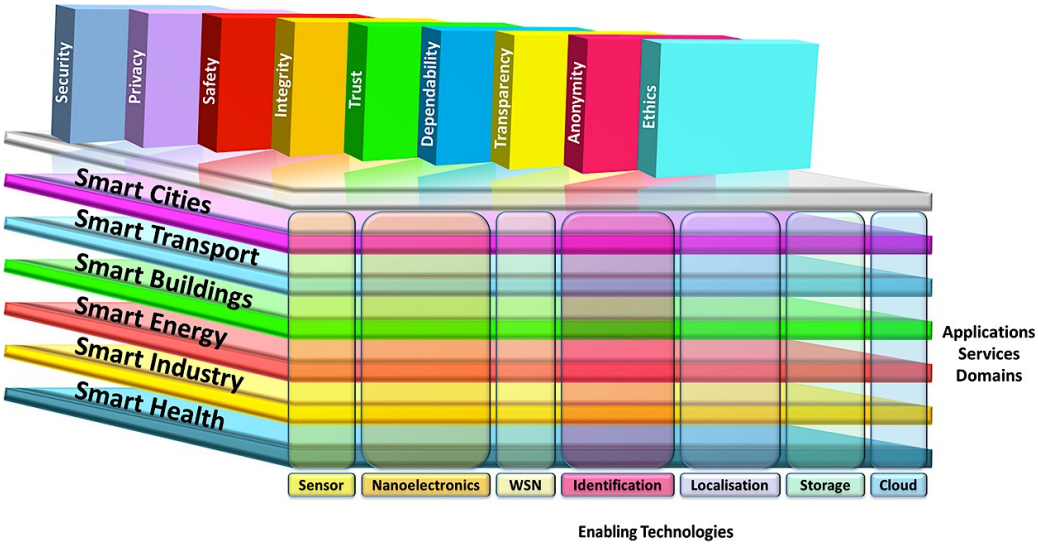

Figura 3. Vista tridimensional de IoT: (Fuente: [1])

Los expertos dicen que estamos yendo hacia lo que podría llamarse una "sociedad de red ubicua", en la cual las redes y los dispositivos interconectados son omnipresentes. La RFID y los sensores inalámbricos prometen un mundo de dispositivos en red interconectados que ofrecen contenido e información relevantes dondequiera que el usuario se encuentre. Todo, desde una llanta a un cepillo de dientes, estará dentro de la cobertura de la red, anunciando el amanecer de una nueva era en la cual la Internet de hoy (de datos y gente) ceda el paso a la Internet de las Cosas del mañana.

Al alba de la revolución de Internet los usuarios quedaban sorprendidos ante la posibilidad de establecer contacto con la gente u obtener información de cualquier parte del mundo y en diferentes zonas horarias. El próximo paso en esta revolución tecnológica (conectar gente a cualquier hora, en cualquier lugar) es el de conectar objetos inanimados a una red de comunicación. La visión que subyace a la Internet de las Cosas permitirá el acceso a la información no sólo a "cualquier hora" y en "cualquier lugar" sino también usando "cualquier cosa". Esto será facilitado con el uso de las WSN y etiquetas RFID para extender el potencial de comunicación y monitoreo de la red de redes, así como para incorporar capacidades de computación en objetos y actividades de uso diario como afeitadoras, zapatos, embalaje.

Las WSN (redes de sensores inalámbricos) son una forma precoz de redes ubicuas de información y comunicación. Son uno de los bloques que constituyen la Internet de las Cosas.

Redes de Sensores Inalámbricos

Una Red de Sensores Inalámbricos (**WSN**) es una red que se auto-configura, formada de pequeños nodos sensores (llamados motas: *mote* en inglés denota el pequeño tamaño, como el de una mota de polvo: N del T) que se comunican entre ellos por señales de radio, y desplegados en gran cantidad para percibir el mundo físico. Los nodos sensores son básicamente pequeños computadores con funciones extremadamente básicas. Consisten de una unidad de procesamiento

con capacidad de computación restringida, una memoria limitada, un dispositivo de comunicación de radio, una fuente de alimentación, y uno o más sensores.

Las motas vienen en diferentes tamaños y formas dependiendo del uso previsto. Pueden ser mínimas si hay que instalarlas en gran cantidad y con poco impacto visual. Pueden tener una fuente de alimentación de baterías recargables si se van a usar en un laboratorio. La integración de estos dispositivos electrónicos mínimos y ubicuos en una gran variedad de contextos garantiza una amplia gama de aplicaciones. Algunos de los campos de aplicación se relacionan con el monitoreo ambiental: agricultura, salud y seguridad.

En una aplicación típica una WSN se despliega en una región con el fin de recolectar datos a través de sus nodos sensores. Estas redes son un puente entre el mundo físico y el mundo virtual. Prometen habilidades inéditas para observar y entender fenómenos del mundo real a gran escala con una resolución espacio-temporal muy fina. Esto puede lograrse porque se despliega un gran número de nodos sensores directamente en el campo donde se realizan los experimentos. Todas las motas o nodos sensores se componen de los cinco elementos que se describen a continuación:

1. Procesador. La tarea de esta unidad es la de procesar la información detectada localmente y la información registrada por otros dispositivos. Al presente, estos procesadores tienen limitaciones en su poder computacional, pero dada la ley de Moore, los dispositivos del futuro serán más pequeños, más potentes y consumirán menos energía. El procesador puede operar en modos diferentes: el modo "dormido" se usa casi todo el tiempo para ahorrar energía; "inactivo" (idle) se usa cuando se esperan datos desde otros nodos sensores, y "activo" se usa cuando se detectan o envían o reciben datos desde otras motas o nodos sensores.

2. Fuente de alimentación. Las motas están hechas para desplegarse en los ambientes más variados que pueden ser remotos u hostiles de tal manera que deberían usar poca energía. Además, tienen poca capacidad de almacenamiento de energía. Es por eso que los protocolos de red deben hacer hincapié en la conservación de la energía. También deberían contar con mecanismos imbuidos que permitan al usuario final la opción de prolongar la vida útil de la red a expensas de disminuir el rendimiento (*throughput*). Los nodos sensores deberían estar provistos de métodos de recolección de energía efectivos, como células solares, de manera que puedan funcionar sin supervisión por meses o años. Las fuentes de energía comunes son baterías recargables, paneles solares y condensadores (incluyendo los supercapacitores).

3. Memoria. Se usa para almacenar programas (instrucciones ejecutadas por el procesador) y datos (datos sin procesar y procesados del nodo sensor).

4. Radio. Los dispositivos de las WSN incluyen un radio inalámbrico de baja velocidad y poca cobertura. La velocidad típica es de 10-100 kbps y la cobertura es de menos de 100 metros. La radiocomunicación es a menudo la actividad que consume más energía, por lo que es una obligación incluir técnicas de ahorro de energía como el "despertar" el dispositivo solo en los momentos en que requiera comunicarse. Se emplean algoritmos y protocolos

sofisticados para resolver los problemas de maximización de vida útil, robustez y tolerancia a fallos.

5. Sensores. Las redes de sensores pueden estar formadas por diferentes tipos de sensores capaces de monitorear una gran variedad de condiciones ambientales. En la Tabla 1 hay una clasificación de las tres principales categorías de sensores basada en el estado de desarrollo (*field-readiness*) y en su capacidad de expansión. Mientras que la capacidad de expansión nos dice si los sensores son lo suficientemente pequeños y baratos como para expandirse a muchos sistemas distribuidos, el estado de desarrollo describe la eficiencia de la ingeniería del sensor en relación con el área donde se instalará. En términos de eficiencia de ingeniería, la Tabla 1 muestra un estado de desarrollo alto para la mayoría de los sensores físicos y para algunos sensores químicos, mientras que la mayoría de sensores químicos caen en la clasificación medio-baja y los biológicos tienen un bajo estado de desarrollo.

Tipo de Sensor	**Parámetro**	**Estado de desarrollo**	**Capacidad de expansión**
Físico	Temperatura	Alto	Alta
	Contenido de humedad	Alto	Alta
	Tasa de flujo, Velocidad de flujo	Alto	Media-Alta
	Presión Alto	Alto	Alta
	Transmisión de luz (Turbidez)	Alto	Alta
Químico	Oxígeno disuelto	Alto	Alta
	Conductividad eléctrica	Alto	Alta
	ph	Alto	Alta
	Potencial de reducción de oxidación	Medio	Alto
	Clase de iones principales (Cl-, Na+)	Bajo-Medio	Baja-Media
	Metales pesados	Bajo	Baja
	Compuestos orgánicos pequeños	Bajo	Baja
	Compuestos orgánicos	Bajo	Baja
Biológico	Microorganismos	Bajo	Baja
	Contaminantes biológicos activos	Bajo	Baja

Algunas aplicaciones comunes son la medición de temperatura, humedad, presión, niveles de ruido, aceleración, humedad del suelo, etc. Debido a limitaciones de ancho de banda y de energía los dispositivos soportan principalmente unidades de pocos datos con una capacidad computacional limitada y una tasa de detección restringida. Algunas aplicaciones precisan detección multimodal de manera que cada dispositivo puede tener varios sensores incluidos.

A continuación se presenta una descripción breve de las características técnicas de las WSN que hacen de ellas una tecnología atractiva.

1. **Conexión inalámbrica**. Las motas se comunican entre sí por radio para intercambiar y procesar los datos recolectados por su unidad de detección. En algunos casos pueden usar otros nodos como relés en cuyo caso la red se define como multi-salto (*multi-hop*). Si los nodos se comunican sólo entre ellos directamente, o con la pasarela, la red se denomina uni-salto (*single-hop*). La conectividad inalámbrica permite recuperar los datos en tiempo real desde puntos de acceso difícil. También hace menos intrusiva la tarea de monitoreo en aquellos sitios donde los cables podrían dificultar la operación normal del ambiente que hay que monitorear. También reduce los costos de instalación: se ha estimado que la tecnología inalámbrica podría reducir en un 80% este costo.

2. **Auto-organización**. Las motas se organizan entre ellas en una red ad-hoc, lo que significa que no necesitan una estructura preexistente. En las WSN, cada mota está programada para realizar un descubrimiento de su entorno, para reconocer cuáles son los nodos a los que puede hablar o escuchar a través de la radio. La capacidad de organizarse espontáneamente en una red hace que sean fáciles de colocar, expandir y mantener, y que sean resistentes al fallo de puntos individuales.

3. **Bajo consumo.** Las WSN pueden instalarse en sitios remotos donde no se dispone de fuentes de alimentación. Dependen, por lo tanto, de energía proveniente de baterías u obtenida a través de técnicas de recolección de energía como paneles solares. Para que puedan funcionar por meses o años, las motas deben usar radios y procesadores de bajo consumo e implementar planes de eficiencia de energía. El procesador debe estar en el modo "dormido" el mayor tiempo posible, y la capa de Acceso al Medio debe diseñarse con esto en mente. Gracias a estas técnicas, las WSN permiten instalaciones duraderas en lugares remotos.

Aplicaciones de las WSN

La integración de estos dispositivos ubicuos y minúsculos en los escenarios más variados asegura una gama amplia de aplicaciones. Entre las más comunes están el monitoreo ambiental, la agricultura, salud, y seguridad. En una aplicación típica, una WSN incluye:

1. Rastreo de movimiento de animales. Una gran red de sensores se ha desplegado para estudiar el efecto de factores microclimáticos en la selección de hábitats de aves marinas

en Great Duck Island, de Maine, EEUU. Los investigadores emplazaron los sensores en las madrigueras y usaron el calor para detectar la presencia de aves anidando, obteniendo de esta mera datos muy valiosos para la investigación biológica. El despliegue fue heterogéneo ya que emplearon nodos en las madrigueras y nodos climáticos.

2. Detección de incendios forestales. Puesto que los nodos sensores pueden ser emplazados estratégicamente en un bosque, pueden informar al usuario final sobre el punto de origen del fuego antes de que este se vuelva incontrolable. Los investigadores de la Universidad de California en Berkeley demostraron la factibilidad de la tecnología de nodos sensores en casos de incendio con su aplicación FireBug.

3. Detección de inundaciones. Un ejemplo de esto es el sistema ALERT instalado en los EUA. Este utiliza sensores que detectan la cantidad de lluvia, nivel del agua y condiciones climáticas. Estos sensores proporcionan información a una base de datos centralizada.

4. Investigación geofísica. Un grupo de investigadores de Harvard instalaron una red de sensores en un volcán activo en Sudamérica para monitorear actividad sísmica y condiciones similares relacionadas con la erupción de volcanes.

5. La aplicación de las WSN en la agricultura incluye el monitoreo detallado de la agricultura y las condiciones que afectan las cosechas y el ganado. Muchos de los problemas para el manejo de las granjas para maximizar la producción y a la vez alcanzar objetivos ecológicos pueden resolverse sólo a través de una cantidad de datos pertinentes. Las WSN pueden también usarse en el control del comercio, particularmente para la preservación de productos cuyo mantenimiento depende de condiciones controladas (temperatura, humedad, intensidad de luz, etc.) [2].

6. Una aplicación de las WSN en seguridad es la de mantenimiento preventivo. El proyecto Loch Rannoch de BP desarrolló un sistema comercial para usar en las refinerías. El sistema monitorea maquinaria rotatoria crítica para evaluar sus condiciones de operación e informar cuando hay desgaste o roturas. De esta manera se puede predecir el desgaste de la máquina y realizar un mantenimiento preventivo. Las redes de sensores se pueden usar para detectar agentes químicos en el aire y en el agua. Pueden también ayudar en la identificación del tipo, la concentración y la ubicación de agentes contaminantes.

7. Un ejemplo del uso de una WSN en aplicaciones de salud es el Bi.Fi, un sistema de arquitectura imbuida para el monitoreo de pacientes internos de un hospital, o para el cuidado de pacientes externos. Fue concebido en UCLA y se basa en la arquitectura SunSPOT de Sun. Las motas miden datos biológicos de como señales neurológicas, oximetría del pulso y electrocardiografías. Después las motas interpretan, filtran y transmiten estos datos para permitir una intervención temprana.

Los roles en una WSN

Los nodos de una WSN pueden tener diferentes roles.

1. Los nodos sensores se usan para detectar su entorno y transmitir las lecturas a un nodo recolector (*sink*) también llamado "estación base". Estos están normalmente equipados con diferentes tipos de sensores. Una mota está dotada de capacidades de procesamiento, capacidades comunicativas y de detección autónomas.

2. Los nodos recolectores (*sink*) o "estaciones base" tienen la labor de recolectar las lecturas de los sensores de los otros nodos y pasar estas lecturas a un portal al cual están conectados directamente para realizar ulteriores procesamientos/análisis. Un nodo recolector tiene capacidades mínimas de procesamiento y comunicación, pero carece de capacidades de detección.

3. Los actuadores son dispositivos que se usan en el control del ambiente que responden a las lecturas de los sensores o a otras entradas. Un actuador puede tener la misma configuración de una mota pero está dotado de capacidades de control, por ejemplo encender una luz cuando hay bajas condiciones de luminosidad.

Las pasarelas, a menudo conectadas a los nodos recolectores, se alimentan normalmente con una fuente de alimentación estable ya que consumen una cantidad considerable de energía. Estos dispositivos son aparatos normales de computación como laptops, notebooks, computadoras de mesa, teléfonos u otros dispositivos que surjan y puedan almacenar y enrutar las lecturas desde el sensor hasta el sitio de procesamiento de los datos. Sin embargo, podrían no estar dotados de capacidad de detección.

Puesto que son limitados en su cobertura, los sensores mota necesitan capacidades multi-salto que permitan: 1) Incremento del alcance más allá de lo que permitiría un solo nodo comunicándose con el nodo adyacente. 2) adaptación a cambios en la red, por ejemplo, circunvalar un nodo defectuoso usando un trayecto diferente para permitir la comunicación, y 3) el uso de menor energía del transmisor ya que cada nodo cubre una distancia más pequeña.

Los nodos sensores se despliegan de tres maneras:

a) Como nodos sensores para detectar o percibir el ambiente,
b) Como nodos de relevo de las medidas provenientes de otros nodos sensores;
c) Como nodo recolector o estación base que se conecta a una pasarela (laptop, tableta, iPod, teléfono inteligente, computadora de escritorio) con presupuesto más alto de energía como para procesar localmente las lecturas de los sensores o para transmitir estos datos a sitios remotos para su procesamiento.

Introducción a IPv6

IPv6 se refiere a la versión 6 del Protocolo de Internet, así que la importancia de IPv6 está implícita en su nombre: ¡es tan importante como Internet! El Protocolo de Internet (de aquí en adelante **IP**) se concibió como una solución a la necesidad de interconectar redes de datos de diferente tecnologías y se ha transformado en el estándar *de facto* para todo tipo de comunicación digital. Hoy en día, IP está presente en todos los dispositivos capaces de enviar y recibir información digital, no solamente la Internet.

El IP está estandarizado por la *Internet Engineering Task Force* (IETF), la organización que está a cargo de todos los estándares de Internet para garantizar la interoperabilidad entre los diferentes vendedores de software. El hecho de que el IP sea estándar es de importancia vital
puesto que hoy en día todo se está conectando a Internet y los dispositivos de diferentes fabricantes deben ser capaces de interactuar. Todos los sistemas operativos actuales y las librerías en Internet usan IP para enviar y recibir datos.

La IoT queda incluida en ese "todo está conectado a Internet", así que ya sabes por qué estás leyendo sobre IPv6, la última versión del Protocolo de Internet.

Los objetivos de este capítulo son:

- Describir brevemente la historia del Protocolo de Internet (IP).
- Averiguar para qué se usa IPv6.
- Cubrir los conceptos relacionados con IPv6 necesarios para entender el resto del libro.
- Dar una visión práctica del IPv6 y su manera de asignar las direcciones y presentar una idea general de cómo es una red basada en IPv6.

Un poco de historia

ARPANET fue el primer intento del Departamento de Defensa (DoD) de los Estados Unidos por construir una red descentralizada que fuera más resistente a los ataques y que a la vez interconectara sistemas completamente diferentes. El primer protocolo usado ampliamente para este propósito se llamó IPv4 (Protocolo de Internet versión 4), la base de Internet para uso civil. Inicialmente sólo estaban conectados los centros de investigación y las universidades, apoyados por la *National Science Foundation* (**NSF**). Las aplicaciones comerciales no estaban permitidas, pero cuando la red comenzó a crecer exponencialmente, la NSF decidió transferir sus operaciones y su financiamiento al sector privado y se eliminaron las restricciones al tráfico comercial. Aunque las aplicaciones principales eran el correo electrónico y la transferencia de archivos, fue con el desarrollo de la *World Wide Web* basada en el protocolo HTML y específicamente, con el navegador gráfico MOSAIC y sus sucesores que hubo una explosión del tráfico, e Internet comenzó a ser usada masivamente. Como consecuencia, hubo un agotamiento de las direcciones IP disponibles en IPv4 ya que este nunca fue diseñado para abarcar a tan alto número de dispositivos.

Con el fin de contar con más direcciones se necesita asignar mayor espacio a cada dirección IP (un mayor número de bits para especificar la dirección), lo que significa una nueva arquitectura que implica cambios a la mayor parte del software de redes.

Después de examinar varias propuestas, la IETF se decidió por el IPv6 descrito en la RFC 1752 de enero de 1995. IPv6 es conocido también como Protocolo de Internet de Nueva Generación, **IPng.** La IETF actualizó el estándar IPv6 en 1998 con la nueva definición descrita en la RFC 2460. Para el 2004, IPv6 ya estaba ampliamente disponible para la industria y era compatible con la mayoría de los equipos nuevos de redes. Hoy en día IPv6 coexiste con IPv4 en Internet y la cantidad de tráfico en IPv6 crece rápidamente en la medida en que más ISP (*Internet Service Providers*) y creadores de contenido están usando IPv6 además de IPv4.

Como vemos, IP e Internet tienen más o menos la misma historia, y es por eso que el crecimiento de Internet se ha visto constreñido por las limitaciones de IPv4 y ha llevado al desarrollo de la nueva versión de IP, IPv6, como el protocolo que se usa para interconectar todo tipo de dispositivos para el envío y recepción de información. Incluso hay tecnologías que se han desarrollado pensando sólo en el IPv6. Un buen ejemplo en el contexto de la IoT es 6LowPAN.

De ahora en adelante nos enfocaremos sólo en el IPv6. Si sabes algo de IPv4 tienes la mitad del camino andado. Si no, no hay que preocuparse. Abordaremos los conceptos principales de manera clara y concisa.

Conceptos de IPv6

Trataremos los conceptos básicos de IPv6 que se necesitan para entender por qué esta última versión del IP es útil para la IoT, y cómo se relaciona con otros protocolos como el 6LowPAN del que se hablará luego. Sería útil tener ya los conceptos básicos de Internet y estar familiarizado con los conceptos de bits, bytes, pilas de red, capa de red, paquetes, cabecera de IP, etc. De todas maneras, aquí proporcionaremos estos conceptos de base para entender IPv6 y sus características. Debe entenderse que IPv6 es un protocolo diferente que no es compatible con IPv4. En la figura siguiente presentamos el modelo de capas usado en Internet.

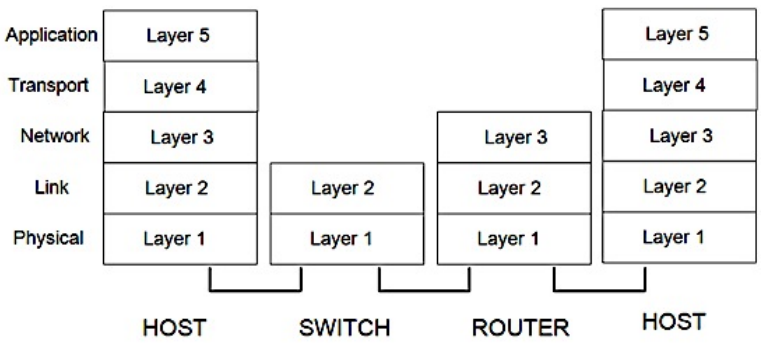

Figura 4. La pila de protocolos de Internet

IPv6 está en la capa 3, también llamada capa de red. Los datos manejados por la capa 3 se llaman paquetes. Los dispositivos conectados a la Internet pueden ser anfitriones (*hosts)* o enrutadores (*routers*). Un *host* puede ser un computador de mesa, una laptop, o una tarjeta de sensores, que envía y/o recibe paquetes de datos. Los *hosts* son los destinatarios de los paquetes. Los enrutadores, en cambio, se encargan del transporte de los paquetes y son responsables de escoger el próximo enrutador que reenviará los datos a su destino final. La Internet se compone de una cantidad de enrutadores interconectados que reciben paquetes de datos en una interfaz y los envía lo más rápido posible, usando otra interfaz, hacia el siguiente enrutador.

Paquete IPv6

En el modelo de capas que vimos antes, cada capa introduce su información en el paquete, y esta información está dirigida y puede ser procesada solamente por la misma capa de otro dispositivo IP. Esta "conversación" entre capas del mismo nivel en dos diferentes dispositivos es el *protocolo.*

Las capas definidas son:

- **Aplicación** (capa 5)**.** Aquí se encuentra el software que utiliza los servicios de red de las capas inferiores. Un ejemplo es el navegador que abre una conexión de red hacia un servidor web. Otro ejemplo es el software que opera en el servidor de algún punto de Internet a la espera de responder las solicitudes de los navegadores-clientes. Ejemplos de protocolos de aplicación son HTTP o DNS.

- **Transporte** (capa 4)**.** Aquí se encuentra el software que utiliza los servicios de red de las capas inferiores. Es una capa que está sobre la capa de red y que le presta servicios adicionales, por ejemplo, retransmitir paquetes perdidos o garantizar que el orden de entrega de los paquetes sea el mismo en el que fueron enviados. Esta capa será la que presta un "servicio de red" a la capa de aplicación. Los dos protocolos de transporte más usados en la Internet son TCP y UDP.

- **Red** (capa 3)**.** Esta capa se encarga de la entrega correcta de los datos recibidos desde la capa de transporte hasta su destino, así como la correspondiente recepción de los datos provenientes de la capa de enlace en el otro extremo de la conexión. En Internet existe un solo protocolo de red que es el IP. Procedencia y destino tienen cada uno su dirección IP.

- **Enlace** (capa 2)**.** La capa de enlace se encarga de organizar en tramas, (grupos de bytes), los datos enviados desde la capa de red en el ámbito de una red de área local o LAN. Esta capa tiene sus propias direcciones que cambiarán según la tecnología que se use y se conocen como direcciones MAC o direcciones físicas, en contraposición con las direcciones IP que se conocen como direcciones lógicas. En el otro extremo, las secuencias de bits originadas en la capa física se predisponen en tramas para ser entregadas a la capa de red.

- **Física** (capa 1)**.** Esta es la capa de más bajo nivel, que describe las codificaciones y el tipo de señales electromagnéticas empleadas para transferir información digital desde un nodo hasta el próximo. Abarca todos los medios físicos tanto cableados como inalámbricos.

La figura siguiente ilustra la idea de que cada una de las capas mencionadas recibe algunos bytes y añade información específica para que esa capa pueda ser procesada en la correspondiente capa al otro extremo de la comunicación. Aquí vemos que los datos procedentes de la capa de aplicación (Capa 5) son procesados por las sucesivas capas hasta llegar a la capa física (Capa 1) que los transmite al otro extremo. Aquí tendrá lugar el proceso inverso, la secuencia de bits se transforma en tramas, paquetes, segmentos y mensajes a medida que transita de la capa física a la de aplicación.

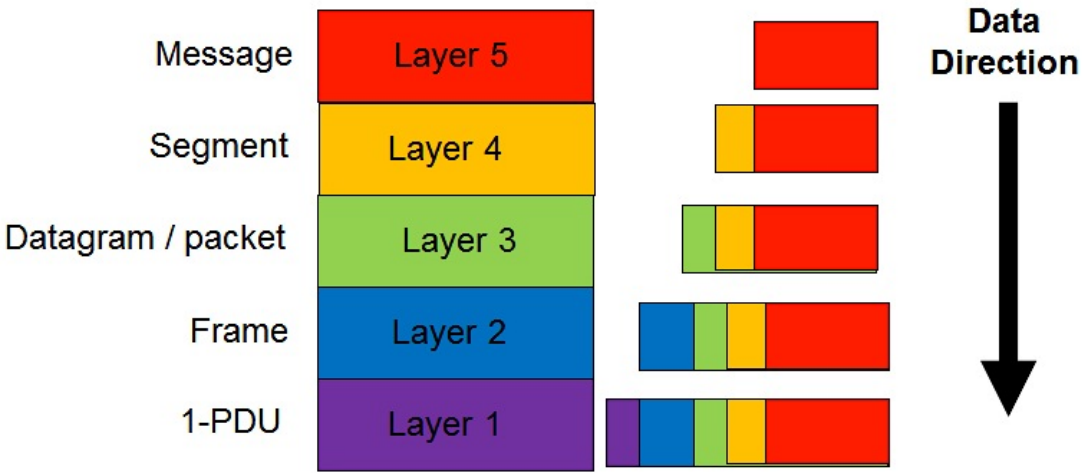

Figura 5. Flujo de datos en la pila

Si nos enfocamos en la capa de red, específicamente en IPv6, los bytes enviados y recibidos en el paquete IP tienen un formato estandarizado. La figura siguiente muestra la estructura de una cabecera IPv6 básica:

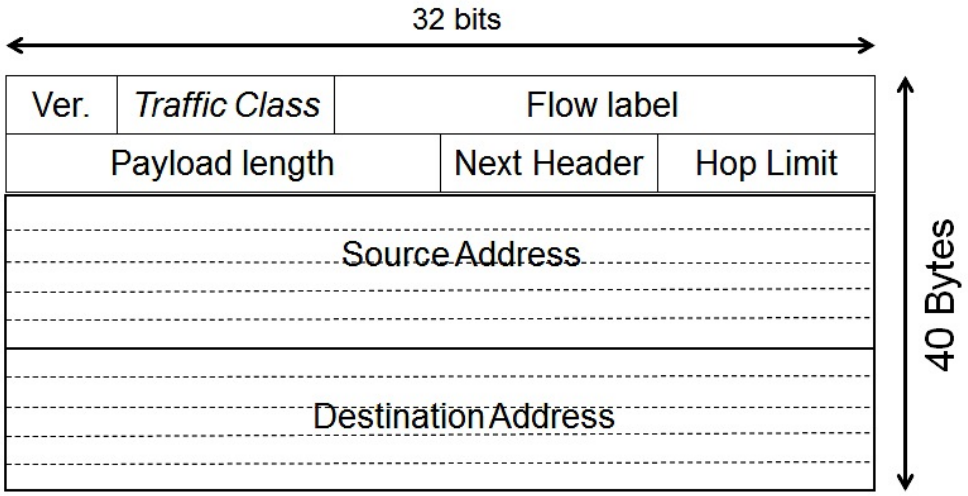

Figura 6. Cabecera de IPv6

Primero tenemos la cabecera **básica IPv6** con un tamaño fijo de 40 bytes seguido opcionalmente por cabeceras de extensión que serán tratadas más tarde. A continuación tenemos la carga útil, constituida por los datos de las capas superiores. Como vemos, hay varios campos en la cabecera del paquete con algunas mejoras en comparación con la cabecera IPv4:

- El número de campos se ha reducido de 12 a 8.
- La cabecera básica IPv6 tiene un tamaño fijo de 40 bytes y está alineado con 64 bits lo que permite un reenvío de paquetes basado directamente en hardware.
- El tamaño de las direcciones se aumentó de 32 a 128 bits.

Los campos más importantes son las direcciones de origen y de destino. Como sabemos, cada dispositivo IP tiene una dirección IP única que lo identifica en Internet. Esta dirección IP es usada por los enrutadores para tomar decisiones de reenvío.

La cabecera IPv6 tiene 128 bits por cada dirección IPv6, lo que permite 2^{128} direcciones (aproximadamente $3{,}4 \times 10^{38}$, es decir 3,4 seguido de 38 ceros), en contraste con IPv4 que tiene 32 bits para codificar las direcciones IPv4, lo que permite 2^{32} direcciones (4 294 967 296).

Hemos visto la cabecera básica de IPv6 y mencionado las cabeceras **de extensión** *(extensión headers)***.** Para mantener la cabecera básica simple y de un tamaño fijo, se han añadido rasgos especiales al IPv6 usando cabeceras de extensión.

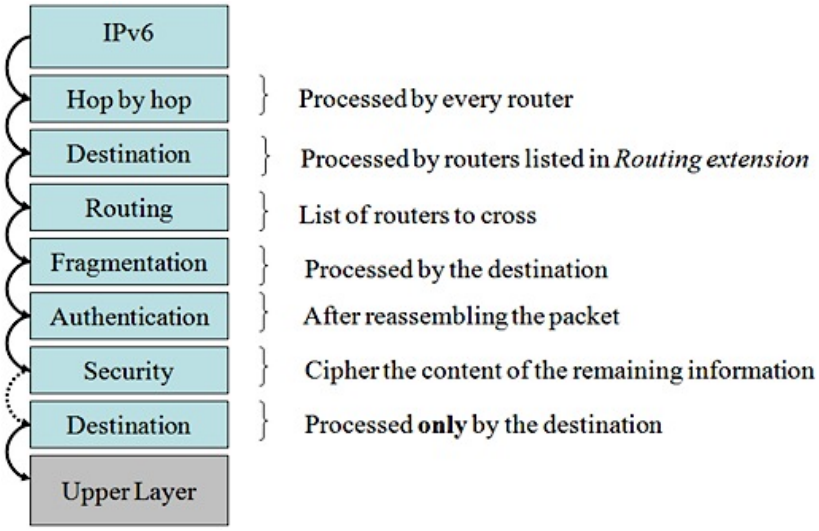

Figura 7. Cabeceras de extensión

Las cabeceras de extensión se han definido como puede verse en la figura previa y deben seguir el orden que se muestra. Ellas:

- Dan flexibilidad, por ejemplo, para activar la seguridad cifrando los datos del paquete.
- Optimizan el procesamiento de los paquetes porque, exceptuando la cabecera salto a salto, las extensiones son procesadas sólo por nodos terminales (origen y destino de los paquetes), y no por cada enrutador en el trayecto.
- Están organizadas como una "cadena de cabeceras" que comienza siempre con la cabecera básica de IPv6 que usa el campo "cabecera próxima" para señalar la próxima cabecera extendida.

Direccionamiento IPv6

Como se observa en la figura de la cabecera básica IPv6, los campos de las direcciones tanto de la fuente como del destino tienen 128 bits para codificar una dirección IPv6. El uso de 128 bits para las direcciones reporta algunos beneficios:

- Proporciona muchas más direcciones para satisfacer las necesidades actuales y futuras, con amplio espacio para innovaciones.
- Mecanismos fáciles de auto-configuración de direcciones.
- Fácil manejo/delegación de las direcciones
- Espacio para más niveles de jerarquía y agregado de rutas
- Habilidad para hacer IPsec (protocolo de seguridad) de extremo a extremo

Las direcciones IPv6 se clasifican en las categorías siguientes (que también existen en IPv4).

- **Unicast** (uno a uno): se usa para enviar paquetes desde un origen a un destino. Son las más comunes y las describiremos, así como sus subclases.

- **Multicast** (uno a muchos): se usa para mandar paquetes desde un origen hacia múltiples destinos. Esto es posible por medio de enrutamiento de reparto múltiple que permite que los paquetes se repliquen en algunos lugares.

- **Anycast**: (uno al más próximo): se usa para enviar paquetes desde un origen hacia el destino más cercano.

- **Reservada**: son las direcciones o grupos de ellas que tienen usos específicos, por ejemplo, direcciones que se van a usar para documentación y ejemplos.

Antes de entrar en más detalles sobre las direcciones IPv6 y los tipos de dirección *unicast*, veamos cuál es su apariencia y cuáles son sus reglas de notación. Hay que tener esto claro porque probablemente el primer problema que encontraremos cuando usamos IPv6 es cómo escribir una dirección.

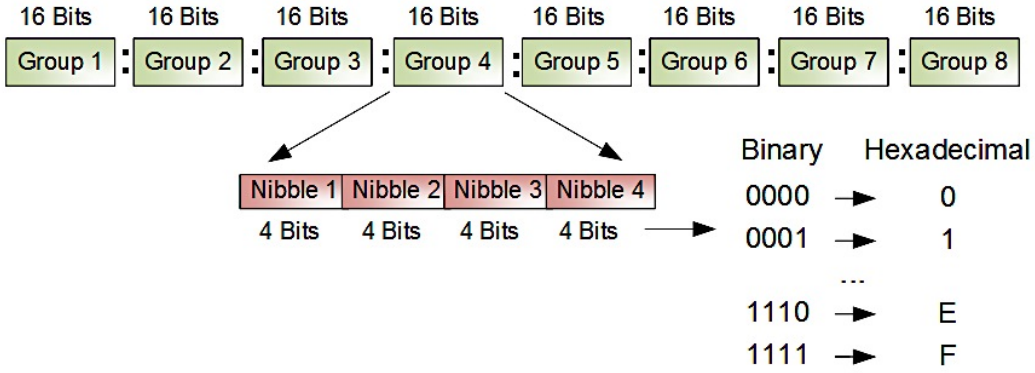

Figura 8. Dirección IPv6

Las reglas de notación son estas:

- 8 grupos de 16 bits separados por ":".
- Notación hexadecimal para cada *nibble* (secuencia de 4 bits)
- Indiferente el uso de mayúsculas o minúsculas
- Los prefijos de red (grupos de direcciones) se escriben Prefijo / Longitud del Prefijo. Es decir, la longitud del prefijo indica el número de bits que son fijos en la dirección.
- Los ceros a la izquierda dentro de cada grupo pueden eliminarse.
- Uno o más grupos de sólo-ceros pueden sustituirse por "::" solamente una vez.

Las primeras tres reglas exponen la base de la notación de las direcciones IPv6. Estas usan notación hexadecimal, es decir, los números se representan por dieciséis símbolos entre el 0 y F. Tendremos ocho grupos de cuatro símbolos hexadecimales, cada uno separado por el símbolo de dos puntos (:). Las dos últimas reglas son para comprimir la notación de las direcciones. Veremos en la práctica cómo funciona todo esto mediante algunos ejemplos.

1) Si representamos todos los bits de la dirección tendremos la forma preferida. Por ejemplo:

2001:0db8:4004:0010:0000:0000:6543:0ffd

2) Si usamos corchetes para encerrar las direcciones tendremos la forma literal de la dirección:

[2001:0db8:4004:0010:0000:0000:6543:0ffd]

3) Si aplicamos la cuarta regla que permite la compresión dentro de cada grupo para eliminar los ceros a la izquierda, tendremos:

2001:db8:4004:10:0:0:6543:ffd

4) Si aplicamos la quinta regla y permitimos la compresión de uno o más grupos de ceros consecutivos usando "::" obtenemos:

2001:db8:4004:10::6543:ffd

Se debe tener cuidado cuando se comprimen o descomprimen direcciones IPv6. El proceso debería ser reversible. Es común cometer algunos errores. Por ejemplo, la dirección siguiente:

2001:db8:A:0:0:12:0:80 podría comprimirse más usando "::". Tenemos dos opciones.

A) 2001:db8:A::12:0:80 B) 2001:db8:A:0:0:12::80

Ambas son direcciones IPv6 correctas. La siguiente dirección 2001:db8:A::12::80 es incorrecta y no sigue la última regla de compresión descrita arriba. El problema con esta dirección mal comprimida es que no estamos seguros de cómo expandirla; es ambigua. No podemos saber si se expande como 2001:db8:A:0:12:0:0:80, o como 2001:db8:A:0:0:12:0:80.

Prefijo de red IPv6

Por último, pero también importante, hay que entender el concepto de **prefijo de red**, que indica algunos bits fijos y otros no definidos que podrían usarse para crear nuevos sub-prefijos o para definir direcciones IPv6 completas.

Veamos algunos ejemplos:

1) El prefijo de red 2001:db8:1::/48 (forma comprimida de 2001:0db8:0001:0000:0000:0000:0000:0000) indica que los primeros 48 bits serán siempre los mismos (2001:0db8:0001), pero que podemos jugar con los 80 bits restantes, como por ejemplo, para obtener dos prefijos más pequeños: 2001:db8:1:a::/64 y 2001:db8:1:b::/64.

2) Si tomamos uno de los prefijos pequeños anteriores, 2001:db8:1:b::/64, en el que los primeros 64 bits son fijos, nos quedan los 64 bits de la extrema derecha todavía por asignar, por ejemplo, a una interfaz: 2001:db8:1:b:1:2:3:4. Este ejemplo nos permite introducir un concepto básico en IPv6: **En una LAN (*Local Area Network*: Red de Área Local) se usa siempre un prefijo /64. Los 64 bits más a la derecha se conocen coma la identidad de la interfaz IID (por sus siglas en inglés) porque identifican claramente la interfaz del host en la red local definida por el prefijo /64.** La siguiente figura ilustra el concepto.

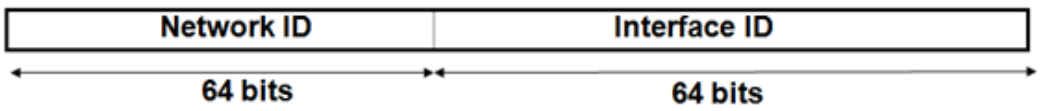

Figura 9. Identificación de red y de interfaz

Ahora que hemos visto la primera dirección IPv6 podemos hablar en más detalle sobre dos tipos de direcciones que encontraremos cuando se trabaja con IPv6: *reserved* y *unicast*.

Las siguientes son direcciones reservadas (*reserved)* o de fines especiales:

- Dirección **no especificada** se usa como dirección vacía cuando no se dispone de una dirección: 0:0:0:0:0:0:0:0 (::/128)

- La dirección **autoconexión (loopback)**, para una interfaz que se envía paquetes a sí misma 0:0:0:0:0:0:0:1 (::/128)

- **Prefijo de documentación**: 2001:db8::/32. Este prefijo se reserva para el uso en ejemplos de documentación como se ha visto antes.

Como se especificó en [RFC6890], IANA mantiene un registro sobre las direcciones IPv6 para propósitos especiales [IANA –IPV6-SPEC].

Las siguientes son algunas direcciones del tipo *unicast* [RFC4291]:

- **Enlace local**. Las direcciones de enlace local (*link-local*) se configuran en cualquier interfaz IPv6 que esté conectada a una red. Todas comienzan con el prefijo FE80::/10 y pueden usarse para la comunicación con otros *hosts* en la misma red local; es decir, todos los *hosts* están conectados al mismo *switch*. No pueden usarse para comunicarse con otras redes como por ejemplo, enviar o recibir paquetes por medio de un enrutador.

- **ULA (Unique Local Address**: (Dirección Local Única). Todas las direcciones ULA comienzan por el prefijo FC00::/7. Se usan para comunicaciones locales, normalmente dentro de un mismo sitio y no se espera que sean "enrutables" hacia la Internet Global, sino ser usadas solamente dentro de una red más limitada manejada por la misma organización.

- **Unicast Global**. Equivalentes a las direcciones públicas de la IPv4. Son únicas en toda la Internet y podrían usarse para enviar paquetes desde y hacia cualquier parte de Internet.

Para qué se usa IPv6

Como hemos visto, IPv6 tiene propiedades que facilitan cosas como el direccionamiento global y la autoconfiguración de las direcciones de los *hosts*. Ya que lPv6 proporciona una cantidad tal de direcciones que podríamos usarlas por cientos de años, se podría asignar una dirección IPv6 unívoca (*unicast)* a casi cualquier cosa que nos venga en mente. Esto nos devuelve al paradigma inicial de Internet en el que cada dispositivo IP puede comunicarse con cualquier dispositivo IP. Esta comunicación de extremo a extremo permite una comunicación bidireccional a lo largo de toda la Internet y entre todos los dispositivos IP, lo que resulta en aplicaciones que colaboran entre sí y en nuevas formas de almacenamiento, envío y acceso a la información.

En el contexto de este libro, por ejemplo, se podría pensar en sensores IPv6 distribuidos en todo el mundo que recolectaran, enviaran y se contactaran desde diferentes sitios para crear una malla global de valores físicos medidos, almacenados y procesados.

La disponibilidad de una enorme cantidad de direcciones ha hecho posible un nuevo mecanismo llamado autoconfiguración de direcciones sin estado (***Stateless Address Autoconfiguration***) (SLAAC) que no existía en IPv4. A continuación presentamos un resumen de las formas en que se puede configurar una dirección en una interfaz IPv6:

- **Estática.** Podemos decidir cuál dirección darle a nuestro dispositivo IP y luego configurarla manualmente en él usando cualquier tipo de interfaz: web, línea de comandos, etc. Normalmente también hay que configurar otros parámetros de red como la pasarela que se va a usar para enviar paquetes fuera de la red local.

- **DHCPv6**. (Configuración Dinámica del Host para IPv6) [RFC3315]. Este es un mecanismo similar al que ya existía en IPv4 y el concepto es el mismo. Se necesita configurar un servidor dedicado que, luego de una negociación breve con el dispositivo a configurar, le asigna una dirección IP. DHCPv6 permite la configuración automática de dispositivos IP, se llama configuración de dirección de estado permanente (*stateful address configuration*), porque el servidor DHCPv6 mantiene un registro del estado de las direcciones asignadas.

- **SLAAC** (*Stateless Address Autoconfiguration*). La autoconfiguración de direcciones sin estado es un nuevo mecanismo introducido con IPv6 que permite configurar automáticamente todos los parámetros de red en un dispositivo IP usando directamente el mismo enrutador que proporciona conectividad a la red.

La ventaja de SLAAC es que simplifica la configuración de dispositivos "tontos" como sensores, cámaras u otros, que tienen baja capacidad de procesamiento. No se necesita usar una interfaz en el dispositivo IP, solo *plug and net* (enchufar y listo). También simplifica la estructura de red que se necesita para construir una red IPv6 básica, porque no necesitas un dispositivo/servidor adicional. Se usa el mismo enrutador que se necesita para enviar paquetes fuera de la red para configurar los dispositivos IP. No vamos a entrar en detalles pero sólo necesitamos saber que en una red local, normalmente llamada LAN (*Local Area Network*) conectada a un enrutador, éste tiene la función de enviar toda la información que los *hosts* necesitan para la configuración usando un mensaje RA (*Router Advertisement*: Anuncio del enrutador). El enrutador va a enviar estos RA periódicamente, pero con la finalidad de acelerar el proceso, un *host* puede también enviar un mensaje RS (*Router Solicitation*: Solicitud al enrutador) cuando su interfaz se conecta a la red. El enrutador envía inmediatamente un mensaje RA en respuesta al RS.

La figura siguiente muestra un intercambio de paquetes entre un host que se acaba de conectar a una red local y un destino IPv6 en Internet.

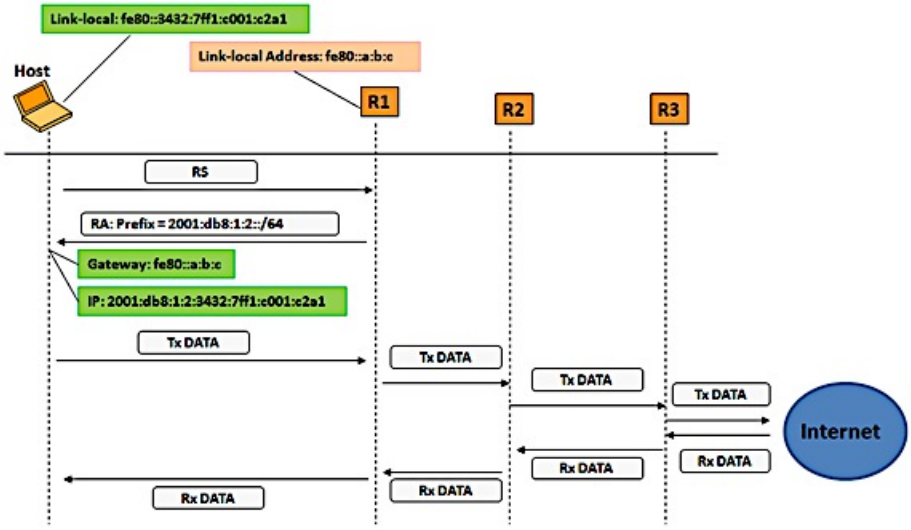

Figura 10. Intercambio de paquetes IPv6

1) R1 es el enrutador que le da conectividad al *host* en la LAN y está enviando mensajes RA regularmente.

2) Tanto R1 como el *host* tienen una dirección de enlace local (*link-local*) en sus interfaces conectada a la LAN del *host*. Esta dirección se configura automáticamente cuando la interfaz está lista. Nuestro *host* crea la dirección de enlace local combinando los 64 bits más a la izquierda del prefijo del enlace local (fe80::/64) y los 64 bits más a la derecha de una IID generada localmente (:3432:7ff1:c001:c2a1). Estas direcciones de enlace local pueden ser usadas en la LAN para intercambiar paquetes, pero no para enviarlos fuera de la LAN.

1. El *host* necesita dos cosas básicas para poder enviar paquetes a otras redes: una dirección IPv6 global y la dirección de la pasarela (Gateway), es decir, de un enrutador al cual enviar los paquetes que se quieren enrutar fuera de su red.

2. Aunque R1 está mandado mensajes RA constantemente (cada cierto número de segundos), cuando se conecta el *host* y tiene la dirección de enlace local configurada, éste manda un RS al cual R1 responde inmediatamente con un RA que contiene dos cosas:

    A) **Un prefijo global de 64 bits de longitud** que se usa para el SLAAC. El *host* toma el prefijo recibido y le agrega una IID generada localmente, normalmente la misma usada para la dirección de enlace local. De esta manera el *host* obtiene una IPv6 global, permitiendo así que se pueda comunicar con la Internet en IPv6.

    B) **La dirección de enlace local de R1** queda incluida implícitamente, porque es la dirección fuente del paquete RA. El *host* puede usar esta dirección para configurar la **pasarela por defecto**, es decir, el sitio al cual enviar los paquetes por defecto para contactar un host IPv6 en cualquier lugar de Internet.

3. Una vez que ambos, la pasarela y la dirección global IPv6 están configurados, el *host* puede recibir o enviar información. En la figura anterior tenemos unos datos que el *host* quiere enviar (**Tx Data**) a otro *host* en Internet, así que crea un paquete IPv6 con la dirección de destino del *host* destinatario y como dirección de origen la dirección global recientemente autoconfigurada, y procede a enviarla a la dirección de enlace local de su pasarela R1. El *host* destinatario puede responder con algunos datos (**Rx Data**).

Ejemplo de red

A continuación ilustramos el aspecto de una red sencilla IPv6 mostrando direcciones IPv6 para todos los dispositivos de red.

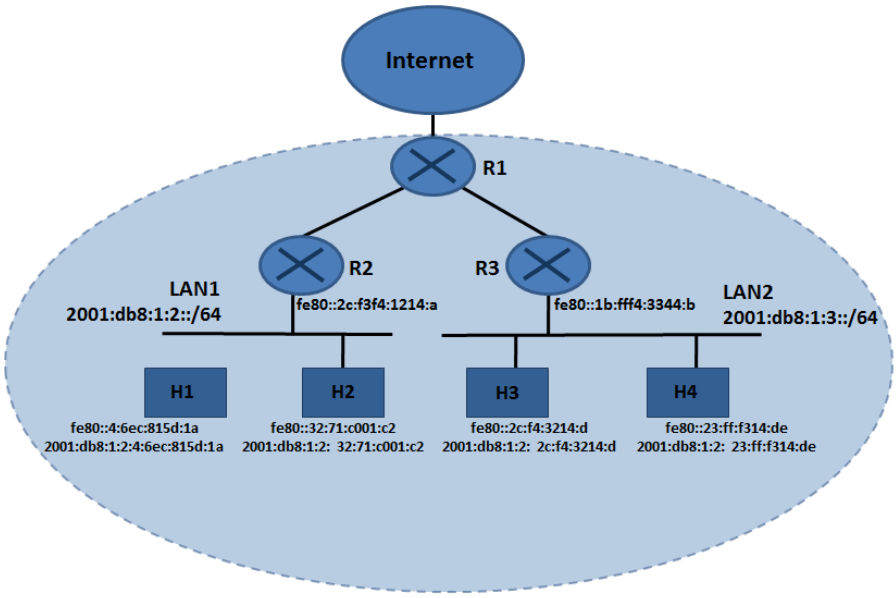

Figura 11. Red IPv6 simple

Tenemos cuatro *hosts* (sensores u otros dispositivos), y queremos colocar un par de ellos en dos sitios diferentes, por ejemplo, dos pisos en un edificio. Estamos manejando cuatro dispositivos IP pero podemos tener hasta 2^{64} (18.446.744. 073.709.551.616) dispositivos conectados a la misma LAN.

Hemos creado dos LAN con un enrutador en cada una. Ambos enrutadores están conectados a un enrutador central (R1) que le da conectividad a Internet. LAN1 está servida por R2 (con dirección de enlace local fe80::2c:f3f4:1214:a en esa LAN) y usa el prefijo 2001:db8:1:2::/64 anunciado por SLAAC. LAN2 está servida por R3 (con dirección de enlace local fe80::1b:fff4:3344:b en esa LAN), y usa el prefijo 2001:db8:1:3::/64 anunciado por SLAAC.

Todos los *hosts* tienen tanto una dirección IPv6 de enlace local, como una dirección IPv6 global autoconfigurada usando el prefijo asignado por el enrutador correspondiente a través de mensajes RA. Además, recordemos que cada host configura también la pasarela usando la dirección de enlace

local usada por el enrutador para mensajes RA. Las direcciones de enlace local pueden usarse para la comunicación entre *hosts* dentro de la LAN, pero para hacerlo con *hosts* en otras LAN u otra red fuera de su propia LAN se necesita una dirección IPv6.

Breve introducción a Wireshark

¿Qué es Wireshark?

Wireshark es un analizador de paquetes gratuito de fuente abierta que permite que las huellas (*traces*) de los paquetes sean rastreadas, captadas y analizadas.

La huella de un paquete es un registro del tráfico en algún punto de la red, como si se tomara una instantánea de todos los bits que pasan a través de un cable en particular. La huella de un paquete registra una instantánea en el tiempo para cada paquete junto con los bits que lo conforman, desde los encabezados de las capas más bajas hasta los contenidos de las capas superiores.

Wireshark es compatible con la mayoría de los sistemas operativos, incluidos Windows, Mac y Linux. Proporciona una interfaz gráfica de usuario que muestra la secuencia de paquetes y el significado de los bits cuando se interpretan como cabeceras de protocolo y datos. Los paquetes tienen código de color para expresar su significado y Wireshark incluye varias formas de filtrarlos y analizarlos para que investigues los diferentes aspectos de su comportamiento. Es muy utilizado para la detección de problemas de la red.

Un ejemplo de uso común es cuando una persona quiere detectar los problemas en su red o examinar el funcionamiento interno de un protocolo de red. Podría, por ejemplo, ver exactamente lo que sucede cuando accede a un sitio web o instala una red de sensores inalámbricos. Es también posible filtrar e investigar los atributos de un paquete dado, lo que facilita el proceso de depuración.

Puedes encontrar más información en:

https://www.wireshark.org/

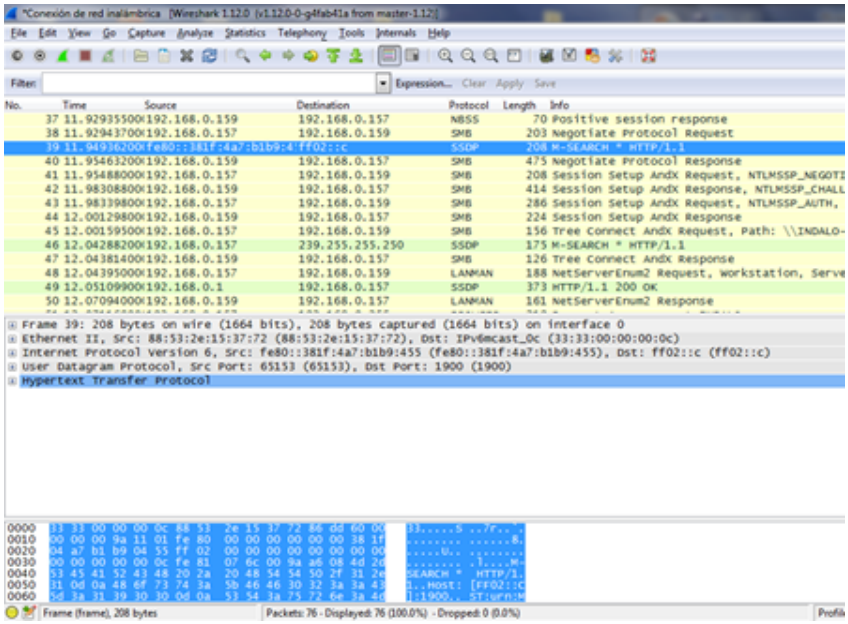

Figura 12. Pantalla de Wireshark

Cuando abres Wireshark, encuentras cuatro áreas principales de arriba a abajo: menús y filtros, lista de paquetes capturados, información detallada sobre el paquete seleccionado, contenido completo del paquete seleccionado en hexadecimal y ASCII. ***Online*** te conecta directamente al sitio de Wireshark, que tiene una guía muy útil sobre el uso y la seguridad de Wireshark. Debajo de ***Files*** se encuentra ***Open*** que te permite abrir datos capturados, guardados previamente, así como datos de muestra (*Sample Captures*). Puedes descargar a través de este sitio web algunas de estas muestras y estudiar los datos. Esto te ayudará a entender qué tipo de datos pueden ser capturados con Wireshark.

La sección ***Capture*** te permite escoger tu interfaz. Puedes ver todas las interfaces disponibles. También te muestra cuáles están activas. Si haces clic en ***details*** obtendrás una información genérica bastante buena sobre las interfaces.

En **Start** puedes seleccionar una o más interfaces para examinar. ***Capture Options*** te permite personalizar cuál información quieres ver durante una *capture*. Dale una mirada a esta sección ya que te da la posibilidad de escoger un filtro, un archivo de captura y otras opciones.
En ***Capture Help*** puedes aprender cómo hacer una captura, y mirar la información que hay en *Network Media* sobre cuáles interfaces funcionan en diferentes plataformas.

Seleccionemos una interfaz y hagamos clic en *Start*. Para interrumpir una captura, presiona el cuadrado rojo arriba, en la barra de herramientas. Si quieres comenzar una nueva captura, presiona al lado, en el triangulo verde que parece una aleta de tiburón. Ahora que ya tienes una captura realizada, puedes pinchar *File* y guardarla, abrirla o fusionarla. Puedes imprimirla, cerrar el programa o exportar tu paquete de muchas maneras.

En ***Edit*** puedes localizar un paquete determinado: con las opciones de ***Search*** puedes copiar los paquetes, marcar (resaltar) un paquete específico o todos ellos. Otra cosa interesante que puedes hace en *Edit* es resetear el valor de tiempo. Habrás notado que el incremento de tiempo se realiza en segundos. Puedes resetearlo desde el paquete que hemos seleccionado. Puedes añadir comentarios a un paquete, configurar sus perfiles y preferencias.

Cuando seleccionas un paquete de una lista de paquetes capturados, Wireshark, a continuación, te muestra informacion detallada de los diferentes protocolos usados por ese paquete, por ejemplo Ethernet.

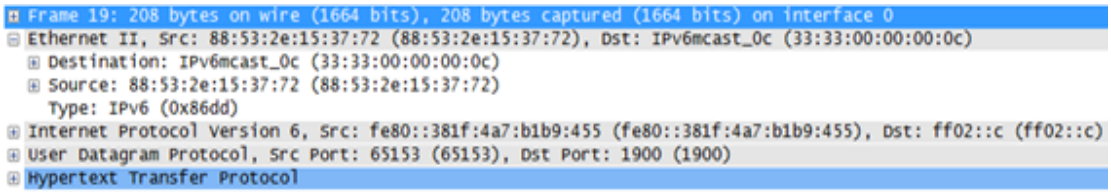

Figura 13. Paquete Ethernet

O en IPv6 , donde podemos ver los campos de los que se ha hablado en la parte teórica: Versión; Clase de tráfico; *Flowlabel* (etiqueta de flujo); Longitud de carga útil (payload); Cabecera próxima, etc.

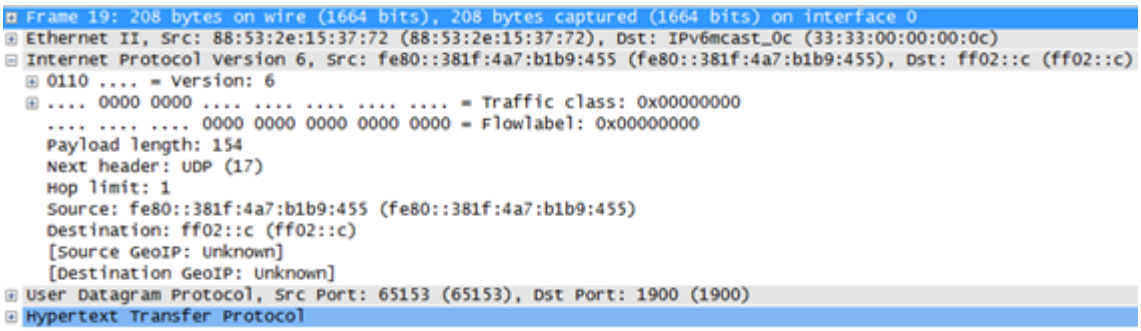

Figura 14. Paquete IPv6

Para aplicar filtros a la lista de los paquetes capturados hay dos métodos:

- Escribe la expresión de un filtro en el espacio correspondiente y aplica el filtro. Se podrían especificar los protocolos (ip, ipv6, icmp, icmpv6). Se pueden crear los campos de un protocolo (ipv6, dst, ipv6.src), e incluso expresiones complejas usando operadores del tipo AND (&&), OR (II) o la negación (I).

Figura 15. Filtro en Wireshark

- Otra opción para crear filtros es hace clic con el botón derecho en un campo de un paquete capturado, en la lista de paquetes. Va a aparecer una opción de menú "***Apply as filter***" con varias opciones de cómo usar ese campo.

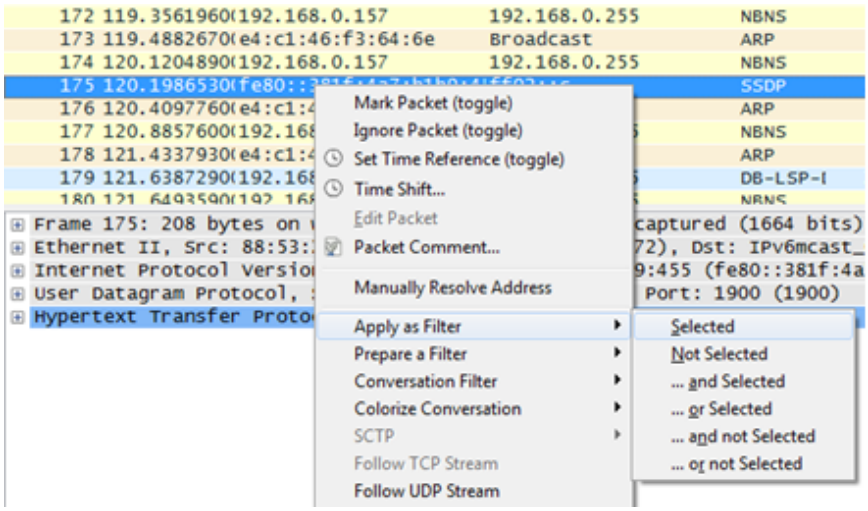

Figura 16. Paquetes capturados

Otra opción interesante de Wireshark es la posibilidad de observar las estadísticas sobre el tráfico capturado. Si hemos aplicado filtros, las estadísticas van a ser sobre el tráfico filtrado. Solo hay que ir al menú ***Statistics*** y seleccionar, por ejemplo, *Protocol Hierarchy*:

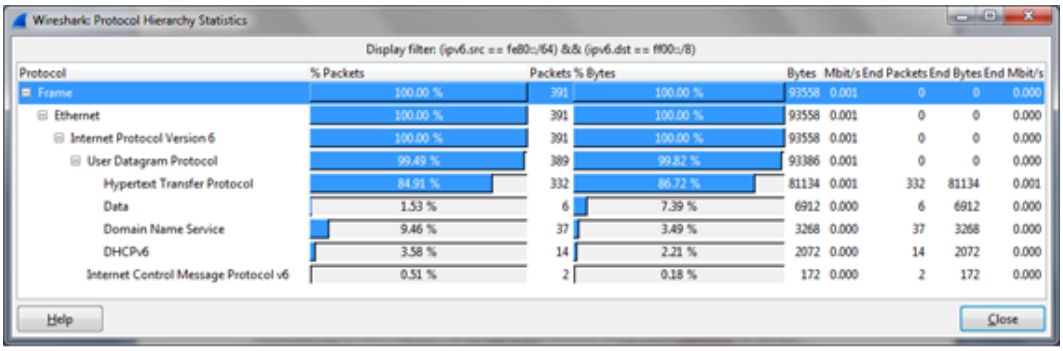

Figura 17. Estadísticas en Wireshark

Otras opciones interesantes son:

- Conversation List ---> IPv6
- Statistics ---> Enpoint List ---> IPv6
- Statistics ---> IO Graph

Esta última opción permite crear y guardar gráficos con diferentes lineas para diferentes tipos de tráfico.

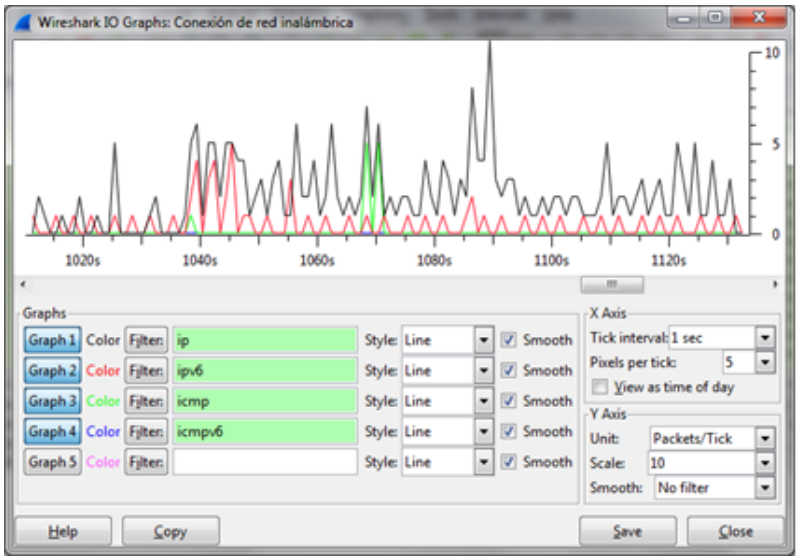

Figura 18. Gráficas en Wireshark

Ejercicios sobre IPv6

Vamos a probar tu conocimiento de IPv6 con los ejercicios siguientes:

1) ¿Qué tamaño tienen las direcciones IPv4 e IPv6, respectivamente?

a. 32 bits, 128 bits
b. 32 bits, 64 bits
c. 32 bits, 112 bits
d. 32 bits, 96 bits
e. ninguna de las anteriores

2) ¿Cuál de las siguiente es una regla de notación válida para las direcciones IPv6?

a. Los ceros a la derecha dentro de un grupo de 16 bits pueden eliminarse.
b. La dirección se divide en 5 grupos de 16 bits separados por ":"
c. La dirección se divide en 8 grupos de 16 bits separados por ":"
d. Uno o más grupos que contengan sólo ceros puede sustituirse por "::"
e. La notación decimal se usa para agrupar bits en 4 (*nibbles*)

3) Los identificadores de Interfaz (IID) o los bits más a la derecha de una dirección IPv6 usada en una LAN tendrán una longitud de 64 bits.

a. Verdadero
b. Falso

4) ¿Cuál de las siguientes es una dirección IPv6 correcta?

a. 2001:db8:A:B:C:D::1
b. 2001:db8:000A:B00::1:3:2:F
c. 2001:db8:G1A:A:FF3E::D
d. 2001:0db8::F:A::B

5) ¿Cuáles de los siguientes sub-prefijos pertenecen al prefijo 2001:db8:0A00::/48? (escoge todos los correctos).

a. 2001:db9:0A00:0200::/56
b. 2001:db8:0A00:A10::/64
c. 2001:db8:0A:F:E::/64
d. 2001:db8:0A00::/64

6) IPv6 tiene una cabecera básica con más campos que el de IPv4

a. Verdadero
b. Falso

7) Las extensiones de cabecera pueden agregarse en cualquier orden

a. Verdadero
b. Falso

8) La autoconfiguración de los dispositivos es la misma en IPv4 e IPv6

a. Verdadero
b. Falso

9) ¿Cuál de las siguientes no es una opción para configurar una dirección IPv6 en una interfaz?

a. DHCPv6
b. Una dirección fija configurada por el vendedor
c. Manualmente
d. SLAAC (Autoconfiguración de Dirección sin Estado)

10) ¿Cuáles paquetes usa SLAAC para autoconfigurar un host IPv6?

a. NS/NA (Solicitud al vecino / Anuncio del vecino)
b. RS/RA (Solicitud al enrutador / Anuncio del Enrutador)
c. Redirección de mensajes
d. NS / RA (Solicitud al vecino / Anuncio del enrutador)

Ejercicios con direcciones

A) Usa las dos reglas de compresión para comprimir al máximos las siguientes direcciones:

1. 2001:0db8:00A0:7200:0fe0:000B:0000:0005
2. 2001:0db8::DEFE:0000:C000
3. 2001:db8:DAC0:0FED:0000:0000:0B00:12

B) Descomprimir al máximo las siguientes direcciones representando todos los 32 *nibbles* en hexadecimal.

1. 2001:db8:0:50::A:123
2. 2001:db8:5::1
3. 2001:db8:C00::222:0CC0

C) Recibes el prefijo IPv6 siguiente para tu red: 2001:db8:A:0100::/56 y tienes la siguiente red:

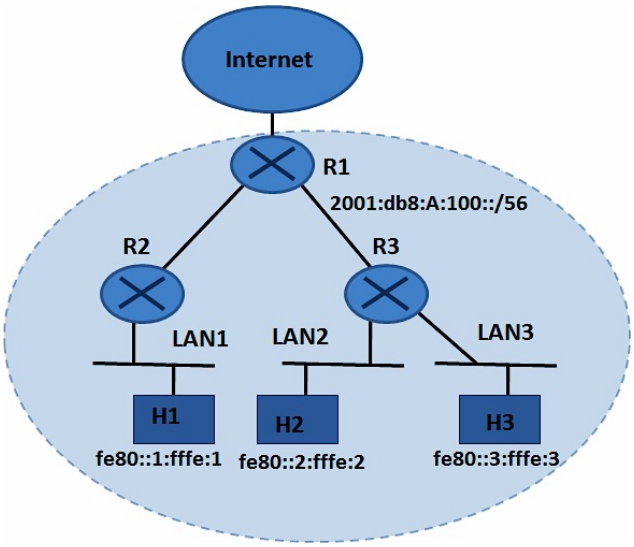

Figura 19. Ejemplo de LAN

Define lo siguiente:

a. El prefijo IPv6 para la LAN 1, un prefijo /64 tomado de el /56 que tienes.
b. El prefijo IPv6 para la LAN 2, un prefijo /64 tomado de el /56 que tienes.
c. El prefijo IPv6 para la LAN 3, un prefijo /64 tomado de el /56 que tienes.
d. Una dirección IPv6 global usando el prefijo de LAN 1 para el *host* H1 (agregado a la dirección de enlace local ya usada).
e. Una dirección IPv6 global usando el prefijo de LAN 2 para el host H2 (agregado a la dirección de enlace local ya usada).
f. Una dirección IPv6 global usando el prefijo de LAN 3 para el host H3 (agregado a la dirección de enlace local ya usada).

Dato	Para dividir el prefijo /56 entre los prefijos /64 hay que cambiar el valor de los bits 57 a 64; es decir, los valores XY en 2001:db8:A01XY::/64.

Conectar nuestra red IPv6 a Internet

Como hemos dicho en la introducción de este libro, la comunicación en la red es uno de los cuatro elementos básicos de la IoT. Ya hemos visto que IPv6 da la posibilidad de asignarle una dirección IP a casi cualquier cosa que podamos imaginar, y puede hacerlo facilitando fácil la autoconfiguración de los parámetros de red en nuestros dispositivos.

Una vez que tenemos todas nuestras "cosas" conectadas utilizando IPv6, éstas pueden comunicarse entre ellas a nivel local o con cualquier otra "cosa" en Internet IPv6. En este capítulo nos centraremos en **el aspecto de Internet de la comunicación de las "cosas"**.

Como veremos en este libro, el tener la posibilidad de conectar nuestros dispositivos a Internet ofrece nuevas posibilidades y servicios. Por ejemplo, si pudiéramos conectar nuestras redes de sensores inalámbricos a un repositorio centralizado, donde toda la información detectada se pudiera procesar junta y almacenar para los registros históricos, esto nos permitiría conocer mejor lo que ha sucedido y tal vez predecir acontecimientos futuros. Esta idea básica es lo que hoy en día se llama "Los Grandes Datos" (*The Big Data*) y tiene todo un conjunto de conceptos y técnicas propios.

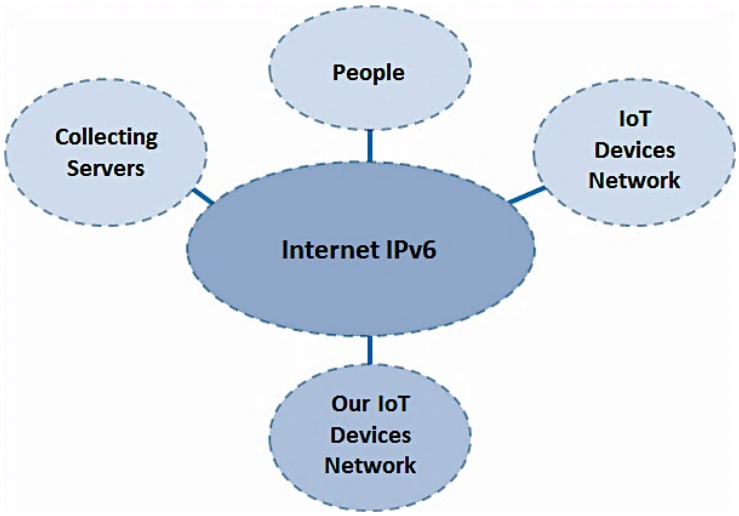

Figura 20. Conectividad IPv6

Volviendo a nuestro campo de la conectividad en la red, el objetivo es conectar nuestros dispositivos IoT a Internet utilizando IPv6, lo que permite la comunicación con otros dispositivos de la IoT, con otros servidores que recolectan datos o incluso con la gente.

Relacionado con la conectividad IPv6 a Internet hay una idea importante: **la comunicación entre los dispositivos de la IoT y la Internet IPv6 podría ser bidireccional.** Esto es importante

resaltarlo porque con IPv4, la conectividad está generalmente diseñada como un canal de una sola dirección entre un cliente y un servidor. Esto cambia con IPv6.

Tener una comunicación bidireccional con los dispositivos IoT traerá posibilidades útiles, porque no es sólo que el dispositivo puede enviar información a algún lugar de Internet, sino que cualquier persona en Internet sería capaz de enviar información, solicitudes u órdenes al dispositivo IoT . Esto se podría utilizar en diferentes situaciones:

- **Gestión.** Para administrar el dispositivo IoT realizando algunas pruebas de estado, actualización de algunos parámetros de configuración o actualización de *firmware,* lo que permite de forma remota usar mejor y más eficientemente la plataforma de hardware y mejorar la seguridad de la infraestructura.
- **Control.** Enviar comandos para controlar los actuadores que hacen que el dispositivo IoT realice una acción.
- **Comunicación**. Para enviar información al dispositivo IoT que se puede utilizar para mostrarla, por medio de algún tipo de interfaz.

IPv6 todavía se está desplegando en las diferentes redes que componen la Internet, lo que significa que podemos encontrar diferentes situaciones al crear nuestra red de dispositivos IoT y al tratar de conectar esta a Internet IPv6. Las siguientes son las tres situaciones más plausibles, ordenadas por preferencia. Esto significa que el mejor objetivo, desde el punto de vista de la conectividad IPv6, es el primero: conectividad nativa IPv6.

- **Conectividad Nativa IPv6**. Esta posibilidad se basa en la disponibilidad de IPv6 tanto por parte del ISP que proporciona conectividad a Internet como en los enrutadores y dispositivos utilizados en nuestra red. IPv6 nativo significa que los paquetes IPv6 fluirán sin ser cambiados o modificados a lo largo de toda su trayectoria, desde el origen hasta destino. Es común encontrar lo que se llama redes de doble pila, donde se utilizan las versiones IPv4 e IPv6 al mismo tiempo en las mismas interfaces y dispositivos. Esta posibilidad de IPv6 nativa cubre tanto sólo IPv6 como doble pila.

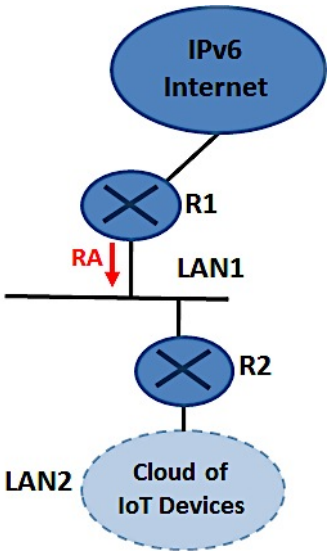

Figura 21. IPv6 Nativo

Como se ve en la figura, nuestra nube de dispositivos IoT está conectada a un enrutador (R2) que les proporciona un prefijo con lo que se crea una LAN (LAN2). El enrutador que proporciona conectividad a la Internet IPv6 (R1) se encargará de autoconfigurar los dispositivos IPv6 en LAN1, por ejemplo R2, enviando mensajes RA (Anuncios del Enrutador) como vimos en la explicación de SLAAC.

- **No hay conectividad IPv6**. En esta situación nos enfrentamos a un problema común en la actualidad, un ISP que no ofrece aún IPv6. Aunque tenemos el soporte de IPv6 en el enrutador que conecta nuestra red a la Internet, el ISP no proporciona conectividad IPv6, sino sólo IPv4. La solución es utilizar uno de los Mecanismos de Transición a IPv6. El más simple y útil en este caso sería el túnel 6in4, basado en la creación de un túnel estático de punto a punto que encapsula los paquetes IPv6 en IPv4.

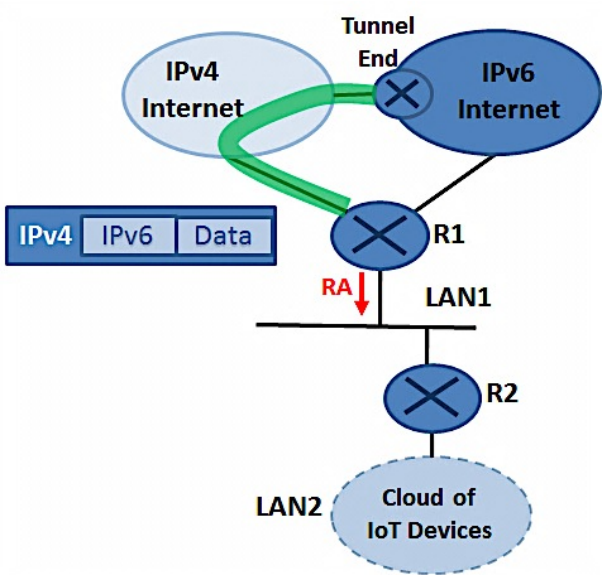

Figura 22. IPv6 en un túnel de IPv4

La figura muestra la solución con un túnel 6in4, creado desde R1 hasta un extremo del túnel, que será un enrutador que tiene conectividad tanto a Internet IPv4 como a Internet IPv6. El tráfico IPv6 nativo de nuestras redes (LAN1 y LAN2) alcanzará R1, que se llevará todo el paquete IPv6 con sus datos y lo colocará dentro de un nuevo paquete IPv4 con dirección de destino IPv4, la del enrutador. El enrutador al final del túnel arrancará el paquete IPv6 y lo pondrá como tráfico IPv6 nativo en la Internet IPv6. La misma encapsulación ocurre cuando el tráfico IPv6 se envía desde Internet IPv6 a nuestras redes.

- **No hay conectividad IPv6 y ningún enrutador para IPv6**. Este caso se presenta cuando el ISP no suministra ISP, ni existe soporte de IPv6 en el enrutador que conecta nuestra red a Internet. Como se vio en el caso anterior, para resolver la falta de conectividad IPv6 en el ISP pudimos utilizar un túnel 6in4, pero en el caso presente también tenemos que resolver la falta de compatibilidad IPv6 del enrutador que no será capaz de crear el túnel. La solución es añadir un nuevo enrutador que soporte IPv6 y IPv4, y crear un túnel 6in4 desde este enrutador hasta un enrutador del extremo del túnel en algún lugar de Internet IPv4.

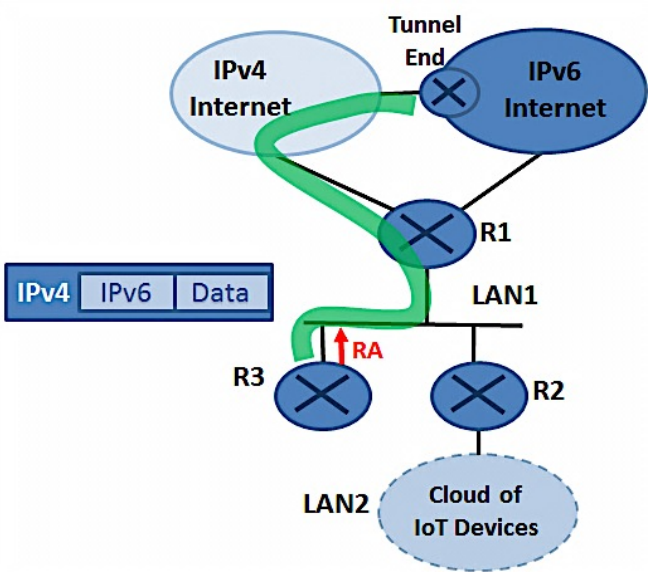

Figura 23. El enrutador local no procesa IPv6

En este caso, se añade un nuevo enrutador R3 para crear un túnel 6in4 hacia el enrutador del extremo del túnel y servir como una pasarela de enlace IPv6 hacia nuestras redes, enviando mensajes RA para configurar automáticamente los dispositivos IPv6 en LAN1. El proceso encapsulado / desencapsulado funcionará exactamente igual que en el caso anterior. La principal diferencia es que los túneles 6in4 necesitan una dirección IPv4 pública, por lo que R3 tendrá que tener una dirección IPv4 pública para poder crear el túnel 6in4. Esto es fácil de conseguir en los enrutadores conectados a los ISP, pero no tan común dentro de nuestra red donde podríamos tener direcciones privadas haciendo uso de NAT.

Los casos mostrados anteriormente se basan en una buena infraestructura, donde tenemos al menos dos enrutadores y un par de redes de área local. Las tres eventualidades podrían simplificarse en un solo caso en el que haya un solo enrutador como se muestra en la siguiente figura:

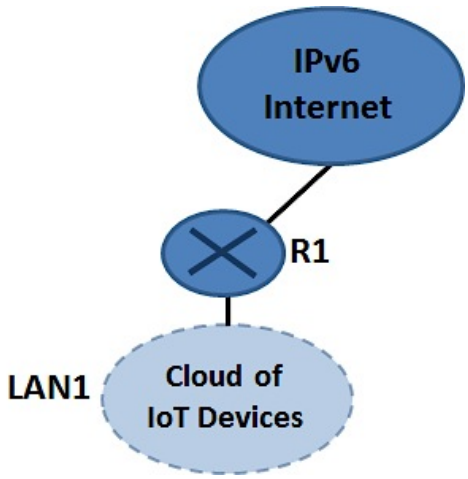

Figura 24. Escenario simplificado

Las consideraciones sobre la falta de conectividad IPv6 por parte del ISP y el soporte de IPv6 en el enrutador son las mismas, aunque para este último la solución es cambiar el único enrutador por uno que respalde IPv6.

Esto es común porque IoT ó WSN podrían ser desplegados en cualquier lugar, incluyendo redes apartadas que se conectan utilizando cualquier tecnología inalámbrica que pueda llegar a lugares remotos. En estos casos hay serias limitaciones sobre el número de dispositivos, el consumo de energía, etc. Por ejemplo, una nube de sensores podría ser desplegada en el país para detectar la temperatura y la humedad, y todos ellos obtendrían conectividad a través de un solo enrutador IPv6 conectado usando la red de telefonía móvil (GPRS, 3G o LTE).

Introducción a 6LoWPAN

Uno de los impulsores de la IoT, donde cualquier cosa puede ser conectada, es el uso de tecnologías inalámbricas para realizar un canal de comunicación por donde enviar y recibir información. La creciente adopción de tecnología inalámbrica permite incrementar el número de dispositivos conectados pero a su vez introduce limitaciones en términos de costo, duración de las baterías, consumo de energía y distancias máximas entre dispositivos. Las nuevas tecnologías y protocolos deberán desenvolverse en un ambiente nuevo llamado Redes de Baja Potencia y sujetas a Pérdidas (*Low Power and Lossy Networks*: **LLN**) que tiene las características siguientes:

- Permiten muchos más dispositivos que las redes de área local actuales.
- Severa limitación del espacio para código y RAM en los dispositivos.
- Limitaciones de distancia máxima alcanzable, potencia consumida y recursos de procesamiento.
- Todos los elementos deben trabajar conjuntamente para optimizar el consumo de energía y del ancho de banda.

Otra característica que ha sido ampliamente adoptada por la IoT es el uso de IP como protocolo de red. El uso de IP presenta varias ventajas porque es un estándar abierto ampliamente disponible que permite su implementación de manera barata y fácil, ofrece buena interoperabilidad y fácil desarrollo de la capa de aplicación. El uso de un estándar común, basado en IP de extremo a extremo evita el problema de las redes interconectadas por protocolos que no son inter-operables.

Para las capas inferiores (de enlace y física) del estándar IEEE 802.15.4 [IEEE 802.15.4], LLN presenta ventajas, sin embargo, hay otros contendores, como Low Power WiFi, Bluetooth® Low Energy, DECT Ultra Low Energy, ITU-T G 9959 networks, o NFC (Near Field Communication).

Una de las componente de la IoT que ha recibido un respaldo importante por parte de los vendedores y de las organizaciones estandarizadoras son las WSN (*Wireless Sensor Networks*).

La IETF (*Internet Engineering Task Force*) tiene diferentes grupos de trabajo (*working groups*: WGs) desarrollando estándares que serán usados por las WSN:

1. **6LoWPAN**: IPv6 sobre Redes Inalámbricas de Área Personal y Baja Potencia (*Low-power Wireless Personal Area Networks* [6LoWPAN]) definió el estándar para comunicación IPv6 con la tecnología de comunicación inalámbrica IEEE 802.15.4. 6LoWPAN actúa como una capa de adaptación entre el mundo del estándar IPv6 y el medio inalámbrico de baja potencia y comunicación con pérdidas (*lossy*) ofrecido por el IEEE 802.15.4. Nótese que este estándar está definido con IPv6 en mente, es decir no funciona con IPv4.
2. **roll**: Enrutamiento en Redes de Baja Potencia con Pérdidas (*Routing Over Low power and Lossy networks*) [roll]) Las redes LLN tienen requisitos de enrutamiento específicos que podrían no ser satisfechos por los protocolos de enrutamiento existentes. Este Grupo de Trabajo se enfoca en soluciones de enrutamiento para un subconjunto de todas las áreas de aplicación de LLN posibles (industrial, conexiones en el hogar, redes de sensores en

edificios y en ciudades) y los protocolos se diseñan para satisfacer los requisitos de enrutamiento específicos de cada aplicación. De nuevo, el Grupo de Trabajo se enfoca solamente en enrutamiento para IPv6.

3. **6lo**: IPv6 en Redes de Nodos de Recursos Restringidos (*Networks of Resource-constrained Nodes* [6lo]). Este Grupo de Trabajo se ocupa de la conectividad IPv6 en redes de nodos restringidos en recursos. Amplía la labor del Grupo de Trabajo 6LoWPAN definiendo las especificaciones de la capa de adaptación *IPv6-over-foo* (IPv6 sobre lo que sea) usando 6LoWPAN para la capa de enlace en las redes de nodos de recursos limitados.

Como vemos, 6LoWPAN es la base del trabajo de estandarización que se lleva a cabo en la IETF para comunicar nodos de recursos restringidos en las LLN usando IPv6. El trabajo sobre 6LoWPAN ha finalizado y está siendo complementado por el Grupo de Trabajo *roll* para satisfacer las necesidades de enrutamiento y por el Grupo de Trabajo *6lo* para extender los estándares 6LoWPAN a cualquier tecnología de la capa de enlace. A continuación daremos más detalles sobre 6LoWPAN como un primer paso hacia la Red Inalámbrica de Sensores/IoT (WSN/IoT) basada en IPv6. El 6LoWPAN y otros estándares están dirigidos a proporcionar conectividad IP a los dispositivos, independientemente de las capas superiores, a excepción de la capa de transporte UDP que es tomada en cuenta específicamente.

Visión panorámica de las LoWPAN

Redes de baja potencia con pérdidas (LLN en inglés) es el término comúnmente usado para referirse a redes formadas por nodos con severas restricciones (limitaciones de CPU, memoria y potencia) interconectados por una variedad de enlaces "con pérdidas" (radio-enlaces de baja potencia). Se caracterizan por tener baja velocidad, bajas prestaciones, bajo costo y conectividad inestable.

Una LoWPAN es un ejemplo particular de LLN formada por dispositivos que adhieren al estándar 802.15.4 de la IEEE.

Las características resaltantes de los dispositivos de una LoWPAN son:

1. **Capacidad limitada de procesamiento**: procesadores de diferentes velocidades y diferentes tipos, partiendo desde 8 bits.

2. **Poca capacidad de memoria**: desde unos pocos kilobytes de RAM y unas docenas de kilobytes de ROM o memoria flash. Se espera que en el futuro las prestaciones mejoren, pero siempre serán limitadas.

3. **Baja potencia**: del orden de decenas de milivatios.

4. **Alcance corto**: el Espacio Personal de Operación (*Personal Operating Space*: POS) definido por el IEEE 802.15.4 sugiere un alcance de 10 metros. En implementaciones reales puede alcanzar hasta 100 metros cuando hay línea de vista.

5. **Bajo costo**: esto determina algunas de las limitaciones anteriores, como bajo procesamiento, poca memoria, etc.

A medida que la tecnología avanza, se espera que se reduzcan estas restricciones en los nodos, pero comparando con otros campos es de prever que las LoWPAN traten siempre de usar dispositivos muy restringidos para mantener los bajos precios y prolongar la vida útil, lo que siempre implica limitaciones en las demás características.

Una LoWPAN normalmente tiene dispositivos que trabajan conjuntamente para conectar el ambiente físico a aplicaciones del mundo real, por ejemplo sensores inalámbricos pero también puede contener actuadores.

Es muy importante identificar las características de una LoWPAN por las restricciones que imponen en el trabajo técnico:

1. Paquetes pequeños: Dado que la trama máxima de la capa física es de 127 bytes, la trama máxima resultante en la capa de control de acceso al medio es de 102 octetos. La seguridad de la capa de enlace impone una tara adicional, lo que deja un máximo de 81 octetos por paquete de datos.
2. IEEE 802.15.4 establece varias modalidades de direcciones: permite bien sea las direcciones extendidas de 64 bits del IEEE, o también (después de un evento de asociación) direcciones de 16 bit únicas dentro de la PAN (*Personal Area Network*).
3. Reducido ancho de banda: Velocidad de datos de 250, 40 y 20 kbps para cada una de las capas físicas definidas (2.4 GHz, 915 MHz, y 868 MHz, respectivamente).
4. La topología puede ser en estrella o en malla.
5. Se espera el despliegue de un gran número de dispositivos durante la vida útil de la tecnología. La ubicación de los dispositivo no está normalmente predefinida, ya que la tendencia es desplegarlos de manera *ad hoc*. Algunas veces la ubicación de los dispositivos puede ser de difícil acceso o estos pueden cambiar de ubicación.
6. Los dispositivos dentro de una LoWPAN son poco fiables por múltiples razones: conectividad de los radios poco confiable, baterías descargadas, dispositivos que se trancan (*lockup*), vandalismo, etc.
7. Dispositivos dormidos: Los dispositivos pueden permanecer en modo "durmiente" (*sleeping mode*) por largos periodos para ahorrar energía, durante los cuales no pueden comunicarse.

Uso de IP en LoWPAN

En esta sección veremos las ventajas y algunos problemas que surgen del uso de IP en las redes LoWPAN.

La aplicación de tecnología IP, en particular IPv6, debería prestar los siguientes beneficios a las LoWPAN:

a. La naturaleza ubicua de las redes IP permite el aprovechamiento de infraestructuras existentes.
b. Las tecnologías basadas en IP ya existen, son bien conocidas, de funcionamiento comprobado y disponibles en muchas partes. Esto va a permitir una adopción fácil y barata, una buena interoperabilidad y un desarrollo de las capas de aplicación más fácil.
c. La tecnología de redes IP está basada en especificaciones abiertas y gratuitas lo que permite que sean mejor entendidas por parte de una audiencia mas amplia que la de las soluciones patentadas (*proprietaries*).
d. Las herramientas para redes IP ya existen.
e. Los dispositivos basados en IP pueden conectarse inmediatamente a otras redes IP sin la necesidad de entidades intermedias como pasarelas de traducción de protocolos o proxies.
f. El uso específico de IPv6 permite una cantidad enorme de direcciones y facilita la autoconfiguración de los parámetros de red usando (SLAAC). Esto es esencial para las 6LoWPAN, que deben manejar un gran número de dispositivos.

Por otra parte, usar comunicación IP en las LoWPAN conlleva ciertos problemas que deben tomarse en cuenta:

a. Cabeceras IP. Una de las características de las 6LoWPAN es la limitación en el tamaño de los paquetes, lo que significa que las cabeceras (*headers*) para IPv6 y las capas superiores deben comprimirse siempre que sea posible.
b. Topologías. Las LoWPAN deben permitir varias topologías incluidas las de malla y estrella. Las tipologías en malla implican enrutamiento multi-salto hasta el destino deseado. En este caso, los dispositivos intermedios actúan como repetidores de paquetes en la capa de enlace. Las tipologías de estrella suponen el manejo de un subconjunto de dispositivos con capacidad de expedición de paquetes. Si, además de IEEE 802.15.4, estos dispositivos usan también otro tipo de interfaces de red, como IEEE 802.3 (Ethernet) o IEEE 802.11, la meta es integrar fluidamente estas redes construidas con tecnologías diferentes. Eso, por supuesto, es otra motivación para el uso de IP.
c. Tamaño limitado de paquetes. Se espera que las aplicaciones en redes LoWPAN originen paquetes pequeños. Los añadidos de todas las demás capas de conectividad IP deberían todavía caber en una sola trama sin recurrir a excesivas fragmentaciones y reensamblajes. Los protocolos también deben diseñarse o escogerse de manera que los paquetes de control quepan en una sola trama 802.15.4.
d. Configuración y Manejo Limitados. Es de esperar que las redes LoWPAN contengan una enorme cantidad de dispositivos. Y también se espera que estos tengan una capacidad limitada de despliegue y de entrada de datos. Además, la ubicación de algunos de estos dispositivos puede ser difícil de alcanzar. En consecuencia, los protocolos que se usen en las LoWPAN deberían requerir una configuración mínima, estar listos para su uso inmediato, ser de fácil arranque y, además, capacitar a la red para la auto-restauración dada la característica inherente de poca confiabilidad de estos dispositivos.
e. Descubrimiento de Servicios. Las LoWPAN necesitan protocolos simples de descubrimiento de servicios de la red para descubrir, controlar y mantener los servicios disponibles.

f. Seguridad. IEEE 802.15.4 exige que la seguridad de la capa de enlace esté basada en Advanced Encryption Standard (AES), pero no da detalles sobre temas como el arranque (*bootstraping*), manejo de claves o seguridad en las capas superiores. Obviamente, una solución de seguridad para los dispositivos LoWPAN debería considerar muy seriamente las necesidades de las aplicaciones.

6LoWPAN

Hemos visto que hay capas inferiores (las capas física y de enlace en la pila TCP/IP) que proporcionan conectividad a los dispositivos en las redes LoWPAN. También que usar IPv6 encima de estas capas comporta varias ventajas. La razón principal del desarrollo del estándar 6LoWPAN por la IETF es la introducción de una capa de adaptación entre la capa de red (IP) y la capa inferior con finalidad de resolver asuntos importantes.

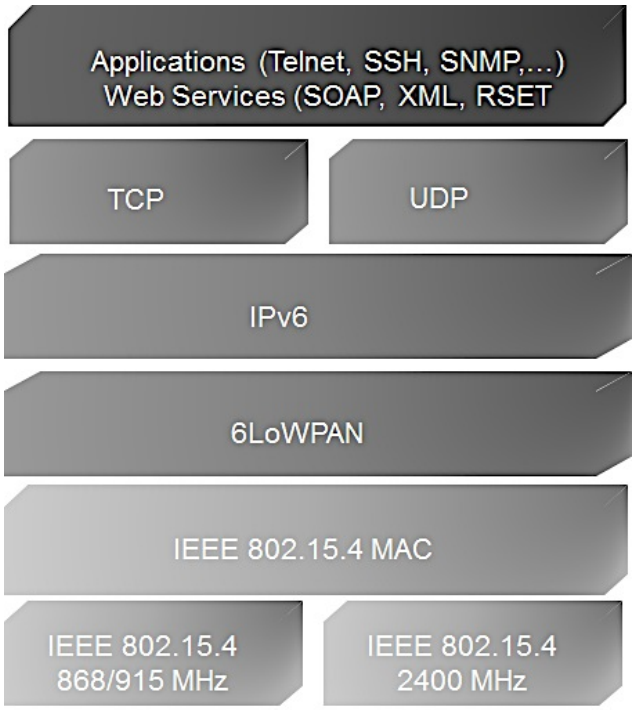

Figura 25. 6LoWPAN en la pila de protocolos

Los objetivos principales de 6LoWPAN son:

1. Capa de segmentación y reensamblaje: La especificación de IPv6 [RFC2460] establece que la MTU (*Maximum Transmission Unit*) más pequeña que la capa de enlace debería ofrecerle a la capa IPv6 es de 1280 bytes. Sin embargo, las PDU (unidades del protocolo de datos) en IEEE 802.15.4 pueden ser de apenas 81 bytes. Para resolver esta diferencia se debe proporcionar una capa de adaptación para la fragmentación y reensamblaje en la capa debajo de IP.

2. Compresión de cabecera: Dado que en el peor de los casos el tamaño máximo disponible para transmisión de paquetes IP en una trama IEEE 802.15.4 es de 81 octetos, y que la longitud de la cabecera IPv6 es de 40 octetos (sin cabeceras extendidas opcionales), esto nos deja sólo 41 octetos para las capas superiores del protocolo, como UDP y TCP. La cabecera UDP usa 8 octetos y la cabecera TCP usa 20 octetos. Esto deja 33 octetos para datos en UDP y 21 octetos para datos en TCP. Además, como se señaló antes, se necesita una capa para fragmentación y reensamblaje lo que utilizará más octetos y dejará menos espacio para los datos. Entonces, si uno fuera a usar el protocolo como está, se estaría haciendo un uso excesivo de la fragmentación y reensamblaje, incluso cuando los paquetes tengan apenas decenas de octetos de largo. Esto explica la necesidad de comprimir la cabecera.

3. Auto-configuración de direcciones: Especifica métodos para la autoconfiguración de direcciones IPv6 sin estado (en contraposición a direcciones de estado completo), lo que es atractivo para redes 6LoWPAN porque reduce la tara de configuración en los hosts. Se necesita un método que genere la IID (*Interface Identifier*) IPv6 a partir del EUI-64 (*Extended Unique Identifier*) asignado al dispositivo IEEE 802.15.4.

4. Protocolo de Enrutamiento en Malla: Es necesario un protocolo de enrutamiento compatible con una red mallada multi-salto. Debe tenerse cuidado al usar los protocolos de enrutamiento existentes (o al diseñar nuevos protocolos) a fin de que los paquetes de enrutamiento quepan en una sola trama IEEE 802.15.4. Los mecanismos definidos en 6LoWPAN se basan en los requisitos de la capa IEEE 802.15.
5. IEEE 802.15.4. define cuatro tipos de tramas: tramas de baliza, de MAC, de reconocimiento y de datos. Los paquetes Ipv6 deben ser transportados en la trama de datos.

6. Las tramas de datos pueden opcionalmente solicitar el reconocimiento (*acknowledge*). Se recomienda que los paquetes IPv6 se transporten en tramas que tengan solicitud de reconocimiento para ayudar a la recuperación en la capa de enlace (*link-layer*).

7. La especificación permite tramas en las cuales las direcciones de origen y destino (o ambas) se omitan. Ambas direcciones, la de origen y la de destino, deben ser incluidas en la trama de cabecera IEEE 802.15.4.

8. Los campos de origen y destino del ID de la PAN tienen que incluirse también. El estándar 6LoWPAN presupone que una PAN se mapea a un enlace IPv6 específico.

9. Se permiten tanto las direcciones extendidas de 64 bits, como las reducidas de 16 bits, sin embargo, hay restricciones de formato para el uso de las direcciones cortas de 16 bits.

10. La multidifusión *per se* no es compatible con IEEE 802.15.4. Por lo tanto, los paquetes multidifusión IPv6 deben transportarse como tramas de difusión (*broadcast*) en las redes IEEE 802.15.4. Esto debe hacerse de manera que las tramas de difusión sean atendidas sólo por dispositivos dentro de la PAN específica del enlace en cuestión.

El formato de adaptación de 6LoWPAN fue especificado para el transporte de datagramas IPv6 en enlaces con restricciones, tomando en cuenta limitaciones en los recursos de ancho de banda, memoria o energía que esperables en aplicaciones como las redes inalámbricas de sensores. Para cada uno de estos objetivos y requisitos hay una solución aportada por las especificaciones 6LoWPAN:

1. Una cabecera de direccionamiento en malla para permitir reenvío sub-IP.

2. Una cabecera de fragmentación para cumplir con la MTU mínima requerida por IPv6.

3. Una cabecera de difusión (*broadcast*) a ser usada cuando se deban enviar paquetes de multidifusión en redes IEEE 802.15.4.

4. Compresión de cabeceras libre de estado para reducir las cabeceras IPv6 y UDP en los datagramas IPv6 a unos pocos bytes (en el mejor de los casos). Estas cabeceras se usan para la encapsulación LoWPAN y podrían usarse al mismo tiempo conformando lo que se llama la pila de cabeceras (*header stack*). Cada cabecera en la pila contiene un campo *tipo de cabecera* que puede ser seguido por campos de cabecera adicionales. Cuando se usa más de una cabecera LoWPAN en el mismo paquete, éstas tienen que aparecer en el orden siguiente: Cabecera de direccionamiento en malla, cabecera de difusión y cabecera de fragmentación.

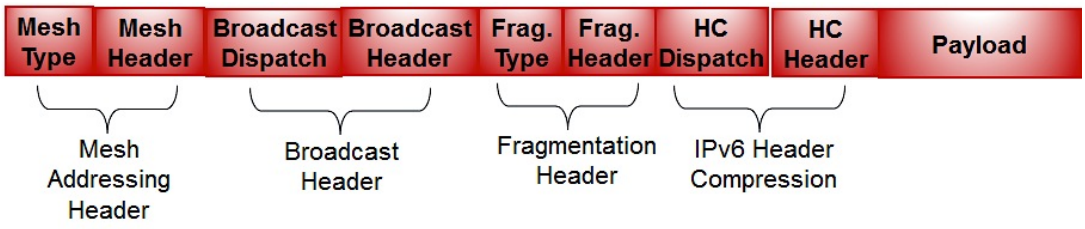

Figura 26. Cabeceras de 6LoWPAN

Identificador de Interfaz IPv6 (IID)

Como hemos mencionado, un dispositivo IEEE 802.15.4. puede tener dos tipos de direcciones que tienen formas diferentes para generar la IID IPv6.

1. Dirección IEEE EUI-64: Todos los dispositivos la tienen. En este caso, el Identificador de Interfaz se forma a partir de EUI-64, complementando el bit "Universal/Local" (U/L) que es el más cercano al bit de orden más bajo del primer octeto de EUI-64. Al complementar este bit generalmente se producirá un cambio de un valor 0 a un valor 1.

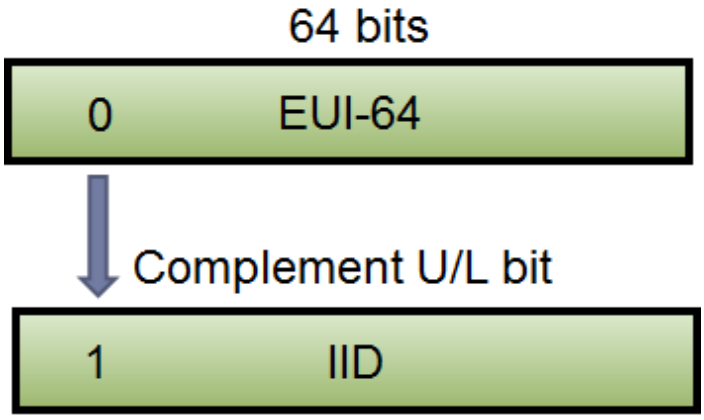

Figura 27. IID desde EUI-64

2. Direcciones de 16 bits: Son posibles, pero no se usan siempre. La IID IPv6 se forma usando la PAN (o ceros, en caso de no conocerse la PAN) y la dirección de 16 bits como en la figura siguiente:

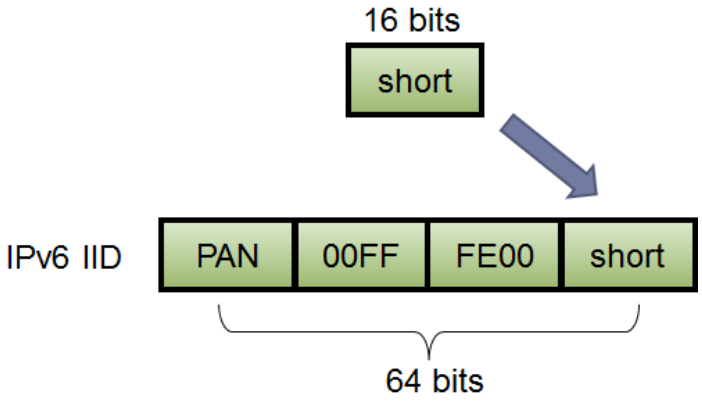

Figura 28. IPv6 IID

Compresión de cabecera

Existen dos formatos de encriptación para la compresión de los paquetes IPv6: LOWPAN_IPHC y LOWPAN_NHC, usado para codificar cabeceras siguientes (*next header*) arbitrarias.

Para realizar una compresión efectiva, LOWPAN_IPHC se basa en información que abarca la totalidad de la red 6LoWPAN. LOWPAN_IPHC asume que en caso más común se cumple siguiente:

1. La versión es 6.

2. La Clase de Tráfico y la Etiqueta de Flujo son ambas cero.

3. La longitud de la carga útil (*payload*) se puede inferir a partir de las capas inferiores, bien sea desde la cabecera de fragmentación 6LoWPAN o de la cabecera IEEE 802.15.4.

4. El límite de número de saltos habrá sido fijado a un valor bien conocido por la fuente.

5. Las direcciones asignadas a las interfaces 6LoWPAN se formarán usando el prefijo de enlace local (link-local) o un subconjunto de prefijos enrutables asignados a toda la 6LoWPAN.

6. Las direcciones asignadas a las interfaces 6LoWPAN se forman con una IID derivado directamente bien sea de la dirección extendidas de 64 bits o de la corta de 16 bits. Dependiendo de qué tan cerca esté un paquete de estas especificaciones que son el caso común, los diferentes campos podrían no ser comprimibles por lo que necesitarían ser también transportados "en-línea". El formato básico usado en la encriptación LOWPAN_IPHC se muestra a continuación:

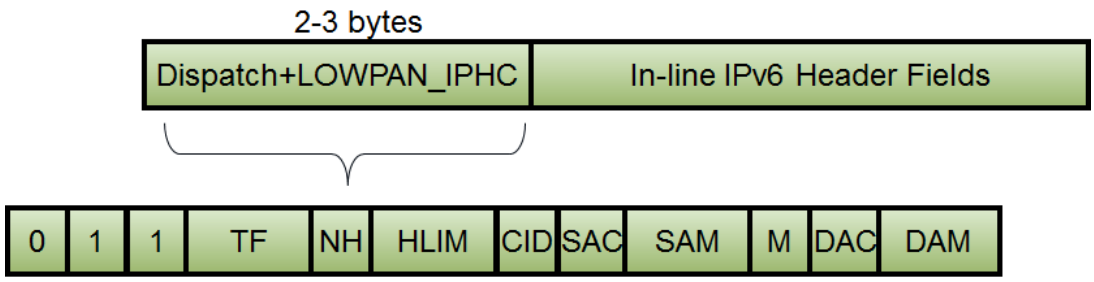

Figura 29. Compresión de cabecera

Donde:

- TF: Clase de Tráfico, Etiqueta de flujo.
- NH: Cabecera siguiente.
- HLIM: Límite de número de saltos.
- CID: Extensión del Identificador de Contexto (*Context Identifier Extension*).
- SAC: Compresión de la Dirección de Origen (*source*).
- SAM: Modo de la Dirección de Origen.
- M: Compresión multicast.
- DAC: Compresión de la Dirección de Destino

- DAM: Modo de la Dirección de Origen.

Sin entrar en detalles, es importante entender cómo funciona la compresión 6LoWPAN. Para ello, veamos dos ejemplos:

1. HLIM (Límite del número de saltos) es un campo de dos bits que puede tener cuatro valores, tres de los cuales comprimen de 8 a 2 bits:

i. 00: Campo de límite de saltos transportado en-línea. No hay compresión, y el campo completo es transportado en-línea después de LOWPAN_IPHC.
ii. 01: El campo de límite de saltos es comprimido y el límite de saltos es 1.
iii. 10: El campo de límite de saltos es comprimido y el límite de saltos es 64.
iv. 11: El campo de límite de saltos es comprimido y el límite de saltos es 255.

2. SAC/DAC usado para la compresión de la dirección IPv6 del origen. SAC indica cuál compresión de dirección se utiliza: sin estado (SAC=0) o de estado completo (*stateful*) basada en el contexto (SAC=1). Dependiendo de SAC, DAC se usa de la manera siguiente:

i. Si SAC=0, entonces para SAM tendremos:

- 00: 128 bits. La dirección completa se transporta en-línea. No hay compresión.

- 01:64 bits. Los primeros 64 bits de la dirección se eliminan, el prefijo de enlace local. Los restantes 64 bits son transportados en-línea.

- 10:16 bits. Los primeros 112 bits de la dirección se eliminan. Los primeros 64 bits son el prefijo de enlace local. Los siguientes 64 bits son 0000:00ff:fe00:XXXX, donde XXXX son los 16 bits transportados en-línea.

- 11:0 bits. La dirección es completamente eliminada. Los primeros 64 bits de la dirección son el prefijo de enlace local. Los restantes 64 bits se computan a partir de la cabecera de encapsulación (por ejemplo, 802.15.4 o la dirección de origen IPv6).

ii. Si SAC=1, entonces para SAM tendremos:

- 00: 0 bits. La dirección no especificada (::).

- 01: 64 bits. La dirección se deriva usando información del contexto y los 64 bits transportados en-línea. Los bits cubiertos por la información del contexto se usan siempre. Cualesquiera bits de la IID no cubiertos por la información del contexto se toman directamente de los correspondientes bits transportados en-línea.

- 10: 16 bits. La dirección se deriva usando la información del contexto y los 16 bits transportados en-línea. Los bits cubiertos por la información del contexto se usan siempre. Cualesquiera bits de la IID no cubiertos por la información del contexto se forman directamente del mapeo de 16 bits a IID dado por 0000:00ff:fe00:XXXX, donde XXXX son los 16 bits transportados en-línea.

- 11: 0 bits. La dirección se elimina completamente y luego se deriva usando información del contexto y la cabecera de encapsulamiento (por ejemplo 802.15.4 o la dirección IPv6 de la fuente). Los bits cubiertos por la información del contexto se usan siempre. Cualesquiera bits de IID no cubiertos por la información del contexto se computan a partir de la cabecera de encapsulamiento.

El formato de base es de dos bytes (16 bits) de largo. Si el campo CID (*Context Identifier Extension*) tiene un valor de 1, significa que tendremos un campo adicional de 8 bits (*Context Identifier Extension*) inmediatamente después de DAM (*Destination Address Mode*). Esto haría que la longitud sea de 24 bits o tres bytes.

Este octeto adicional identifica el par de contextos a ser utilizados cuando se comprime la dirección IPv6 de fuente y/o destino. El identificador de contexto es de 4 bits para cada dirección, permitiendo hasta 16 contextos. El contexto 0 es el contexto predeterminado. Los dos campos en la Extensión del Identificador de Contexto (*Context Identifier Extension*) son:

- SCI: Identificador de Contexto del Origen (*Source Context Identifier*). Identifica el prefijo que se usa cuando la dirección IPv6 del origen está comprimida en estado completo (*statefully*).

- DCI: Identificador de Contexto del Destino (*Destination Context Identifier*). Identifica el prefijo que se usa cuando la dirección IPv6 del destino está comprimida en estado completo.

El campo Próxima Cabecera en la cabecera Ipv6 se trata de dos maneras diferentes dependiendo de los valores indicados en el campo NH (*Next Header*: Próxima Cabecera) de la codificación LOWPAN_IPHC que se mostró antes.

Si NH=0, entonces este campo no se comprime y todos los 8 bits son transportados en-línea después de LOWPAN_IPHC.

Si NH=1, el campo Próxima Cabecera se comprime y la próxima cabecera se codifica usando LOWPAN_NHC. Los resultados se ven en la figura siguiente.

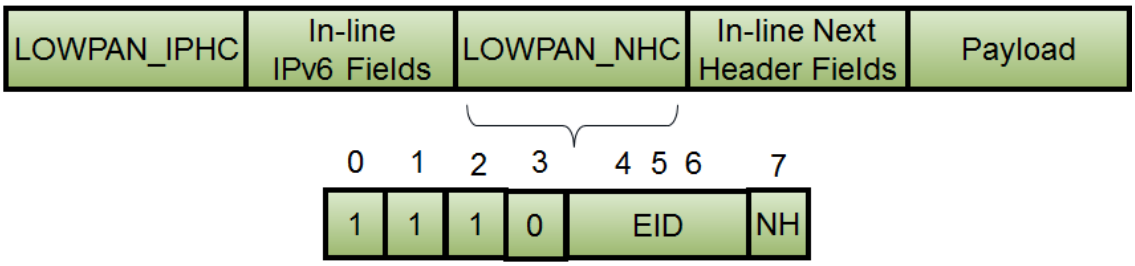

Figura 30. Cabecera LoWPAN

Para las cabeceras extendidas IPv6 el LOWPAN_NHC tiene el formato que se muestra en la figura, donde:

- EID: Extension Header ID de IPv6:
  - 0: Cabecera IPv6 opción Salto a Salto (*Hop-by-Hop*)
  - 1: Cabecera IPv6 de enrutamiento.
  - 2: Cabecera IPv6 de fragmentación
  - 3: Cabecera IPv6 para opciones del destino.
  - 4: Cabecera IPv6 de movilidad.
  - 5: Reservado.
  - 6: Reservado.
  - 7: Cabecera IPv6.

- NH: Next Header (Próxima Cabecera)

  - 0: Los 8 bits completos para Próxima Cabecera son transportados en-línea.

  - 1: El campo Próxima Cabecera se borra y se codifica usando LOWPAN_NHC. En su mayor parte, la Cabecera Extendida se transporta sin modificaciones en los bytes que van inmediatamente después del octeto LOWPAN_NHC.

Optimización de NDP

IEEE 802.15.4 y otras tecnologías de capa de enlace semejantes tienen un uso limitado o inexistente de señales multicast a fin de conservar energía. Además, la red inalámbrica puede no seguir estrictamente el concepto de subredes IP y enlaces IP. El descubrimiento de vecinos (*Neighbor Discovery*) de IPv6 no fue diseñado para enlaces inalámbricos no-transitivos, ya que su dependencia del concepto tradicional de IPv6 y su uso excesivo de multicast lo hacen ineficiente y poco práctico en una red con pérdidas y de poca potencia.
Por esta razón, se han realizado algunas optimizaciones sencillas para el *Neighbor Discovery* de IPv6, mecanismos de direccionamiento y detección de direcciones duplicadas para redes LoWPAN [RFC6775]:

1. Interacción auto-iniciada por el host para permitir hosts “dormidos”.

2. Eliminación de resolución de direcciones para hosts basadas en multicast.

3. Una forma de registro de dirección mediante una nueva opción en mensajes unicast de NS (*Neighbor Solicitation* - Solicitud de Vecino) y NA (*Neighbor Advertisement* - Anuncio de Vecino).

4. Una nueva opción de ND (Descubrimiento de Vecino) para distribuir el contexto de compresión de cabecera 6LoWPAN a los hosts.

5. Distribución multi-salto de prefijo y contexto de compresión de cabecera 6LoWPAN.

6. Detección de Direcciones Duplicadas multi-salto (DAD), que usa dos nuevos tipos de mensaje ICMPv6.

Los dos elementos multi-salto pueden sustituirse por un mecanismo de protocolo de enrutamiento si se desea.

Se definen tres nuevas opciones de mensajes ICMPv6:

1. La ARO, Opción Registro de dirección (*Address Registration Option*).

2. La ABRO, Opción de Enrutador de Frontera Acreditado (*Authoritative Border Router Option*).

3. La Opción 6CO, Contexto 6LoWPAN (*6LoWPAN Context Option*).

También se definen dos tipos nuevos de mensajes ICMPv6:

1. DAR, Solicitud de Dirección Duplicada (*Duplicate Address Request*).

2. DAC, Confirmación de Dirección duplicada (*Duplicate Address Confirmation*).

Introducción a Contiki

Contiki es un sistema operativo de código abierto para Internet de las Cosas, conecta microcontroladores de bajo costo y de bajo consumo a Internet.

Contiki ofrece una comunicación a Internet potente y de bajo consumo, es completamente compatible con IPv6 e IPv4 y con los últimos estándares inalámbricos para dispositivos de bajo consumo: 6LoWPAN, RPL, COAP. Utilizando ContikiMAC, incluso dispositivos enrutadores pueden operar a baja potencia y ser alimentados por baterías.

Con Contiki, el desarrollo es fácil y rápido: las aplicaciones de Contiki están escritas en C estándar; con el emulador Cooja las redes Contiki pueden ser simuladas antes de programarlas en hardware; Instant Contiki proporciona un entorno de desarrollo completo en una sola descarga.

Más información disponible en:

http://contiki-os.org/

El resto de esta sección se propone dar una introducción a Contiki 3.0, sus componentes y características centrales. Los siguientes sitios son las referencias más útiles para buscar información detallada y respuestas a dudas:

- La página wiki de Contiki. https://github.com/contiki-os/contiki/wiki
- La lista de correos de Contiki. http://sourceforge.net/p/contiki/mailman/contiki-developers/
- La página de recursos de Zolertia. http://zolertia.io/resources

Por último, pero no menos importante, este libro no habría sido posible sin la Comunidad Contiki, demasiado grande para nombrarlos a todos (más de 102 colaboradores para la fecha de este documento). Un reconocimiento especial a Adam Dunkels, George Oikonomou, Enric Calvo, Joakim Eriksson, Marcus Lundén y Niclas Finne, quienes han contribuido al desarrollo de las plataformas Zolertia.

Instale Contiki

Hay varias formas de instalar Contiki: desde cero a partir de las fuentes, o usando entornos virtuales. La escogencia dependerá de los gustos y del tiempo que estemos dispuestos a gastar.

Para trabajar con Contiki se requieren tres cosas:

- Obtener el código fuente Contiki.
- Una plataforma (plataforma virtual o un hardware real).

- Un entorno de desarrollo para compilar el código fuente para la plataforma.

El resto del libro presupone que Contiki se ejecutará en un entorno Unix, ya que los entornos virtualizados se ejecutan específicamente en Ubuntu. Sin embargo, luego se proporcionan las instrucciones para Windows y MAC.

Instalación desde las fuentes

El código fuente de Contiki es apoyado activamente por distintos colaboradores de universidades, centros de investigación y desarrolladores de todo el mundo.

El código fuente se encuentra alojado en GitHub:

https://github.com/contiki-os/contiki

Para extraer el código fuente, abre un terminal y escribe la instrucción siguiente:

```
sudo apt-get -y install git
git clone --recursive https://github.com/contiki-os/contiki.git
```

Qué es GIT

Git es un software de control de versiones gratuito y de código abierto diseñado para manejar desde pequeños a grandes proyectos con rapidez y eficiencia. La principal diferencia entre Git y otras herramientas previas de control de cambios es la posibilidad de trabajar localmente ya que la copia local es también un repositorio y así puedes obtener todos los beneficios de un control de fuente.

Figura 31. Git: sistema de control de versiones. Imagen tomada de https://git-scm.com/

Crear nuevas ramas o fusionar existentes es realmente fácil usando Git. Existen buenos tutoriales en línea para aprender más sobre GIT:

http://try.github.io

http://rogerdudler.github.io/git-guide/

GitHub es un servicio de alojamiento de repositorios GIT que ofrece las funciones de control de las revisiones distribuidas y el manejo de los códigos fuente (SCM) propios de Git, añadiendo a su vez una serie de características propias. GitHub proporciona una interfaz gráfica basada en Web y también una integración a teléfono móvil y *desktop*. Proporciona control de acceso y algunas características de colaboración como wikis, manejo de tareas, rastreo de fallos, y solicitudes de inclusión de funcionalidades para cada proyecto.

Figura 32. Github: plataforma de repositorios GIT. Imagen tomada de http://www.ryadel.com/

La ventaja de usar GIT y de alojar el código en Github es que le permite a la gente bifurcar el código, aportar modificaciones y luego compartir los cambios hechos.

Para instalar un entorno de desarrollo y las dependencias necesarias, abre un terminal y escribe:

```
sudo echo "deb http://ppa.launchpad.net/terry.guo/gcc-arm-embedded/ubuntu
trusty main" > /etc/apt/sources.list.d/gcc-arm-embedded.list
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key
FE324A81C208C89497EFC6246D1D8367A3421AFB
sudo apt-get update sudo apt-get -y install build-essential automake gettext
sudo apt-get -y install gcc-arm-none-eabi curl graphviz unzip wget
sudo apt-get -y install gcc gcc-msp430
sudo apt-get -y install openjdk-7-jdk openjdk-7-jre ant
```

Esto instalará el código para las plataformas ARM Cortex-M3 y MSP430 y para Cooja, el simulador de Contiki que discutiremos en las próximas secciones.

La máquina virtual Instant Contiki

Instant Contiki es un ambiente completo de desarrollo Contiki en una sola descarga. Es una máquina virtual Ubuntu (Linux) y tiene ya preinstalados el sistema operativo Contiki, las herramientas de desarrollo, los compiladores y simuladores que se necesitan.

Descarga Instant Contiki de la página web de Contiki: http://www.contiki-os.org/start.html

La última edición de Instant Contiki es la 3.0. Este libro se basa en esta, por lo que no se recomienda el uso de versiones anteriores.

La etiqueta de esta versión se encuentra en: https://github.com/contiki-os/contiki/tree/release-3-0

En sistemas Windows y Linux puedes usar VMWare para ejecutar la Máquina Virtual:

https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0

En OSX puedes descargar VMWare Fusion: http://www.vmware.com/products/fusion

Usando VMWare, abre la carpeta InstantContiki2.7.vmx. Si te pregunta sobre la procedencia de la VM, escoge la opción "I copied it" Luego espera el lanzamiento del Ubuntu virtual.

Regístrate en Instant Contiki. La clave y el nombre del usuario es user. No hagas una actualización todavía.

Recuerda actualizar el repositorio Contiki y de obtener las últimas actualizaciones en:

```
cd /home/user/contiki
git fetch origin
git pull origin master
```

Receta de Vagrant

Figura 33. Vagrant: entornos de desarrollo virtuales. Imagen tomada de vagrantup.com

Vagrant crea y configura ambientes de desarrollo reproducibles, livianos y portables, proporcionando todo lo necesario para empezar a trabajar con Contiki. Vagrant es gratis y disponible en todas las plataformas: https://github.com/Zolertia/zolertia_vagrant

Sigue las instrucciones del README para la instalación y ejecución.

Contenedor Docker

Figura 34. Docker: entornos de desarrollo virtualizados. Imagen tomada de docker.com

Docker es un proyecto de código abierto para empacar, enviar y ejecutar cualquier aplicación como un contenedor (*container*) liviano. También es portable a todas las plataformas, lo que simplifica trabajar en cualquier ambiente.
El repositorio Docker de Zolertia está disponible en: https://github.com/Zolertia/zolertia_docker

Siga las instrucciones del `README` para instalar.

Prueba de la instalación de Contiki

En primer lugar, vamos a chequear la instalación de un entorno de desarrollo. El compilador del MSP430 se puede probar con:

```
msp430-gcc –version
msp430-gcc (GCC) 4.7.0 20120322 (mspgcc dev 20120716)
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Y el compilador para el ARM Cortex-M3:

```
arm-gcc-none-eabi-gcc --version
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.9.3 20150529
(release) [ARM/embedded-4_9-branch revision 224288]
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Ahora estamos listos para empezar.

La estructura de Contiki

Contiki tiene la siguiente estructura de archivos:

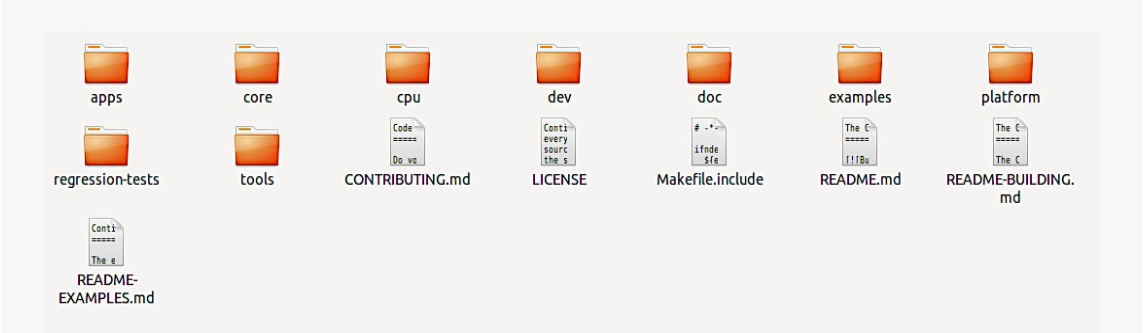

Figura 35. Estructura de archivos de Contiki

- En `examples/zolertia` encuentras ejemplos relacionados específicamente con Zolertia.
- app: aplicaciones de Contiki.
- cpu: archivos específicos al MCU. El archivo cpu/msp430/contiene los *drivers* MCU que usa la mota Z1. La cpu/cc2538 contiene los *drivers* para el CC2538 ARM Cortex-M3 System on Chip (SoC).
- plataforma: archivos y *drivers* de un dispositivo específico. En `tools/z1` están los drivers y configuraciones para la mota Z1. `platform/remote` están los archivos específicos de RE-Mote.
- core: los sistemas de archivos y componentes centrales de Contiki.
- tools: una gama de herramientas de depuración, simulación y desarrollo. En `tools/z1` hay herramientas específicas como el script BSL para programar los dispositivos. Contiki también incluye dos herramientas de simulación: COOJA y MSPSim.
- doc: documentación Doxygen auto-generada.
- Regression tools: pruebas de regresión: un conjunto de pruebas de regresión que chequean aspectos importantes de Contiki diariamente.

Contiki en hardware real

En el resto del libro usaremos como plataformas de desarrollo los dispositivos Zolertia Z1 y RE-Mote. También se pueden usar otras plataformas, pero no las cubriremos aquí.

Como las librerías y los drivers de Contiki son (normalmente) independientes de la plataforma, la mayoría de los ejemplos pueden funcionar indistintamente en las plataformas Z1 y RE-Mote, con los ajustes correspondientes.

Encontramos ejemplos específicos para la plataforma Z1 en `examples/zolertia/z1`, y en `examples/zolertia/remote` para la plataforma RE-Mote.

Tenemos más información sobre ambas plataformas y guías para su actualización en:

http://www.zolertia.

La mota Zolertia Z1

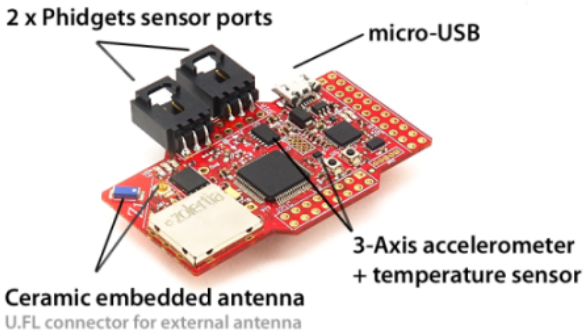

Figura 36. La mota Z1 de Zolertia.

La mota Z1 usa un microcontrolador MSP430F2617 de segunda generación, con CPU RISC de 16-bit @16 MHz, 8 kB de RAM y una memoria Flash de 92 kB. También incluye el conocido radio CC2420, que cumple con IEEE 802.15.4 y opera a 2.4 GHz con una velocidad efectiva de transmisión de datos de 250 kbps.

La mota Zolertia Z1 funciona con TinyOS, Contiki OS, OpenWSN y RIOT, y se ha usado activamente por más de 5 años en universidades, centros de desarrollo e investigación así como en productos comerciales en más de 43 países; igualmente se encuentra mencionada en más de 50 publicaciones científicas.

La mota Z1 está completamente simulada en MSPSIM y Cooja.

Zolertia RE-Mote

RE-Mote es una plataforma de desarrollo que sigue las especificaciones desarrolladas conjuntamente por universidades y socios industriales en el marco del proyecto de investigación europeo **RERUM**: ***REliable, Resilient and secUre IoT for sMart city applications*** (IoT confiable, resistente y segura para aplicaciones de ciudad inteligente).

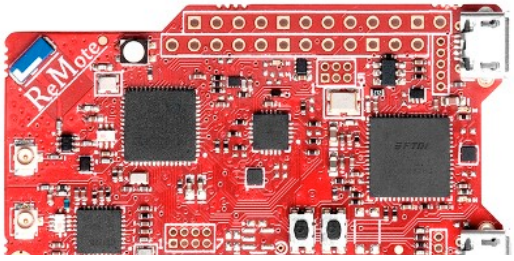

Figura 37. RE-Mote de Zolertia

Importante	Esta es una foto de la RE-Mote prototipo A. La nueva RE-Mote revisión A está todavía en producción y estará pronto disponible. El resto del documento atañe a las especificaciones y características de la RE-Mote revisión A. Para puesta al día y novedades ver http://www.zolertia.io

La RE-Mote de Zolertia tiene como objetivo cerrar la brecha que persiste hasta ahora en las plataformas disponibles relativa a la carencia de la combinación de diseño con robustez industrial y bajo consumo. RE-Mote es fácil de usar y de integrar en dispositivos o en instalaciones ya existentes.

La RE-Mote se basa en el SoC (*System On a Chip*) CC2538 ARM Cortex-M3, con una interfaz RF a 2.4 GHz que respalda el protocolo IEEE 802.15.4. El CC2538 permite operar hasta a 32 MHz, con 512 KB de memoria flash programable y 32 KB de RAM, además de un radio CC1200 868/915 MHz RF, lo que permite operación en ambas bandas de frecuencia. En síntesis, la RE-Mote:

- Cumple con IEEE 802.15.4 y Zigbee , con transceptor a 2. 4 GHz.
- Tiene un segundo radio a 863-950 MHz.
- Motor de encriptación AES-128/256, SHA2 en hardware.
- Motor de aceleración ECC-128/256 y RSA en Hardware para intercambio de claves seguro.
- Consumo de apenas 300 nA en el modo "s*hutdown*".
- Programación en BSL con facilidades para el "*bootloader mode*".
- Cargador de batería incorporado (500 mA), con facilidad para conectar paneles solares y baterías LiPo.
- Amplia gama de voltaje de alimentación, de 2 a 16 V.
- Factor de forma pequeño
- MicroSD sobre SPI.

- Reloj de tiempo real programable RTC (*Real Time Clock*) y WDT (*Watchdog Timer*) externo.
- Interruptor RF programable para conectar la antena externa alternativamente a la interfaz de 2.4 GHz o Sub-1 GHz a través del conector RP-SMA.

¡Comienza con Contiki!

¡Vamos a construir nuestro primer ejemplo en Contiki! Abre un terminal y escribe:

```
cd examples/hello-world make
TARGET=remote savetarget
```

Esto le ordenará a Contiki procesar el ejemplo *hello world* para la plataforma RE-Mote. Si en cambio usamos la mota Z1, escribimos:

```
make TARGET=z1 savetarget
```

Esto es necesario hacerlo sólo una vez por cada aplicación. Ahora compilemos la aplicación (ignora las advertencias):

```
make hello-world
```

si todo va bien deberíamos ver algo así:

```
CC        symbols.c
AR        contiki-z1.a
CC        hello-world.c CC        ../../platform/z1/./contiki-z1-main.c
LD        hello-world.z1
rm obj_z1/contiki-z1-main.o hello-world.co
```

El archivo `hello-world.z1` debería haberse creado y podemos programarle la aplicación al dispositivo.

En cualquier punto se puede anular el objetivo guardado y redefinirlo en el momento de la compilación escribiendo:

```
make TARGET=remote hello-world
  AR        contiki-remote.a
  CC        ../../cpu/cc2538/cc2538.lds
  CC        ../../cpu/cc2538/./startup-gcc.c
  CC        hello-world.c
  LD        hello-world.elf
arm-none-eabi-objcopy -O ihex hello-world.elf hello-world.hex
arm-none-eabi-objcopy -O binary --gap-fill 0xff hello-world.elf hello-
world.bin
cp hello-world.elf hello-world.remote
rm http://hello-world.co/[hello-world.co] obj_remote/startup-gcc.o
```

Esto hará que se ignore el archivo guardado `Makefile.target` y que se use en su lugar `remote` como objetivo.

Los ejemplos de las secciones siguientes pueden compilarse para cualquiera de las dos plataformas Z1 y RE-Mote, a menos que se especifique explícitamente.

Hello world explicado

Veamos los componentes del ejemplo *Hello world*.

Explora el código con

```
gedit hello-world.c
```

O con tu editor de texto favorito

Cuando inicias Contiki, declaras los procesos con un nombre. En cada código tienes varios procesos que enuncias de la siguiente manera:

```
PROCESS(hello_world_process, "Hello world process"); (1)
AUTOSTART_PROCESSES(&hello_world_process);           (2)
```

1. `hello_world_process` es el nombre del proceso y `"Hello world process"` es el nombre legible del proceso cuando lo imprimes en el terminal.

2. `AUTOSTART_PROCESSES(&hello_world_process)`le ordena a Contiki que comience el proceso cuando termine el lanzamiento (booting).

```
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data) (1)
{
        PROCESS_BEGIN(); (2)
        printf("Hello, world\n"); (3)
        PROCESS_END(); (4)
}
```

3. Declaras el contenido del proceso en el hilo de procesos, el nombre del proceso y las funciones de retorno (*callback*): manejador de eventos y manejador de datos.

4. El proceso arranca dentro del hilo.

5. Haz lo que quieras.

6. Y finalmente termina el proceso.

Makefile explicado

Las aplicaciones necesitan un Makefile para compilarse. Veamos el Makefile de `hello-world`:

```
CONTIKI_PROJECT = hello-world        (1)
all: $(CONTIKI_PROJECT)              (2)
CONTIKI = ../..                      (3)
include $(CONTIKI)/Makefile.include  (4)
```

1) Le dice al sistema de construcción cuál es la aplicación que se va a compilar
2) Si se usa `make all` compilará las aplicaciones definidas
3) Especifica nuestro nivel de sangría (*indentation*) respecto a la carpeta raíz de Contiki.
4) El archivo Contiki Makefile global, también apunta a la Makefile de la plataforma

Podemos definir banderas (*flags*) específicas de compilación en Makefile, sin embargo, la forma recomendada sería añadir una cabecera `project-conf.h`
y definir allí cualquier valor o bandera de compilación. Esto se hace añadiendo lo siguiente a Makefile:

```
DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
```

Y luego creando una cabecera `project-conf.h` para la carpeta en la posición del ejemplo.

Recomendable

Antes de embarcarse en Contiki es útil recordar que los drivers normalmente tienen un switch como este:

```
#define DEBUG 0
#if DEBUG #define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...) #endif
```

Habilita el `DEBUG` cambiando a `1` o `DEBUG_PRINT`. Esto va a imprimir información de depuración en la consola. Normalmente deberías hacer esto en cada archivo del driver que quieres examinar.

Para añadir un LED al ejemplo

El próximo paso es añadir un LED (*Light Emiting Diode*) para interactuar con nuestra aplicación.

Tienes que añadir `dev/leds.h` que es la librería que maneja los LED. Para ver las funciones disponibles ve a `/home/user/contiki/core/dev/leds.h`

Comandos disponibles para interactuar con los LED:

```
unsigned char leds_get(void);
void leds_set(unsigned char leds);
void leds_on(unsigned char leds);
void leds_off(unsigned char leds);
void leds_toggle(unsigned char leds);
```

Normalmente todas las plataformas contienen la siguiente lista de LEDS:

```
LEDS_GREEN
LEDS_RED
LEDS_BLUE
LEDS_ALL
```

En la mota Z1 estos LED y otras definiciones de hardware están descritos en `platform/z1/platform-conf.h`.

La RE-Mote usa un LED RGB (*Red, Green, Blue*), que es básicamente LED de tres colores en un dispositivo único, lo que permite producir cualquier color resultante de la combinación de los primarios. En la cabecera `platforms/remote/dev/board.h` están predefinidos los siguientes:

```
LEDS_LIGHT_BLUE
LEDS_YELLOW
LEDS_PURPLE
LEDS_WHITE
```

Trata de encender el LED rojo para ver qué pasa

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  leds_on(LEDS_RED);
  PROCESS_END();
}
```

A continuación vamos a compilar y a cargar el nuevo proyecto con:

```
make clean && make hello-world.upload
```

El comando `make clean` se usa para borrar objetos previamente compilados.

¡Ahora, el LED rojo debería estar encendido!

Advertencia	Si haces cambios al código fuente, debes reconstruir los archivos, de otra manera, los cambios realizados podrían quedar fuera. Es siempre recomendable dar el comando `make clean` antes de compilar.
Ejercicio	Trata de encender los otros LED

Para visualizar mensajes en la consola

Puedes usar **printf** para visualizar en pantalla lo que está pasando en tu aplicación. Es realmente útil para depurar tu código, puesto que imprime los valores de las variables, cuando se ejecuta un bloque de código, etc.

Tratemos de imprimir en pantalla el estatus del LED usando `unsigned char leds_get(void);` Esta función está disponible en las funciones documentadas (ver arriba).

Obtén el estatus del LED e imprímelo en la pantalla.

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
char hello[] = "hello from the mote!";
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
      PROCESS_BEGIN();
      leds_on(LEDS_RED);
      printf("%s\n", hello);
      printf("The LED %u is %u\n", LEDS_RED, leds_get());
      PROCESS_END();
}
```

Si un LED está encendido, obtendrás el número del LED correspondiente el cual se encuentra definido para cada plataforma en su cabecera `platform-conf.h` o en la `board.h`

Ejercicio	¿Qué pasa cuando enciendes más de un LED? Qué número se obtiene? ¿Son los mismos números que para las motas Z1 y RE-Mote?

Para añadir eventos con botones (*button events*)

Ahora queremos detectar si **el botón de usuario** ha sido pulsado.

Contiki considera el botón como un tipo de sensor. Vamos a usar la librería `core/dev/button-sensor.h`

Dato	La plataforma RE-Mote tiene funciones extra para el botón, por ejemplo, la detección de secuencias largas de pulsado, lo que permite expandir aún más los eventos que pueden ser disparados usando el botón. Para detalles adicionales examina `platform/remote/dev/button-sensor.c` Y para un ejemplo específico: `examples/zolertia/remote/remote-demo.c`

Añade a nuestro ejemplo la cabecera `dev/button-sensor.h` y añade un evento con botón (*button event):*

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
   PROCESS_BEGIN();
   SENSORS_ACTIVATE(button_sensor);
   while(1) {
   PROCESS_WAIT_EVENT_UNTIL((ev==sensors_event) &&
   (data == &button_sensor));
   printf("I pushed the button! \n");
   }
   PROCESS_END();
}
```

Este proceso tiene un bucle infinito dado por el wait(), para esperar por el pulsado del botón. Las dos condiciones deben cumplirse (evento desde un sensor, y que ese evento sea el pulsado del botón).
Al pulsar el botón se imprime la secuencia en pantalla.

Ejercicio	Enciende el LED cuando el botón esté pulsado. Apaga el LED cuando el botón se pulse de nuevo.

Temporizadores

Usando temporizadores podremos arrancar un evento a una hora determinada, aumentando la velocidad de la transición entre los diferentes estados del proceso y haciendo que una tarea o un proceso se ejecuten automáticamente, por ejemplo, destellar un LED cada 5 segundos sin que el usuario tenga que pulsar el botón cada vez.

El OS (sistema operativo) de Contiki tiene cuatro tipos de temporizadores:

- Temporizador simple: es un simple reloj, la aplicación debe chequear manualmente si el temporizador se ha detenido. Más información: `core/sys/timer.h`
- Temporizador de retro-llamada (*callback timer*): cuando un temporizador expira puede re-invocar una función determinada. Más información: `core/sys/ctimer.h`
- Temporizador de evento: lo mismo que el anterior, pero cuando el temporizador expira, en lugar de invocar una función , escribe un mensaje informando de su expiración . Más información: `core/sys/etimer.h`
- Temporizador de tiempo real: El módulo de tiempo real maneja la programación y ejecución de tareas en tiempo real; hay un solo temporizador de este tipo disponible por el momento. Más información: `core/sys/rtimer.h`

Para nuestra implementación vamos a escoger el temporizador de evento, porque queremos cambiar el comportamiento de la aplicación cuando el temporizador pare cada cierto tiempo.

Creamos una estructura de temporizador y fijamos éste para expirar cada cierta cantidad de segundos. Cuando el temporizador se para, ejecutamos el código y lo reiniciamos.

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h> #define  SECONDS 2
/*---------------------------------------------------*/
PROCESS(hello_world_process, "hello world example");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
PROCESS_BEGIN();
     static struct etimer et;
      while(1) {
        etimer_set(&et, CLOCK_SECOND*SECONDS); (1)
        PROCESS_WAIT_EVENT(); (2)
        if(etimer_expired(&et)) {
        printf("Hello world!\n");
        etimer_reset(&et);
   }
    }
PROCESS_END();
}
```

1) `CLOCK_SECOND` es un valor relacionado con la cantidad de *ticks* por segundo del microcontrolador. Como Contiki es compatible con diferentes plataformas de hardware, el valor de `CLOCK_SECOND` también difiere.

2) `PROCESS_WAIT_EVENT()` espera que ocurra **cualquier** evento.

Ejercicio	¿Puedes visualizar el valor de `CLOCK_SECOND` para contar cuántos *ticks* tienes en un segundo? Trata de hacer destellar el LED durante un número determinado de segundos. Escribe otra aplicación que arranca sólo cuando se pulsa el botón y se detiene cuando se pulsa de nuevo.

Sensores

Un sensor es un transductor cuyo propósito es percibir o detectar una característica de su ambiente proporcionando una salida, generalmente como una señal eléctrica u óptica relacionada con el valor de la variable medida.

Los sensores en Contiki se implementan de esta manera:

```
SENSORS_SENSOR (sensor, SENSOR_NAME, value, configure, status);
```

Esto significa que se debe implementar un método `configure` para configurar el sensor, sondear su `status`, y solicitar una lectura con el método `.`

La estructura del `sensor` contiene apuntadores para estas funciones. Los argumentos para cada función se muestran a continuación:

```
struct sensors_sensor {
  char *       type;
  int          (* value)     (int type);
  int          (* configure) (int type, int value);
  int          (* status)    (int type);
};
```

Para una mejor comprensión consulte `platform/remote/dev/adc-sensors.c` donde encontrará un ejemplo de cómo pueden implementarse los sensores analógicos externos (que serán discutidos en más detalle en las secciones próximas).

Las siguientes funciones y macros pueden usarse para trabajar con el sensor:

```
SENSORS_ACTIVATE(sensor)    (1)
SENSORS_DEACTIVATE(sensor)  (2)
sensor.value(type);         (3)
```

1. Habilita al sensor. Normalmente configura y enciende el sensor.
2. Deshabilita el sensor. Es útil para ahorrar energía.
3. Le solicita un valor al sensor. Como el chip de un sensor puede hacer diferentes tipos de lecturas, (temperatura, humedad, etc.), esta función se usa para especificar cuál medida se debe tomar.

Importante	Esta es la manera predeterminada para implementar sensores en Contiki, pero dependiendo del sensor y del desarrollador pude hacerse también de otras maneras. Chequea siempre las implementaciones específicas del driver del sensor para ver los detalles. Normalmente los sensores están puestos en la misma ubicación del ejemplo, en la carpeta `dev`, o en el directorio `platform/remote/dev`, por ejemplo.

La macro `SENSORS` proporciona una conexión externa a los sensores definida para la plataforma y la aplicación lo que permite la reutilización para otros sensores. A continuación vemos los sensores predeterminados para la plataforma RE-Mote.

```
SENSORS (&button_sensor, &vdd3_sensor, &cc2538_temp_sensor, &adc_sensors);
```

Nota	El botón es considerado un sensor en Contiki porque comparte la misma implementación que los sensores.

Esta macro se extiende a:

```
const struct sensors_sensor *sensors[] = {
  &button_sensor,
  &vdd3_sensor,
  &cc2538_temp_sensor,
  &adc_sensors,
  ((void *)0)
};
```

El número de sensores se encuentra con la macro SENSORS_NUM :

```
#define SENSORS_NUM (sizeof(sensors) / sizeof(struct sensors_sensor *))
```

Los sensores pueden ser de dos tipos, analógicos y digitales. En las siguientes secciones mostraremos ejemplos de ambos tipos detallando cómo conectar y usar tanto los sensores internos como los externos de la plataforma.

Sensores analógicos

Los sensores analógicos normalmente necesitan ser conectados a un convertidor ADC (de analógico a digital) para traducir la lectura analógica (continua) a valores binarios que pueden ser manejados en el mundo digital. El primer paso es muestrear la señal analógica par captar sus valores en los instantes de muestreo. La frecuencia de muestreo debe ser como mínimo el doble de la frecuencia máxima del fenómeno que se está midiendo. Por ejemplo, para muestrear la voz, típicamente filtrada a 4 kHz, se debe muestrear por lo menos a 8 kHz. Las lecturas muestreadas son todavía continuas, y pueden abarcar una gama infinita de valores. Para convertirlas a digital, donde sólo puede existir un número finito de valores, es necesario un proceso de redondeo similar al que se realiza cuando se desprecian cierto número de decimales, por ejemplo cuando expresamos PI como 3.14. Mientras mayor sea la cantidad de bits que usamos para cada muestra, menor será el error incurrido. Esto define la resolución del ADC. Tanto la ZI como la Re-mote ofrecen hasta 12 bits por muestra.

La mota Z1 tiene integrado un sensor de voltaje con un ADC y también tiene otro genérico para leer sensores analógicos externos.

Dato

Los sensores analógicos pueden conectarse tanto en Z1 como en RE-Mote a través de los puertos *phidget,* que son básicamente conectores de 3 pines (Tierra, VCC y Señal) con un espaciamiento de 2.54 mm. El nombre *phidget* es un legado del lanzamiento de la Z1 ya que estos puertos eran originalmente para los sensores comerciales Phidget. Hoy en día hay también sensores de otros proveedores, por ejemplo *seeedstudio* con los mismos pines pero espaciados a 2 mm. Esto no es un problema ya que existen los cables para realizar la adaptación.

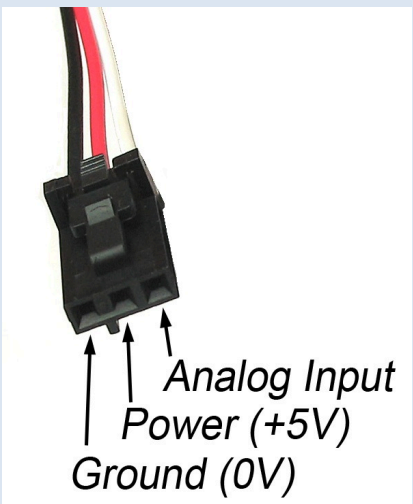

Figura 38. Conector Phidget

Cuando se requiere una lectura del voltaje en milivoltios necesitamos convertir la salida del ADC mediante la siguiente fórmula: Se multiplica el valor medido por el voltaje de referencia y se divide el producto entre el valor máximo de la salida del ADC. Como la resolución del ADC es de 12 bits (2^{12}= 4096), la fórmula queda:

```
Voltaje,mv = (unidades * Vref) / 4096;
```

El voltaje de referencia limita el rango de nuestra medida, es decir que si usamos un `Vref` de 2.5 V, el sensor se saturará cuando al leer valores de un voltaje superior. La referencia puede escogerse cuando se configura el ADC. En los casos de Z1 y RE-Mote se usa normalmente 3.3 V.

Z1 y RE-Mote permiten la conexión de hasta 6 sensores analógicos, pero sólo se pueden soldar dos conectores *phidget* al mismo tiempo.

Para la RE-Mote es posible tener un solo conector phidget para un sensor analógico (ADC1) de 3.3 V, y otro para sensores de 5 V (ADC3).

Dato	La RE-Mote se basa en ARM Cortex-M3 lo que permite mapear cualquier pin a un controlador (lo que no hace el MSP430, para el cual un pin tiene funciones predefinidas además de las correspondientes a GPIO), pero sólo los pines del puerto PA pueden usarse como ADC.

Para la mota Z1 es posible tener estas combinaciones (ver la figura siguiente):

- Dos conectores phidget para sensores de 3.3V.
- Dos conectores phidget para sensores de 5 V.
- Dos conectores phidget para sensores de 3,.3 V y de 5V.

JP1A

Features	MSP430 Port#	Pin Name	Pin#		Pin Name	MSP430 Port#	Features
Phidget powered @5V, ADC input has resistor divider to allow 5V inputs		ADCGND	1	2	ADCGND		*Phidget @3V*
		USB+5V	3	4	Vcc+3V		
	P6.3	ADC3*	5	6	ADC7	P6.7	
	P6.4	ADC4	7	8	ADC6/DAC0	P6.6	
	P6.2	ADC2	9	10	ADC5/DAC1	P6.5	
Phidget powered @5V, ADC input has resistor divider to allow 5V inputs		ADCGND	11	12	ADCGND		*Phidget @3V*
		USB+5V	13	14	Vcc+3V		
	P6.0	ADC0*	15	16	ADC1	P6.1	

Table 3 — JP1A Pinout description

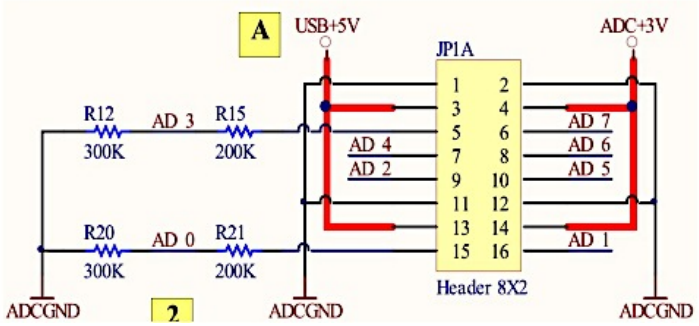

Figura 39. Asignación de pines

Como se muestra en el código siguiente, los canales ADC se pueden usar para conectar sensores analógicos externos.

```
/* MemReg6 == P6.0/A0 == 5V 1 */
ADC12MCTL0 = (INCH_0 + SREF_0);
/* MemReg7 == P6.3/A3 == 5V 2 */
ADC12MCTL1 = (INCH_3 + SREF_0);
/* MemReg8 == P6.1/A1 == 3V 1 */
ADC12MCTL2 = (INCH_1 + SREF_0);
/* MemReg9 == P6.7/A7 == 3V_2 */
ADC12MCTL3 = (INCH_7 + SREF_0);
```

Para tomar una medida de un sensor conectado al canal ADC7 de la mota Z1, necesitas seguir los siguientes pasos:

```
#include "dev/z1-phidgets.h"                                    (1)
SENSORS_ACTIVATE(phidgets);                                     (2)
printf("Phidget 5V 1:%d\n", phidgets.value(PHIDGET3V_2));       (3)
```

1) Incluye la cabecera del driver
2) Activa los sensores ADC. Por defecto se activan los 4 canales ADC.
3) Solicita una lectura

La Z1 está alimentada a 3.3 V, pero cuando se conecta al puerto USB (a 5 V) permite conectar sensores 5 V a los puertos phidget, es decir, 5V1 (ADC·) y 5V2 (ADC0) ya que hay un divisor de voltaje en la entrada para adaptar la lectura de 5 V a 3.3 V.

Hay detalles sobre la implementación del driver de los sensores analógicos de Z1 en `platform/z1/dev/z1-phidgets.c`. También hay un driver para el sensor del voltaje interno del MSP430 en `platform/z1/dev/battery-sensor.c`, con un ejemplo en `examples/zolertia/z1/test-battery.c`.

Veamos la implementación con un ejemplo, el sensor de Luz de precisión Phidget 1142.

Figura 40. Sensor de luz Phidget 1142. Imagen tomada de phidgets.com

Hay un ejemplo llamado `test-phidgets.c` en `examples/zolertia/z1`. Este lee los valores de un sensor analógico y los imprime en pantalla.

Conecta el sensor de luz al conector Phidget 3V2 y compila el ejemplo:

```
make clean && make test-phidgets.upload && make z1-reset && make login
```

Dato

Puedes encadenar varios comandos para que se ejecuten uno después de otro. En este caso `make z1-reset` reinicia la mota después de lanzada, luego `make login` imprime la salida de cualquier comando `printf` de nuestro código. Podrías necesitar el argumento `MOTES=/dev/ttyUSB[0…9]` para especificar una mota en un puerto USB específico.

Este es el resultado:

```
Starting 'Test Button & Phidgets'
Please press the User Button
Phidget 5V 1:123
Phidget 5V 2:301
Phidget 3V 1:1710
Phidget 3V 2:2202
```

El sensor de luz se conecta al conector 3V2, así que el valor bruto es 2202. Trata de iluminar el sensor con una linterna (la del móvil, por ejemplo) y luego cúbrelo con la mano de manera que no le llegue luz.

Del sitio web de Phidget tenemos la siguiente información sobre el sensor:

Parámetro	**Valor**
Tipo de sensor	Luz
Nivel mínimo de luz	1 lux
Suministro mínimo de voltaje	2.4 V
Suministro máximo de voltaje	5.5 V
Consumo máximo de corriente	5 mA
Nivel de luz máximo (3.3 V)	660 lux
Nivel de luz máximo (5 V)	1000 lux

Como vemos, el sensor de luz puede conectarse al Phidget de 5 V o al de 3.3 V. El valor máximo medido cambia en correspondencia.

La fórmula para convertir SensorValue en luminosidad es: Luminosity(lux) = SensorValue

La **plataforma RE-Mote** también permite conectar sensores de 5 V al puerto ADC3 usando el mismo divisor de voltaje que Z1. Como se ve en el siguiente fragmento, el puerto ADC3 está mapeado en el pin PA2.

```
/*
* This driver supports analogue sensors connected to ADC1, ADC2 and AND3
inputs
* This is controlled by the type argument of the value() function. Possible  *
choices are:
* - REMOTE_SENSORS_ADC1 (channel 5)
* - REMOTE_SENSORS_ADC2 (channel 4)
* - REMOTE_SENSORS_ADC3 (channel 2)
*/
```

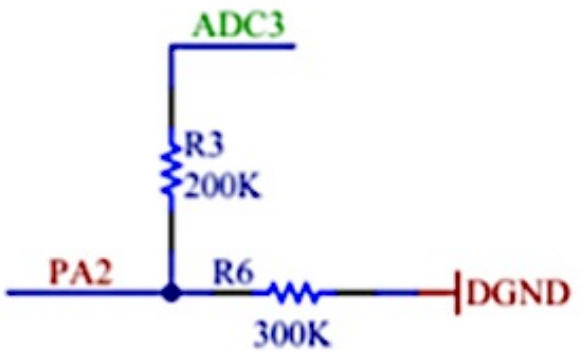

Figura 41. Divisor de tensión

Entonces, para leer los datos desde un sensor conectado (como hicimos en el ejemplo anterior):

```
#include "dev/remote-sensors.h"                                          (1)
adc_sensors.configure(SENSORS_HW_INIT, REMOTE_SENSORS_ADC_ALL);          (2)
printf("Phidget ADC2 = %d raw\n", adc_sensors.value(REMOTE_SENSORS_ADC2));(3)
printf("Phidget ADC3 = %d raw\n", adc_sensors.value(REMOTE_SENSORS_ADC3));
```

1) Incluye la cabecera del driver ADC
2) Activa y configura los canales ADC (puede seleccionar cuáles activar)
3) Solicita una lectura

Para más detalles examina el ejemplo RE-Mote en `example/remote/remote-demo.c`

Ejercicio	Haz que el sensor realice las lecturas lo más rápido posible. Lleva a la pantalla los valores brutos de ADC y los milivoltios (como este sensor es lineal, el voltaje corresponde a las luxes). ¿Cuáles son los valores máximos y mínimos que puedes obtener? ¿Cuál es el valor promedio de luz en la habitación? Crea una aplicación que encienda el LED rojo cuando esté oscuro y que encienda el LED verde cuando haya luz. En el intermedio, apaga todos los LED. Añade un temporizador para medir la luz cada 10 segundos.

Sensores digitales

Los sensores digitales normalmente se interrogan con un protocolo de comunicación digital como por ejemplo I2C, SPI, 1-Wire, Serial o, dependiendo del fabricante, un protocolo privado controlado por un reloj.

Los sensores digitales aceptan un conjunto más amplio de órdenes (prender, apagar, configurar interrupciones). Con un sensor digital de luz por ejemplo, puedes fijar un valor umbral y hacer que el sensor envíe una interrupción cuando se alcance ese valor, sin la necesidad de leer continuamente el sensor.

Recuerda chequear la información especifica del sensor y la hoja de datos para más información.

La **mota Z1** tiene **dos sensores digitales incorporados**: de temperatura y acelerómetro de 3 ejes. Vamos a empezar con este último. Tenemos el ejemplo `test-adxl345.c` para prueba.

El ADLX345 es un sensor I2C de ultra bajo consumo, capaz de leer hasta 16 G y muy apropiado para aplicaciones de dispositivos móviles. Mide la aceleración estática de la gravedad en aplicaciones que detectan inclinación, así como la aceleración dinámica resultante de movimiento o choque. Su alta resolución (3.9 mg/LSB) le permite medir cambios de inclinación menores que 1.0^0.

```
[37] DoubleTap detected! (0xE3) -- DoubleTap Tap
x: -1 y: 12 z: 223
[38] Tap detected! (0xC3) -- Tap
x: -2 y: 8 z: 220
x: 2 y: 4 z: 221
x: 3 y: 5 z: 221
x: 4 y: 5 z: 222
```

El acelerómetro mide datos en los ejes x, y y z. y responde a tres tipos de interrupciones: un toque simple, un toque doble y una caída libre (*free-fall)* (cuidado con dañar la mota!).

El código tiene dos hilos, uno para las interrupciones y el otro para los LED. Cuando Contiki arranca dispara ambos procesos.

El hilo `led_process` dispara un temporizador que espera antes de apagar los LED. Esto lo hace principalmente para filtrar la señal rápida que viene del acelerómetro. El otro proceso es el de aceleración. Asigna una función a invocar al evento `led_off`. Las interrupciones pueden ocurrir en cualquier momento, son totalmente asincrónicas.

Las interrupciones pueden dispararse por fuentes externas (sensores, GPIO, *Watchdog Timer*, etc.) y deben atenderse lo más pronto posible. Cuando ocurre una interrupción, el controlador del interruptor (que es un proceso que chequea los registros de interrupción para averiguar cuál es el origen de la misma) se encarga del problema e invoca a la función asignada.

En este ejemplo, el acelerómetro se inicializa y luego las interrupciones se mapean a las funciones asignadas. La fuente de interrupción 1 se mapea a la función que maneja los eventos de f*ree fall* y los eventos de toques o *clicks* se mapean en la fuente de interrupción 2.

```
/*
* Start and setup the accelerometer with default
* values, _i.e_ no interrupts enabled.
*/ accm_init();
/* Register the callback functions */
ACCM_REGISTER_INT1_CB(accm_ff_cb);
ACCM_REGISTER_INT2_CB(accm_tap_cb);
```

Necesitamos activar las interrupciones:

```
accm_set_irq(ADXL345_INT_FREEFALL,
       ADXL345_INT_TAP +
       ADXL345_INT_DOUBLETAP);
```

En el bucle *while* leemos los valores desde cada eje cada segundo. Si no hay interrupciones, esto será lo único que se muestra en la terminal.

Ejercicio	Pon la mota en diferentes posiciones y chequea los valores del acelerómetro. Trata de entender lo que es X, Y y Z. Agita la mota y mide la aceleración máxima. Prende y apaga el LED de acuerdo a la aceleración en un eje.

En el siguiente ejemplo usaremos el **ZIG-SHT25,** un sensor digital I2C de temperatura y humedad basado en el sensor SHT25 de Sensirion.

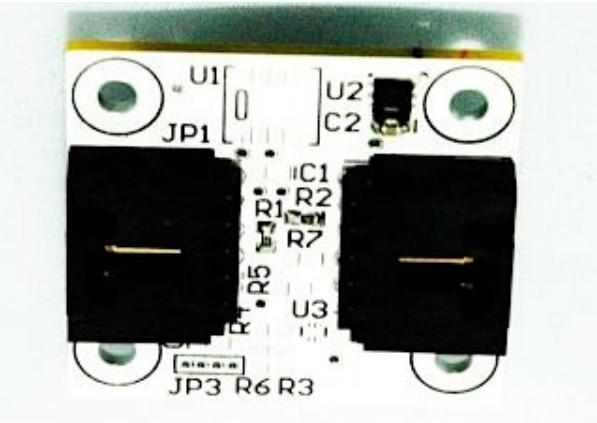

Figura 42. Sensor Sensirion

Parámetro	**Valor**
Tipo de sensor	Temperatura y humedad
Voltaje de alimentación [V]	2.1 – 3.6
Consumo de energía	3.2 uW (a 8 bit, 1 medición / s)
Tasa de datos	0-100 % RH (humedad) 40-125ºC (temperatura)
Resolución máxima	14 bits (temperatura), 12 bits (humedad)
Consumo máximo de corriente	300 uA

El ejemplo del sensor ZIG-SHT25 está en `examples/zolertia/remote/test-sht25.c` y en `examples/zolertia/z1/test-sht25.c`. Una salida típica sería:

```
Starting 'SHT25 test'
Temperature 23.71 ºC
Humidity 42.95 % RH
Temperature 23.71 ºC
Humidity 42.95 % RH
Temperature 23.71 ºC
Humidity 42.95 % RH
Temperature 23.71 ºC
Humidity 42.98 % RH
```

Como siempre, puedes chequear la implementación del driver en los directorios `platform/z1/dev` y `platform/remote/dev`. Examina los otros sensores disponibles y otros ejemplos que pueden ayudarte a implementar los tuyos.

Ejercicio	Convierte la temperatura a Fahrenheit. Trata de poner la temperatura y la humedad lo más alto posible (¡sin dañar la mota!). Trata de obtener sólo los valores "posibles" (si desconectas el sensor no se debería visualizar nada, o ver un mensaje de error).

Emular Contiki con Cooja

Cooja es el emulador de Contiki. Cooja permite emular pequeñas y grandes redes de motas en Contiki. La emulación es más lenta que la ejecución en hardware pero permite una inspección precisa del comportamiento del sistema. También permite enfocar a un nivel menos detallado, lo que es más rápido y permite la simulación de redes más grandes.

Cooja es una herramienta muy útil para el desarrollo de Contiki ya que le permite a los desarrolladores probar su código y su sistema mucho antes de ejecutarlo en el hardware definitivo. Los desarrolladores crean continuamente nuevas simulaciones para depurar el software y para verificar el comportamiento de sus sistemas.

La mayoría de los ejemplos que se dan en esta sección pueden emularse en Cooja también (excepto los de sensores)

Para empezar Cooja, ve al directorio:

```
cd contiki/tools/cooja
```

Inicia Cooja con el comando siguiente:

```
ant run
```

Cuando Cooja finaliza la compilación, comienza con una ventana azul vacía. Ahora que ya lo tenemos cargado y funcionando, podemos ensayar con un ejemplo de simulación.

Vamos a `File, Open Simulation, Browse,` luego navega a `examples/hello-world` y selecciona el ejemplo `hello-world-example.csc`. Esto va a cargarnos el ejemplo hello world guardado anteriormente.

Si quieres editar un ejemplo (en caso de usar un tipo diferente de plataforma), en lugar de seleccionar la opción `Browse` selecciona `Open` y `Reconfigure`. Esto te conducirá por los pasos que debes seguir para configurar tu ejemplo.

Crea una nueva simulación

Haz clic en el menú `File` y luego clic en `New simulation`. Cooja ahora abre el diálogo `Create new simulation`. Aquí podemos escoger un nuevo nombre para nuestra simulación, pero en nuestro ejemplo dejaremos `My simulation`. Deja las otras opciones como opciones por defecto. Haz clic en el botón `Create.`

Cooja muestra la nueva simulación. Puedes escoger lo que quieres visualizar usando el menú Tools. La ventana Network muestra todas las motas en la red simulada que en este momento debería estar vacía porque no tenemos motas en nuestra simulación. La ventana Timeline muestra todos los eventos de comunicación de nuestra simulación en el tiempo —muy útil para entender lo que sucede en la red. La ventana Mote output muestra las lecturas de todos los puertos seriales de todas las motas. La ventana Notes es donde podemos poner notas para nuestra simulación. Y la ventana Simulation control es donde iniciamos, pausamos, o recargamos nuestra simulación.

Añade motas a la simulación

Antes de que podamos simular nuestra red, tenemos que añadir una o más motas. Esto lo hacemos usando el menú Motes en el que hacemos clic en Add motes. Como esta es la primera mota que añadimos, debemos primero crear un tipo de mota para añadir. Haz clic en Create new mote type y selecciona uno de los tipos de mota disponibles. Para este ejemplo, seleccionamos Z1 mote para crear una mota Z1 simulada. Cooja abre el diálogo Create Mote Type en el cual podemos escoger un nombre para nuestro tipo de mota, así como la aplicación Contiki que nuestro tipo de mota ejecutará.

Para este ejemplo, nos quedamos con el nombre ya sugerido, y hacemos clic en el botón Browse a la derecha para escoger nuestra aplicación Contiki.

Inalámbrico con Contiki

En la sección anterior hemos cubierto algunas de las características principales de Contiki, fundamentos sobre los sensores y una visión general de cómo se construyen, se programan y se simulan en Contiki las aplicaciones. En esta sección se presenta la comunicación inalámbrica, los detalles sobre los radios e ideas básicas acerca de la configuración de nuestras plataformas.

Preparación del dispositivo

El primer paso es la comprensión de cómo está configurada nuestra plataforma. Cada plataforma implementa su propio conjunto de valores predeterminados y configuraciones para ser utilizados por los módulos subyacentes como los radios, puertos seriales, etc.

Para la plataforma Zolertia los sitios a visitar son:

- Las configuraciones específicas del hardware: parámetros como pines I2C predeterminados, los canales ADC, asignación de pines para un módulo específico e información sobre la plataforma se encuentran en `platform/remote/dev/board.h` y en `platform/z1/platform-conf.h`.

- Las configuraciones específicas de Contiki: Configuraciones de UART, driver MAC, canal de radio, IPv6, RIME y de la memoria *buffer* de la red se encuentran en `platform/remote/contiki-conf.h` y en `platform/z1/contiki-conf.h`.

Como buena práctica general las aplicaciones pueden sobreescribir los parámetros configurables por parte del usuario; esto sirve de guía para distinguir cuáles son los valores que pueden ser cambiados por un usuario corriente de aquellos valores que se van a cambiar solamente si el usuario sabe muy bien lo que está haciendo.

```
#ifndef UART0_CONF_BAUD_RATE
#define UART0_CONF_BAUD_RATE 115200
#endif
```

Al definir UART0_CONF_BAUD_RATE en nuestra aplicación por medio de project-conf.h, podemos cambiar el valor predeterminado de 115200 baudios para la tasa de transmisión. Nótese que en general es una buena práctica añadir CONF a los parámetros configurables por el usuario.

Dato	Una de las herramientas más usadas es probablemente grep, un comando muy cómodo para buscar una secuencia de texto específica en cualquier documento o ubicación. Una forma de ejecutar este comando es grep -lr "alinan. Si se ejecuta en la raíz de la instalación de Contiki, proporciona una lista recursiva de todos los archivos creados por Antonio Linan. Esta es una buena forma de precisar la ubicación de una definición específica si no se está usando un IDE (Integrated Development Environment) como Eclipse. Ver 'man grep' para más información. Para Windows Astrogrep es una buena opción. [http://astrogrep.sourceforge.net/]

En la próxima sección revisaremos los parámetros de configuración más importantes. Sin embargo, como siempre, dependiendo de la aplicación y configuración, la mejor forma de asegurarse de que

todo funcione adecuadamente es revisando los archivos de configuración de la plataforma usada y modificarlos o redefinirlos en según nuestras necesidades.

Direcciones de dispositivos

Para iniciar a trabajar debes comenzar por definir la ID para cada nodo. Esto se usará para generar la dirección MAC de la mota y las direcciones IPv6 (de enlace local y global).

Direcciones de la RE-Mote

La plataforma **RE-Mote** viene con dos direcciones MAC pre-cargadas, almacenadas en la memoria flash interna, pero el usuario puede elegir otra (*hardcoded*). Los *switches* siguientes en platform/remote/contiki-conf.h seleccionan la elegida.

```
/**
 * \name IEEE address configuration
 *
 * Used to generate our RIME & IPv6 address
 * @{
       */
/**
 * \brief Location of the IEEE address
 * 0 => Read from InfoPage,
 * 1 => Use a hardcoded address, configured by IEEE_ADDR_CONF_ADDRESS
 */
#ifndef IEEE_ADDR_CONF_HARDCODED
#define IEEE_ADDR_CONF_HARDCODED            0
#endif

/**
 * \brief Location of the IEEE address in the InfoPage when
 * IEEE_ADDR_CONF_HARDCODED is defined as 0
 * 0 => Use the primary address location
 * 1 => Use the secondary address location
 */
#ifndef IEEE_ADDR_CONF_USE_SECONDARY_LOCATION
#define IEEE_ADDR_CONF_USE_SECONDARY_LOCATION 0
#endif
```

Si usas tu propia dirección *hardcoded* el siguiente *define* puede ser sobreescrito por la aplicación:

```
#ifndef IEEE_ADDR_CONF_ADDRESS
#define IEEE_ADDR_CONF_ADDRESS { 0x00, 0x12, 0x4B, 0x00, 0x89, 0xAB,
0xCD, 0xEF }
#endif
```

Direcciones de la mota Z1

Vamos a escoger la ID de la lista de motas conectadas:

```
Reference  Device       Description
-----------------------------------
Z1RC3301   /dev/ttyUSB0 Silicon Labs Zolertia Z1
```

La ID del nodo debería ser 3301 (decimal) si no se ha encontrado otra ID previamente almacenada en la memoria flash.

Veamos cómo Contiki usa esto para derivar las direcciones IpV6 y MAC completas. En
`platforms/z1/contiki-z1-main.c`

```
#ifdef SERIALNUM
       if(!node_id) {
              PRINTF("Node id is not set, using Z1 product ID\n");
              node_id = SERIALNUM;
       }
#endif
node_mac[0] = 0xc1; /* Hardcoded for Z1 */
node_mac[1] = 0x0c; /* Hardcoded for Revision C */
node_mac[2] = 0x00; /* Hardcoded to arbitrary even number so that the
802.15.4 MAC address is compatible with an Ethernet MAC address - byte 0
(byte 2 in the DS ID) */
node_mac[3] = 0x00; /* Hardcoded */
node_mac[4] = 0x00; /* Hardcoded */
node_mac[5] = 0x00; /* Hardcoded */ node_mac[6] = node_id >> 8;
node_mac[7] = node_id & 0xff;
       }
```

Así que la mota debería tener las direcciones siguientes:

```
MAC c1:0c:00:00:00:00:0c:e5
Node id is set to 3301.
Tentative link-local IPv6 address
fe80:0000:0000:0000:c30c:0000:0000:0ce5
```

Donde ce5 es el valor hexadecimal que corresponde a 3301. La dirección global se establece sólo cuando se asigna un prefijo IPv6 (esto fue explicado en las secciones anteriores).

Si en cambio, quieres tener tu propio esquema de direcciones, puedes editar los valores de node_mac en el archivo contiki-z1-main.c. Si quieres reemplazar la id del nodo obtenido a partir de la id del producto, necesitas almacenar una nueva en la memoria flash. Por suerte existe ya una aplicación para esto:

Ve a la ubicación examples/z1 y reemplaza el 158 por el valor que necesites:
make clean && make burn-nodeid.upload nodeid=158 nodemac=158 && make z1-reset && make login

Deberías ver lo siguiente:

```
MAC c1:0c:00:00:00:00:0c:e5 Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 3301.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address
fe80:0000:0000:0000:c30c:0000:0000:0ce5
Starting 'Burn node id'
Burning node id 158
Restored node id 158
```

Como puedes ver, la ID del nodo ha cambiado a 158. Cuando reinicies la mota se debería ver los cambios aplicados:

```
MAC c1:0c:00:00:00:00:00:9e Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address
fe80:0000:0000:0000:c30c:0000:0000:009e
```

Configura el ancho de banda y el canal

El ancho de banda y los canales permitidos dependen de la banda de frecuencia utilizada. Estos serán determinados por el ente regulador del espectro de cada país, al igual que la potencia máxima de transmisión.

El estándar IEEE 802.15.4

El estándar IEEE 802.15.4 es un estándar para comunicaciones inalámbricas. Describe las capas física y de control de acceso al medio para redes de área personal de baja tasa de transmisión (LR-WPAN).

El estándar describe el uso de la banda 868-868.6 MHz (en Europa y muchos otros países), de 902-928 MHz (en Estados Unidos, Canadá y algunos países de Latinoamérica) o la más usada, la de 2.400-2.4835 GHz que es parte de las denominadas aplicaciones Industriales, Científicas y Médicas (ISM).

OPTIONS FOR FREQUENCY ASSIGNMENTS			
Geographical regions	Europe	Americas	Worldwide
Frequency assignment	868 to 868.6 MHz	902 to 928 MHz	2.4 to 2.4835 GHz
Number of channels	1	10	16
Channel bandwidth	600 kHz	2 MHz	5 MHz
Symbol rate	20 ksymbols/s	40 ksymbols/s	62.5 ksymbols/s
Data rate	20 kbits/s	40 kbits/s	250 kbits/s
Modulation	BPSK	BPSK	Q-QPSK

Figura 43. Requisitos regulatorios de IEEE 802.15.4 (electronicdesign.com, 2013)
http://electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless

En la práctica, la banda de 2.4 GHz es ampliamente usada debido a su disponibilidad en todo el mundo. El protocolo patentado ZigBee de la alianza ZigBee fue uno de los primeros en adoptar IEEE 802.15.4. Lo hizo aprovechando las capas física y MAC del IEEE 802.15.4, añadiéndoles funciones adicionales de enrutamiento y conexión a fin de construir redes en malla.

Más recientemente el Thread Group [https:// threadgroup.org] ha propuesto su propio protocolo simplificado para redes malladas basado en IPv6, para conectar dispositivos dentro de las casas, entre sí, a la Internet y a la nube. Esto seguramente impulsará la adopción de IEEE 802.15.4 y de IPv6.

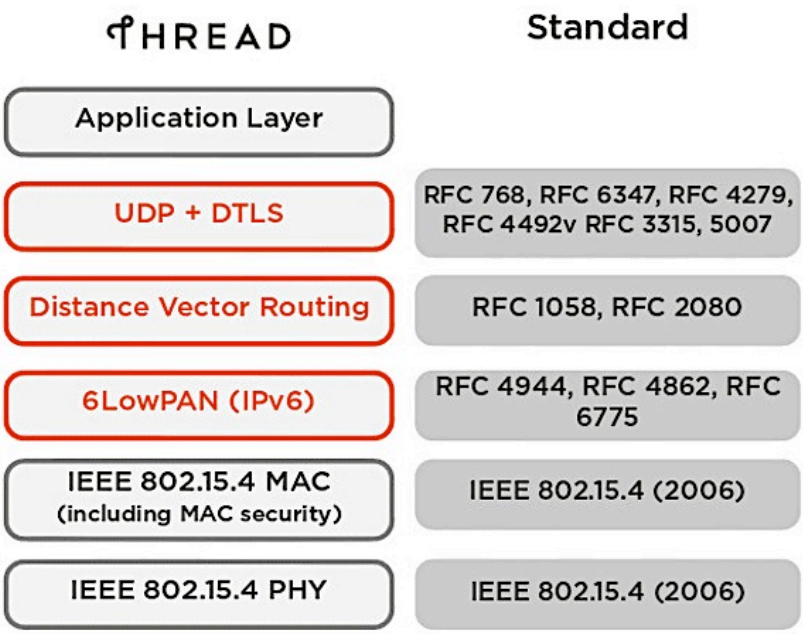

Figura 44. Capas Thread y estándares. (Thread group, 2015) http://threadgroup.org/

Trabajando en 2.4 GHz

Puesto que la banda de 2.4 GHz también la usan otras tecnologías como WiFi y Bluetooth, este espectro es compartido y pueden ocurrir solapamientos. La figura siguiente muestra la adjudicación de canales a 2.4 GHz en el IEEE 802.15.4 y los canales recomendados para evitar solapamientos con otros dispositivos coexistentes. Con el surgimiento de Bluetooth Low Energy y la ubicuidad de WiFi en nuestras vidas, la selección de un canal de radio frecuencia apropiado se hace crucial para cualquier despliegue.

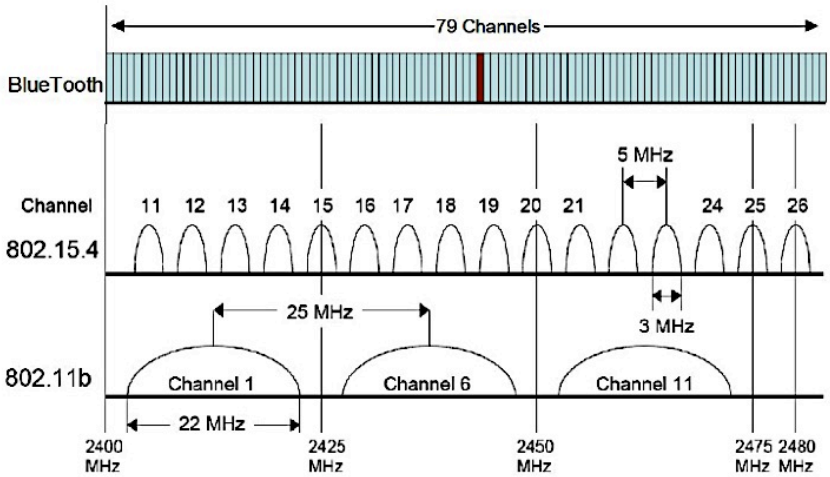

Figura 45. Asignación de canales a 2.4 GHz. Tomado de https://sing.stanford.edu/pubs/sing-08-03.pdf "An Empirical Study of Low Power Wireless"

El canal predeterminado de la **RE-Mote** se define a continuación:

```
#ifndef CC2538_RF_CONF_CHANNEL
#define CC2538_RF_CONF_CHANNEL              25
#endif /* CC2538_RF_CONF_CHANNEL */
```

El canal predeterminado para la **mota Z1** se define así:

```
#ifdef RF_CHANNEL
#define CC2420_CONF_CHANNEL RF_CHANNEL
#endif

#ifndef CC2420_CONF_CHANNEL
#define CC2420_CONF_CHANNEL             26
#endif /* CC2420_CONF_CHANNEL */
```

El canal de radio puede definirse a partir de del archivo project-conf.h de la aplicación, o a partir del Makefile.

Los drivers del radio en Contiki están implementados para ajustarse a `struct radio_driver en core/edv/radio.h.` Esta abstracción permite interactuar con el radio usando una interfaz API estandarizada independientemente del hardware del radio. Las funciones para establecer y leer los parámetros del radio se explican a continuación.

```
/** Get a radio parameter value. */
radio_result_t (* get_value)(radio_param_t param, radio_value_t *value);

/** Set a radio parameter value. */
radio_result_t (* set_value)(radio_param_t param, radio_value_t value);
```

Para cambiar el canal en la aplicación use RADIO_PARAM_CHANNEL como sigue:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, value);
```

Donde *value* puede ser cualquier valor entre 11 y 26, y rd puede ser:

`RADIO_RESULT_INVALID_VALUE` o `RADIO_RESULT_OK`.

Trabajando en 863-950 MHz

La **RE-Mote** tiene dos interfaces RF, a 2.4 GHz y a 863-950 MHz, que pueden ser seleccionadas alternativamente, o usadas simultáneamente (actualmente, lo último no es posible con Contiki).

Por defecto, la RE-Mote usa el modo IEEE 802.15.4g obligatorio para la banda de 868 MHz, configurado para modulación 2GFSK, tasa de transmisión de datos de 50 kbps y con 33 canales disponibles.

La RE-Mote usa el radio CC1200 de Texas Instruments, que en Contiki está implementado en dev/cc1200. El archivo de la configuración por defecto se encuentra en dev/cc1200/cc1200-802154g-863-870-fsk-50kbps.c.

Para cambiar canales en la aplicación usamos la API RF:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, value);
```

Donde *value* puede ser cualquier valor desde 11 a 26 y rd será:

`RADIO_RESULT_INVALID_VALUE` o `RADIO_RESULT_OK`.

Establecer la potencia de transmisión

La potencia de transmisión de radio frecuencia es aquella a la salida del transmisor antes de tomar en cuenta el efecto de la ganancia de la antena. A mayor potencia de transmisión, mayor alcance inalámbrico; pero el consumo de energía crece también. El alcance depende en gran medida de la frecuencia, de la antena usada y su altura sobre el suelo.

Cambiar la potencia de transmisión en la RE-Mote (CC2538) y en la Z1 (CC2420)

La plataforma **RE-Mote** usa el radio a 2.4GHz incorporado en el SoC CC253. Por defecto, en la cabecera cpu/cc2538/cc2538-rf.h la potencia de transmisión es de 3 dBm (2 mW), controlable como se muestra a continuación.

```
CC2538_RF_TX_POWER_RECOMMENDED 0xD5
```

Este valor recomendado fue tomado de SmartRF Studio.
http://www.ti.com/tool/smartrftmstudio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio

En la siguiente tabla se muestran otros valores y sus correspondientes niveles de potencia de salida:

Tabla 1. Valores recomendados de potencia de transmisión (de SmartRF Studio)

Potencia TX (dBm)	Valor
+7	0xFF
+5	0xED
+3	0xD5
+1	0xC5
0	0xB6
-1	0xB0
-3	0xA1
-5	0x91
-7	0x88
-9	0x72
-11	0x62
-13	0x58
-15	0x42
-24	0x00

Dato	Por ilógico que parezca, hay buenas razones para reducir la potencia de transmisión: • Para reducir el consumo de energía. • Para hacer ensayos en una red multi-salto sin tener que ubicar los nodos demasiado lejos (el receptor puede saturarse si la potencia es demasiado alta). • Para evitar interferencia entre redes coexistentes en la misma área.

El consumo de corriente puede ir de 24 mA a 34 mA cuando se cambia la potencia de transmisión de 0 dBm a 7 dBm (AN125) [http://www.ti.com/lit/an/swra437/swra437.pdf]

La **mota Z1** usa el radio RF CC2420 de Texas Instruments. La potencia de transmisión está fijada por defecto en 0 dBm (1mW), que es la máxima permitida por el radio.

En la siguiente tabla se muestran la potencia de salida y sus correspondientes valores de configuración, así como el consumo de corriente en cada caso.

Tabla 2. Potencia de transmisión del CC2420 CC2420 datasheet, page 51)
[http://www.ti.com/lit/ds/symlink/cc2420.pdf]

Potencia TX (dBm)	Valor	mA
0	31	17.4
-1	27	16.5
-3	23	15.2
-5	19	13.9
-7	15	12.5
-10	11	11.2
-15	7	9.9
-25	3	8.5

En ambas plataformas la potencia de transmisión puede cambiarse con:

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_TXPOWER, value);
```

Donde value puede ser cualquier valor de la tabla anterior, y rd puede ser

`RADIO_RESULT_INVALID_VALUE` o `RADIO_RESULT_OK`.

Para cambiar la potencia de transmisión en la RE-Mote (CC1200)

Como dijimos antes, la **RE-Mote** tiene también una interfaz que opera en la banda de 863 a 950 MHz usando el radio CC1200. La potencia máxima de transmisión permitida depende de la banda específica y de las regulaciones del lugar, que pueden también establecer el valor máximo de la ganancia de la antena. Hay que estar seguro sobre las regulaciones locales antes de cambiar la potencia de salida. Las regulaciones son específicas de cada país y no serán tratadas en este libro.

Los siguientes valores se han tomado de SmartRF Studio, usando por defecto la configuración IEEE 802.15.4g que adhiere al estándar ETSI.
http://www.ti.com/tool/smartrftmstudio&DCMP=hpa_rf_general&HQS=Other+OT+smartrfstudio

Tabla 3. Valores de potencia de transmisión de CC1200 recomendados (de SmartRF Studio)

Potencia TX (dBm)	Valor
+14	0x7F
+13	0x7C
+12	0x7A
+11	0x78
+8	0x71
+6	0x6C
+4	0x68
+3	0x66
+2	0x63
+1	0x61
0	0x5F
-3	0x58
-6	0x51
-11	0x46
-24	0x42
-40	0x41

Estos valores corresponden al registro CC1200_PA_CFG1

En Contiki, el driver CC1200 driver
[https://github.com/contikios/contiki/blob/master/dev/cc1200/cc1200.c]
usa por defecto el máximo de la potencia de transmisión por defecto, y se define de esta manera:

```
/* The maximum output power in dBm */
#define RF_CFG_MAX_TXPOWER              CC1200_CONST_TX_POWER_MAX
```

Los valores mínimos y máximos permitidos están establecidos en dev/cc1200/cc1200-const.h como se muestra a continuación:

```
  /* Output power in dBm */
  /* Up to now we don't handle the special power levels PA_POWER_RAMP <
  3, hence
 * the minimum tx power is -16. See update_txpower().
 */ #define CC1200_CONST_TX_POWER_MIN       (-16)
 /*
    * Maximum output power will depend on the band we use due to
    * regulation issues
    */
 #define CC1200_CONST_TX_POWER_MAX       14
```

El driver CC1200 calcula el valor apropiado del registro CC1200_PA_CFG1, así que necesitamos pasar como argumento value la potencia de transmisión solicitada.

```
rd = NETSTACK_RADIO.set_value(RADIO_PARAM_TXPOWER, value);
```

Donde *value* puede ser cualquier valor desde -14 hasta 16 y rd puede ser

```
RADIO_RESULT_INVALID_VALUE o  RADIO_RESULT_OK.
```

Análisis del enlace inalámbrico

Debido a las condiciones ambientales cambiantes que afectan normalmente los sistemas inalámbricos, como la lluvia, interferencia, obstáculos, etc., medir el medio inalámbrico y la calidad de los enlaces es importante.

El chequeo del medio inalámbrico debería hacerse en tres etapas: antes de desplegar la red, en la fase de despliegue y más tarde, cuando la red esté funcionando para asegurarnos de que los nodos creen y seleccionen las mejores rutas a disponibles.

Cálculo de la calidad del enlace

El cálculo de la calidad del enlace es parte integral de asegurar la confiabilidad en las redes inalámbricas. Varias mediciones para el cálculo del enlace se han propuesto para medir efectivamente la calidad de los enlaces inalámbricos.

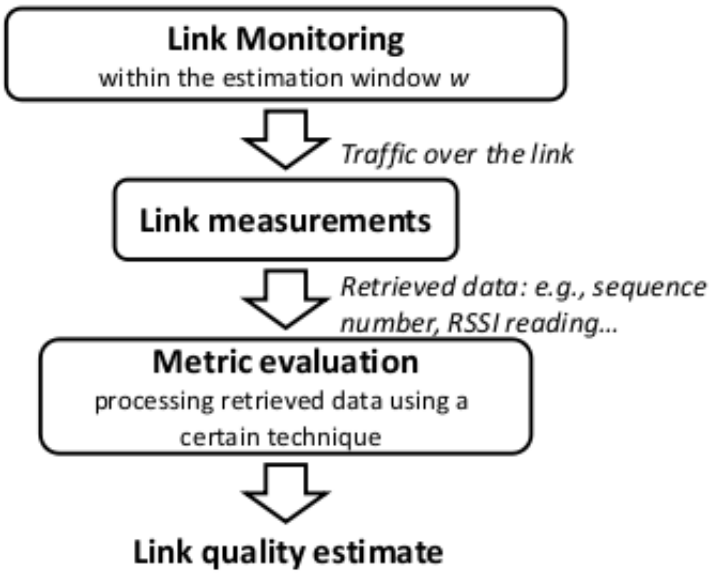

Figura 46. Estimación de la calidad del enlace. Tomado del artículo referenciado en: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5362

La ETX (*Expected Transmission Count)* o número esperado de retransmisiones, es una medida de la calidad del enlace entre dos nodos en una red inalámbrica de transmisión de paquetes de datos. ETX es un estimado del número de veces que se espera que haya que transmitir un paquete para que este sea recibido en su destino sin errores. Este número varía de 0 a infinito. Un ETX de 1 indica un enlace perfecto y una ETX con valor infinito representa un enlace completamente disfuncional. Nótese que ETX es una cantidad referida a eventos futuros, no a un conteo de eventos pasados. Es, por lo tanto, un número real, generalmente no entero.

ETX puede usarse como un parámetro de medida para comparar diferentes rutas. Las rutas con un valor más bajo se prefieren. En una ruta que incluye saltos múltiples, la medida es la suma de las ETX de los saltos individuales.

Una primera aproximación sobre las calidad del enlace se puede juzgar a partir de los valores de RSSI y LQI

¿Qué es RSSI?

RSSI (*Received Signal Strenght Indicator*) o Indicador de Intensidad de la Señal Recibida es una medida genérica que se usa internamente en un dispositivo de red inalámbrica para indicar la cantidad de energía que se recibe en un canal dado. Su valor se puede leer usando herramientas de monitoreo de redes inalámbricas tales como Wireshark, Kismet o Inssider.

La imagen siguiente muestra cómo la Packet Reception Rate (PRR) o Tasa de Recepción de Paquetes disminuye dramáticamente a medida que caen los valores de RSSI en el CC2420.

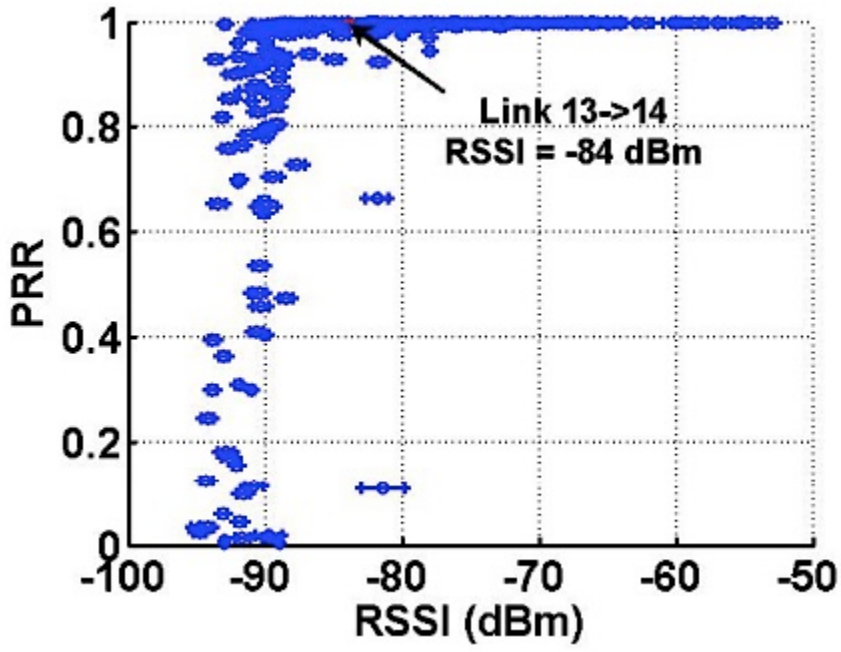

Figura 47. Tasa de recepción de paquetes versus indicador de intensidad de señal recibida. Tomado del artículo referenciado en:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5362

No hay una relación estandarizada entre algún parámetro físico en particular y el valor del RSSI. Los vendedores y fabricantes de chipset usan diferentes criterios para establecer la granularidad, precisión y rango de los valores RSSI y su relación con la potencia real (medida en mW o dBm) de la señal.

Existen dos lecturas de RSSI diferentes:

- La primera indica la cantidad de potencia presente en el medio inalámbrico a la frecuencia y el tiempo dados. En ausencia de algún paquete en el aire, esto representa el piso de ruido. Esta medida se usa también para decidir si el medio está

libre y disponible para el envío de un paquete. Un valor alto podría deberse a interferencia o a la presencia de un paquete en el aire.

• La segunda medida se realiza sólo después de que un paquete ha sido decodificado correctamente, y nos da la intensidad del paquete recibido desde un nodo específico.

La primera medida se puede leer usando la API del radio de esta manera:

```
rd = NETSTACK_RADIO.get_value(RADIO_PARAM_RSSI, value);
```

Donde *value* es una variable pasada como un puntero para almacenar el valor de RSSI, y rd será RADIO_RESULT_INVALID_VALUE o RADIO_RESULT_OK.

La segunda medida, correspondiente al valor RSSI de un paquete correctamente decodificado, en la función de retorno `receive:`

```
packetbuf_attr(PACKETBUF_ATTR_RSSI);
```

Más información sobre los atributos de `packetbuf` en core/net/packetbuf.h

Para el radio CC24120 de la **mota Z1**, el RSSI puede fluctuar entre valores de 0 a -100. Los valores cerca de 0 indican buenos enlaces, mientras que los cercanos a -100 indican enlaces malos, que pueden ser debidos a múltiples factores como distancia, ambiente, obstáculos, interferencia, etc.

¿Qué es LQI?

El Indicador de la Calidad del Enlace (Link Quality Indicator: **LQI**) es un valor digital proporcionado por los vendedores de chipsets como un indicador de cuán bien se está demodulando la señal. Mide el error que presenta la modulación de los paquetes recibidos satisfactoriamente en comparación con la señal esperada en un canal perfecto.

El ejemplo que sigue muestra cómo la Tasa de Recepción de Paquetes (PRR) disminuye cuando el Indicador de la Calidad del Enlace (LQI) se reduce.

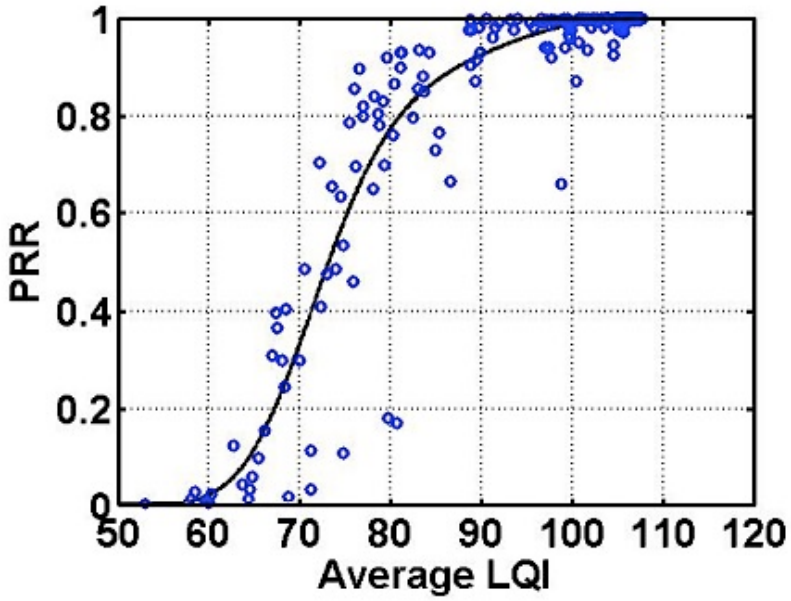

Figura 48. Tasa de recepción de paquetes versus calidad del enlace. Tomado del artículo referenciado en:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5362

Para leer el valor LQI usamos la API del radio:

```
rd = NETSTACK_RADIO.get_value(PACKETBUF_ATTR_LINK_QUALITY, value);
```

Donde value es una variable pasada como un puntero para almacenar valor del LQI , y rd será RADIO_RESULT_INVALID_VALUE o RADIO_RESULT_OK.

Este valor en el radio CC2420 usado por la mota Z1 normalmente oscila entre 110 (indica una trama de máxima calidad) y 50 (normalmente las tramas de más baja calidad detectables por el transceptor).

Más información sobre el cálculo de LQI de CC2538 se encuentra en la guía del usuario del CC2538: http://www.ti.com/lit/ug/swru319c/swru319c.pdf]

Configuración de la capa MAC

Protocolos MAC

Los protocolos del Medium Access Control (MAC) o Control del Acceso al Medio describen el acceso al medio adoptado en una red fijando las reglas que establecen cuándo a un determinado nodo le está permitido transmitir paquetes.

Los protocolos pueden clasificarse en basados en contención (*contention-based)* y basados en reserva (*reservation-based)*.

Los primeros se basan en la detección de portadora (*carrier sensing)* para decidir si el medio está ocupado, son susceptibles a las colisiones y tienen baja eficiencia cuando la red está muy cargada, pero son fáciles de implementar. El segundo grupo es eficiente en términos de rendimiento (*throughput*) y energía, pero requiere una sincronización muy precisa y son menos adaptable al tráfico dinámico.

La implementación de acceso al medio en Contiki tiene 3 capas diferentes: Framer, Radio Duty-Cycle (RDC) y Control de Acceso al Medio (MAC) propiamente dicha.

Figura 49. La pila de control de acceso al medio en Contiki
(ANGR, USC 2014)
http://http://anrg.usc.edu/contiki/index.php/MAC_protocols_in_ContikiOS

La capa de red puede accederse por las variable globales NETSTACK_FRAMER, NETSTACK_RDC y NETSTACK_MAC, que se definen en el momento de la compilación.

Estas variables están en core/net/netstack.h y pueden definirse como predeterminadas por cada plataforma y son sobreescritas por las aplicaciones.

Driver MAC

Contiki trae dos drivers: CSMA y NullMAC

CSMA (*Carrier-Sense Multiple Access*). Acceso Múltiple por Detección de Portadora: recibe los paquetes entrantes de la capa RDC y usa esta capa para la transmisión de los paquetes. Si RDC o la capa de radio detectan que el medio está ocupado, la capa MAC puede retransmitir el paquete más tarde. El protocolo CSMA mantiene una lista de paquetes enviados a cada vecino y realiza ciertas estadísticas, como por ejemplo número de retransmisiones, colisiones, aplazamientos, etc. El chequeo del acceso al medio lo realiza el driver RDC.

NullMAC es un protocolo de transferencia simple (*pass-through*). Invoca las funciones RDC apropiadas.

Tanto **Z1** como **RE-Mote** usan el driver CSMA por defecto.

```
#ifndef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC     csma_driver
#endif
```

Alternativamente, se puede utilizar NullMac de esta manera:

```
#define NETSTACK_CONF_MAC nullmac_driver
```

Driver RDC

La capa *Radio Duty-Cycle* (RDC) maneja los periodos durmientes de los nodos. Esta capa decide cuándo se van a transmitir los paquetes y asegura que los nodos estén despiertos cuando se van a recibir los paquetes.

La implementación de los protocolos de Contiki se encuentra en core/net/mac.
Los siguientes drivers están implementados: contikimac, xmac, lpp, nullrdc y sicslowmac. La implementación y detalles de los mencionados drivers RDC no forman parte del objetivo de esta sección. El más común es ContikiMAC.
NullRDC es una capa de transferencia que nunca apaga el radio.

```
#ifndef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC contikimac_driver
#endif
```

Los drivers RDC tratan de mantener el radio apagado por el mayor tiempo posible, chequeando regularmente la actividad en el medio inalámbrico. Cuando se detecta alguna actividad, el radio se enciende para permitir que se reciban paquetes, luego puede volver a "dormir".

La tasa de chequeo del canal se da en Hz y especifica el número de veces por segundo que se verifica el estado del canal. Las tasas de chequeo del canal se expresan en potencias de dos y los valores típicos son 2, 4, 8 y 16 Hz, siendo 8 Hz el valor por defecto.

```
#ifndef NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
#define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE 8
#endif
```

Generalmente, un paquete debe ser retransmitido varias veces (*strobed*) hasta que el receptor esté encendido y lo reciba. Esto aumenta el consumo de energía del transmisor y el tráfico en el medio, pero el ahorro de energía cuando el receptor está dormido es más significativo, siendo el efecto global un ahorro neto de energía.

Una alternativa para optimizar RDC es activar "optimización de fase" que retrasa el envío de paquetes hasta un momento justo antes de que el receptor despierte. Sin embargo, esto exige una muy buena sincronización entre el transmisor y el receptor (ver más detalles en RDC Phase Optimization [https://github.com/contiki-os/contiki/wiki/RDC-Phase-optimization]). Para activar la optimización de fase, cambia el 0 a 1 en lo que sigue:

```
#define CONTIKIMAC_CONF_WITH_PHASE_OPTIMIZATION 0
#define WITH_FAST_SLEEP 1
```

El driver Framer

Este driver es en realidad un conjunto de funciones para enmarcar (*framing*) los datos que van a transmitirse y para analizar los datos recibidos. Las implementaciones *Framer* se encuentran en `core/net/mac`, de las cuales las más importantes son framer-802154 y `framer-nullmac`.

En la plataforma **RE-Mote** la configuración predeterminada es:

```
#ifndef NETSTACK_CONF_FRAMER
#if NETSTACK_CONF_WITH_IPV6
#define NETSTACK_CONF_FRAMER  framer_802154
#else /* NETSTACK_CONF_WITH_IPV6 */
#define NETSTACK_CONF_FRAMER  contikimac_framer
#endif /* NETSTACK_CONF_WITH_IPV6 */
#endif /* NETSTACK_CONF_FRAMER */
```

Lo que quiere decir que cuando se usa **IPv6,** se selecciona framer-802154, de otra manera, se usa contikimac_framer (el *framer* por defecto para el contikimac_driver).

El *framer* framer-nullmac debería usarse junto con el nullmac_driver (capa MAC). Este simplemente llena los dos campos de nullmac_hdr que son: dirección del receptor y dirección del remitente.

El framer-802154 se implementa en core/net/mac/framer-802154.c.
El driver encuadra los datos adhiriendo al estándar IEEE 802.15.4 (2003). El *framer* inserta y extrae los datos en/de la estructura packetbuf.

IPV6 y enrutamiento

Una de las características más importantes de Contiki es su compatibilidad con los protocolos IP. De hecho, ha sido uno de los primeros sistemas operativos incrustados compatibles con IPv6.

Contiki también es compatible con IPv4 y otras comunicaciones no-IP (Rime)
https://github.com/alignan/contiki/tree/master/core/net/rime.

Sin embargo, en el resto del libro nos enfocaremos en IPv6. Hay una buena cantidad de ejemplos *rime* en examples/rime. El ejemplo más básico de **RE-Mote** remote-demo.c que se encuentra en examples/remote también usa *rime.*

IPv6

El **uIP** es una pila TCP/IP de fuente abierta diseñada para usarse con microcontroladores de 8 y 16 bits. Fue desarrollado inicialmente por Adam Dunkels [http://dunkels.com/adam/] cuando estaba en el Swedish Institute of Computer Science (SICS). **uIP** tiene una licencia tipo BSD (*Berkeley Software Distribution*) y fue ulteriormente desarrollado por un numeroso grupo de trabajadores.

Los detalles de la implementación del uIP/IPv6 no serán tratados aquí. A continuación se explican las configuraciones básicas de la plataforma y la aplicación.

Para activar IPv6 hay que definir lo siguiente, bien sea en el archivo de la aplicación Makefile , o en el archivo project-conf.h:

```
#define UIP_CONF_IPV6  1

#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
/* uIP */
#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
```

RPL

Hay muchas posibilidades de escogencia respecto al enrutamiento, pero a la postre todas hacen lo mismo: garantizan que los paquetes lleguen a buen destino. Esto se va a hacer de diferentes maneras dependiendo de factores como la métrica de enrutamiento (*routing metric*), es decir, qué criterio se usa para comparar rutas alternativas (cómo una ruta va a ser clasificada mejor que otra), el hecho de utilizar enrutamiento dinámico o estático, etc.

RPL es el protocolo de enrutamiento predeterminado en Contiki. Hay otros, como *Ad hoc On-Demand Distance Vector* (AODV) que no serán tratados aquí.

Los detalles de RPL no son nuestro objetivo. Aquí describiremos solamente la configuración común y haremos una breve introducción a RPL. Para más detalles, ir a core/net/rpl.

¿Qué es RPL?

RPL es un protocolo de enrutamiento para redes de bajo consumo con pérdidas, diseñado por el grupo **ROLL**: *Routing Over Low Power and Lossy network* de la *IETF* (Enrutamiento en Redes de baja Potencia con Pérdidas de la *Internet Engineering Task Force*). RPL es un protocolo por vector de distancia, proactivo, ya que comienza a encontrar las rutas tan pronto como se inicializa la red RPL.

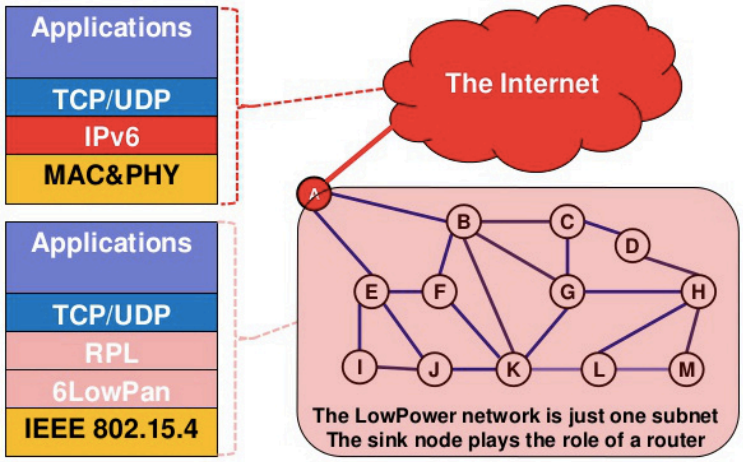

Figura 50. RPL en la pila de protocolos. Tomado de http://www.slideshare.net/asobimat/rpl-dodag

RPL permite tres patrones de tráfico:

- Multipunto a punto (MP2P)
- Punto a multipunto (P2MP)
- Punto a punto (P2P)

RPL construye rutas usando DODAG (Destination Oriented Directed Acyclic Graphs) (Gráficos Acíclicos Orientados al Destino) hacia un recolector (DAG ROOT) identificado por un identificador único DODAGID. Los DODAG se optimizan usando la métrica Objective Function , OF (Funcion Objetiva), identificada por un Objective Code Point, OCP (Punto Objetivo del Código), que indica las restricciones dinámicas y las mediciones, como conteo de saltos, latencia, conteo de transmisión esperada, selección de parents, consumo de energía, etc. Se asigna un rango (rank) a cada nodo, que puede usarse para determinar su posición y distancia respecto a la raíz en el DODAG.

Dentro de una red determinada puede haber varias instancias (instances) de RPL lógicamente independientes. Un nodo RPL puede pertenecer a varias instancias de RPL, y puede actuar como un enrutador en unos y como hoja (leaf) en otros. Un conjunto de múltiples DODAG pueden estar en una instancia de RPL, y un nodo puede ser miembro de varias instancias de RPL, pero puede pertenecer solamente a un DODAG por cada instancia de DAG.

Un temporizador de goteo regula la transmisión de mensajes DODAG Information Object (DIO), que son usados para crear y mantener en ascenso las rutas del DODAG, informando su instancia RPL, DODAG ID, RANK, y el número de la versión DODAG.

Un nodo puede solicitar información DODAG enviando mensajes DODAG de Solicitud de Información (DIS), solicitando mensajes DIO a sus vecindades para actualizar la información de enrutamiento y juntarse a una instancias.

Los nodos tienen que monitorizar los mensajes DIO antes de unirse a un DODAG, y luego unirse a un DODAG seleccionando un nodo parent entre sus vecinos usando los datos de latencia anunciada, OF y RANK. Los mensajes DAO (Destination Advertisement Object) se usan para mantener las rutas en descenso usando la selección de parent preferido de menor rango y enviando un paquete al DAG ROOT a través de cada nodo intermedio.

RPL tiene dos mecanismos para reparar la topología del DODAG, el primero es para evitar los lazos y permitir a los nodos asociarse y reasociarse. El otro se llama reparación global y es iniciado en DODAG ROOT al incrementar el número de la versión DODAG para crear una versión DODAG nueva.

Más información sobre RPL en RFC6550. https://tools.ietf.org/html/rfc6550

Para habilitar el enrutamiento en las plataformas **Z1** y **RE-Mote** deben activarse:

```
#ifndef UIP_CONF_ROUTER
#define UIP_CONF_ROUTER  1
#endif
```

Para habilitar RPL, añade lo siguiente a la Makefile de tu aplicación o su archivo project-conf.h

```
#define UIP_CONF_IPV6_RPL  1
```

La configuración por defecto de la **RE-Mote** es la siguiente**:**

```
/* ND and Routing */
#define UIP_CONF_ND6_SEND_RA        0  (1)
#define UIP_CONF_IP_FORWARD         0  (2)
#define RPL_CONF_STATS              0  (3)
```

1) Deshabilita el envío de anuncios de enrutamiento
2) Deshabilita el reenvío de IP
3) Desactiva la configuración de estadísticas de RPL

El parámetro RPL_CONF_OF configura la función objetiva de RPL. El Minimum Rank with Hysteresis Objective Function (MRHOF) usa ETX como métrica de enrutamiento y también tiene derivaciones (stubs) para mediciones de energía.

```
#ifndef RPL_CONF_OF
#define RPL_CONF_OF rpl_mrhof
#endif
```

La (ETX) computa cuántos intentos se necesitan para recibir el reconocimiento (ACK) de un paquete enviado manteniendo un promedio para cada vecino y computando la suma de todas las EXT para crear las rutas.

Contiki usa por defecto "*storing mode*" para rutas RPL descendientes. Básicamente, todos los nodos almacenan en una tabla de enrutamiento las direcciones de sus nodos hijos.

Para instalar un rastreador de paquetes (sniffer)

Una de las herramientas que se deben tener cuando se desarrollan aplicaciones inalámbricas es un sniffer que permite ver lo que se está transmitiendo en el aire y verifica que las transmisiones se están realizando, que las tramas/paquetes estén formateados apropiadamente y que la comunicación se produzca en un canal determinado.

Podemos contar con opciones comerciales, por ejemplo el SmartRF packet Sniffer [http://www.ti.com/tool/packet-sniffer] de Texas Instruments, que puede usarse con la dongle USB CC2531 :[http://www.ti.com/tool/CC2531EMK] para captar paquetes como el siguiente:

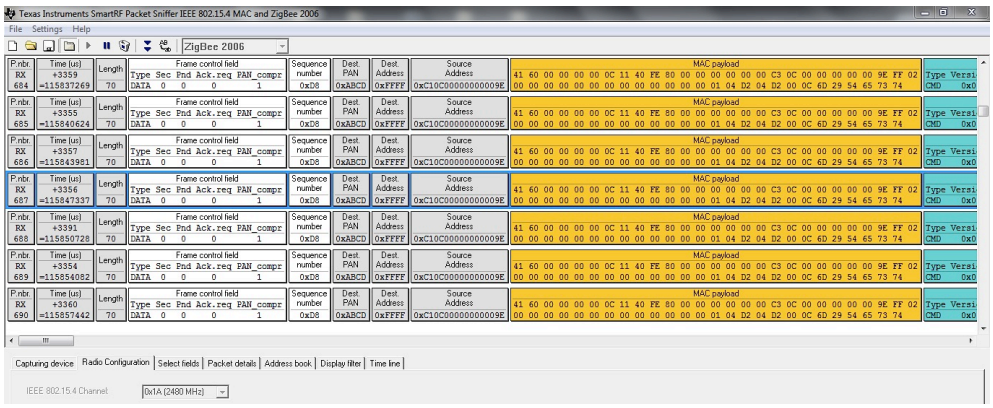

Figura 51. Pantalla de Sniffer. (Imagen propia)

Para este ejercicio usaremos la aplicación SensSiff [https://github.com/g-oikonomou/sensniff] en una **RE-Mote** y Wireshark (ya instalado en instant Contiki). Esta combinación nos permitirá entender cómo se realiza la comunicación inalámbrica en Contiki.

Para programar la **RE-Mote** como un rastreador (*sniffer*) de paquete:

```
cd examples/cc2538dk/sniffer
```

Compila y programa:

```
make TARGET=remote sniffer.upload
```

Información	Al momento de esta redacción el sniffer Z1 no estaba oficialmente incluido en Contiki. Sin embargo, una rama con su implementación se encuentra en: https://github.com/alignan/contiki/tree/z1_sniffer/examples/z1/sniffer

Abre un nuevo terminal y clona el proyecto sensniff en tu carpeta home:

```
cd $HOME
git clone https://github.com/g-oikonomou/sensniff
cd sensniff/host
```

Ahora lanza la aplicación sensniff con el comando siguiente:

```
python sensniff.py --non-interactive -d /dev/ttyUSB0 -b 115200
```

Sensniff va a leer los datos de la mota usando el puerto serial, a diseccionar las tramas y conectarlas (pipe) a /tmp/sensniff por defecto. Ahora necesitamos conectar el otro extremo de la tubería a wireshark. De otra manera, recibirás la advertencia siguiente: "Remote end not reading"

Esto no debe alarmarnos. Significa que el otro extremo de la tubería no está conectado. También puedes guardar las tramas "rastreadas" para abrirlas más tarde con wireshark añadiendo el siguiente argumento al comando anterior: -p name.pcap. Esto guardará la salida de la sesión en un archivo name.pcap. Cambia el nombre y la ubicación para almacenar el archivo adecuadamente.

Información	En el momento de la redacción de este tutorial el cambio de canales de la aplicación sensniff no estaba implementado, sino propuesto como una característica. Revisa el archivo `README.md` de Sensniff para ver los cambios y el estado actual.

Abre otro terminal e instala wireshark con el comando siguiente que añadirá la tubería (*pipe*) como una interfaz de captura.

```
sudo wireshark -i /tmp/sensniff
```

Selecciona la interfaz /tmp/sensniff de la lista descendente y haz clic en `Start` justo al comienzo.

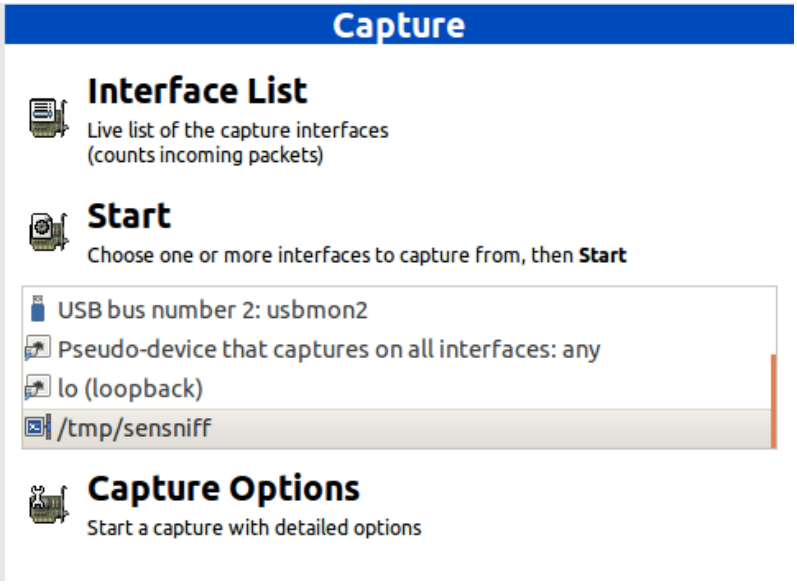

Figura 52. Pantalla Capture. (Imagen propia)

Asegúrate de que la tubería esté configurada para captura de paquetes en modo promiscuo. Si lo necesitas, se puede incrementar el tamaño del buffer, pero 1 MB normalmente basta.

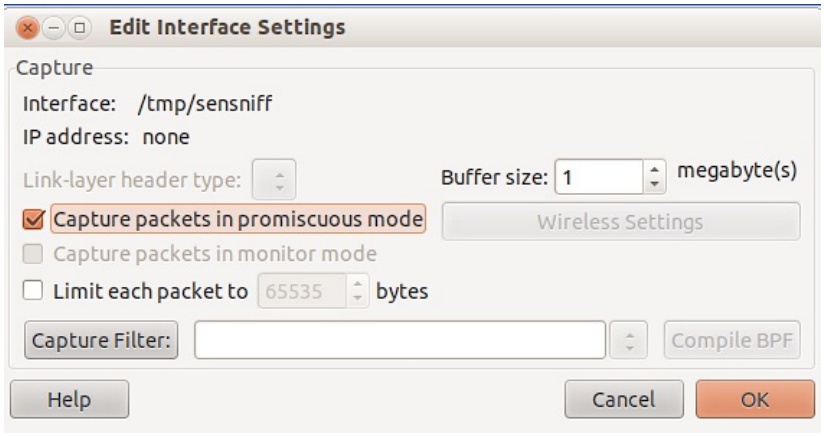

Figura 53. Edición de Interfaces. (Imagen propia)

Ahora, las tramas capturadas deberían comenzar a aparecen en la pantalla.

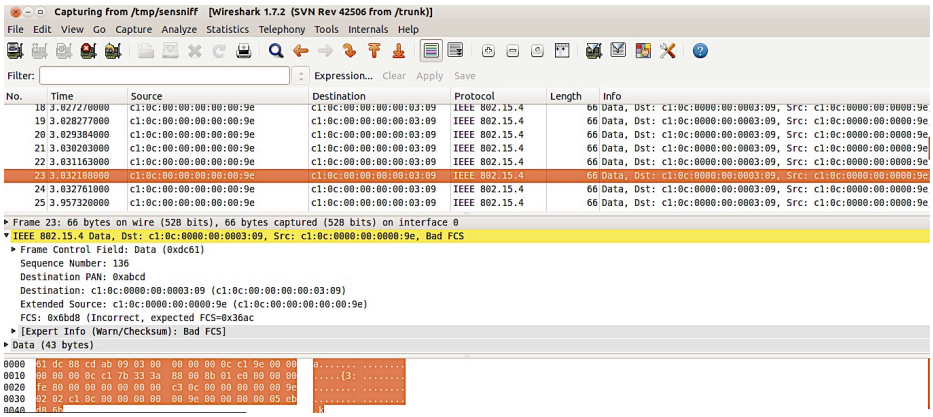

Figura 54. Tramas capturadas en Wireshark. (Imagen propia)

Puedes añadir filtros específicos para limitar las tramas que serán mostradas en pantalla. Para este ejemplo, haz clic en el botón Expression y aparecerá una lista de atributos disponibles por protocolo. Baja hasta IEEE 802.15.4 para chequear los filtros disponibles. También puedes encadenar los diferentes argumentos de los filtros usando la casilla `Filter.` En este caso, sólo queríamos las tramas pertenecientes a PAN 0xABCD procedentes del nodo c1:0c::0309, así que usamos los atributos wpan.dst_pan y wpan.src64.

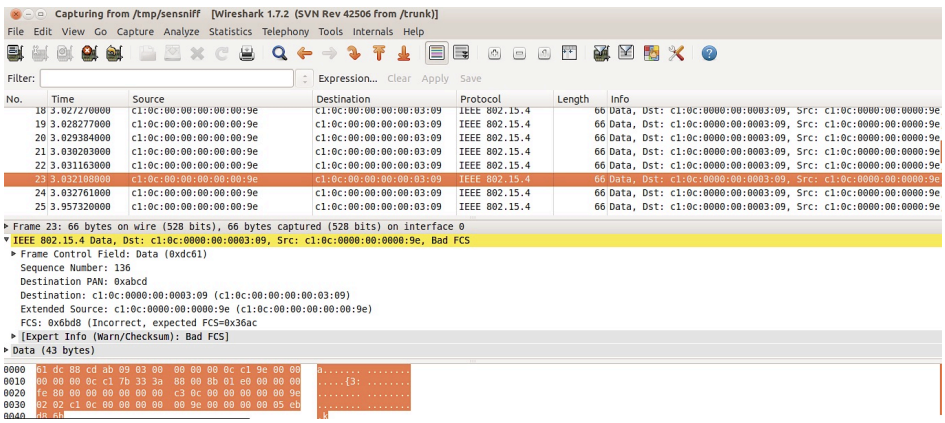

Figura 55. Filtros en Wireshark

Cuando se cierra la aplicación Sensniff aparece una sesión de información con las siguiente estadísticas:

```
Frame Stats:
  Non-Frame: 6
  Not Piped: 377
Dumped to PCAP: 8086
  Piped: 7709
  Captured: 8086
```

Ejercicio	¡Rastrea el tráfico! Trata de filtrar los datos salientes y entrantes usando tus propias reglas.

El enrutador de frontera

El enrutador de frontera o enrutador de borde (*border router*) es normalmente un dispositivo colocado en el borde de nuestra red que nos va a permitir hablar con redes externas usando sus interfaces de red incrustadas, por ejemplo WiFI, Ethernet, Serial, etc.

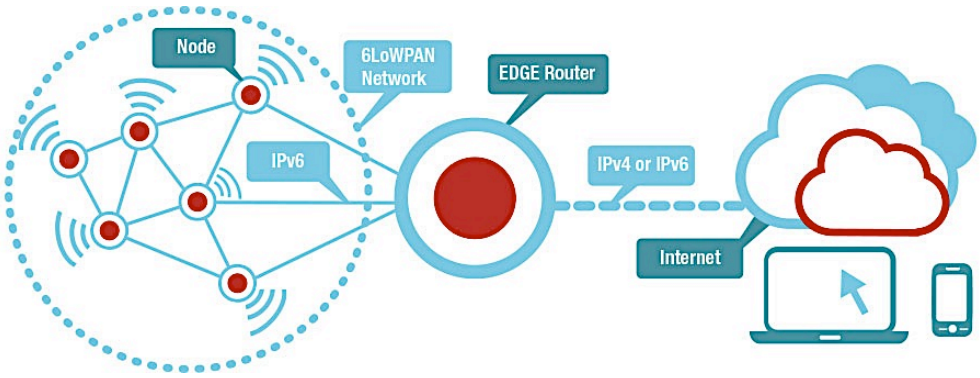

Figura 56. El entorno del enrutador de frontera. Tomado de http://processors.wiki.ti.com/index.php/Contiki-6LOWPAN

En Contiki, la aplicación de enrutador de frontera más corriente usa una interfaz serial llamada SLIP que permite la conexión de una mota dada con un host usando scripts como tunslip6 en tools/tunslip6 a través del puerto serial. Esto crea una interfaz de red en túnel a la que se puede asignar un prefijo IPv6 para establecer las direcciones globales de red IPv6.

La aplicación enrutador de frontera se encuentra en examples/ipv6/rpl-border-router. Los siguientes fragmentos de código son los más relevantes:

```
/* Request prefix until it has been received */
while(!prefix_set) {
  etimer_set(&et, CLOCK_SECOND);
  request_prefix();    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
}
```

```
dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)dag_id);
if(dag != NULL) {
  rpl_set_prefix(dag, &prefix, 64);
  PRINTF("created a new RPL dag\n");
}
```

El fragmento anterior induce una espera hasta que se asigne un prefijo válido. Una vez asignado, el nodo lo asume y se auto-convierte en el nodo raíz (DODAG).

Normalmente es preferible configurar el enrutador de frontera como un dispositivo no-durmiente para que el radio esté siempre encendido. Puedes configurar las especificaciones de este enrutador usando el archivo project-conf.h.

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC      nullrdc_driver
```

Por defecto, las aplicaciones de enrutador de frontera traen un servidor web incorporado que muestra información sobre la red, por ejemplo vecinos inmediatos (ubicados a un sólo salto) y las rutas conocidas a los nodos de sus redes. Para activar el servidor web debe habilitarse la bandera WITH_WEBSERVER lo que añadirá por defecto la aplicación `httpd-simple.c`.

Práctica: instalación del enrutador de frontera

La sección siguiente presupone el uso de **RE-Mote**, pero **Z1** también puede usarse.

```
make TARGET=remote savetarget
```

Para compilar, reprograma (flash) la mota y conecta el enrutador de frontera al host.
Ejecuta:

```
make border-router.upload && make connect-router
```

Por defecto, tratará de conectarse a una mota en el puerto /dev/ttyUSB usando la siguiente configuración del puerto serie: 115200 baudios, 8 bits, sin paridad, y 1 bit de parada. Si no le estableces un prefijo, IPv6 usará aaaa::1/64 como predeterminado. Para especificar uno diferente, emplea la herramienta tunslip tool usando lo siguiente:

```
make connect-router PREFIX=2001:abcd:dead:beef::1/64
```

Puedes también compilar y ejecutar la herramienta tunslip6 desde la ubicación de las herramientas. Para compilar, sólo escribe:

```
cd tools
cc tunslip6.c -o tunslip6
```

Y para trabajar con argumentos específicos, por ejemplo, conectarte a un puerto serie, designa el túnel con un nombre específico, o re-direcciónalo a una dirección y puerto dados usando:

```
./tunslip -s /dev/ttyUSB0 -t tun0 2001:abcd:dead:beef::1/64
```

Ejecuta tunslip –H para obtener más información.

Dato	6lbr [http://cetic.github.io/6lbr/] es una solución de enrutador de frontera 6LoWPAN basado en Contiki listo para despliegue. Es compatible con la **mota Z1** y en poco tiempo lo será para la plataforma **RE-Mote,** (el CC253 ya es compatible, la adaptación es trivial). Para llevar tu enrutador de frontera al siguiente nivel, esta es la herramienta que estabas buscando.

UDP y TCP básicos

Ahora que hemos visto las configuraciones de la mota y las capas MAC y de enrutamiento, instalemos una red UDP.

¿Qué es UDP?
UDP (User Datagram Protocol) es un protocolo de comunicación que ofrece una cantidad limitada de servicios para mensajes intercambiados entre dispositivos en una red que usa el Protocolo de Internet (IP).

UDP es una alternativa al Protocolo de Control de Transmisión (TCP) y junto con IP se conocen a menudo como UDP/IP. Al igual que TCP, UDP usa el Protocolo de Internet (IP) para mover una unidad de datos (llamada datagrama) de un computador a otro.

A diferencia de TCP, UDP no hace fragmentación y re-ensamblado de mensajes en el otro extremo, lo que significa que la aplicación debe garantizar que el mensaje llegue completo y en el orden correcto.

Las aplicaciones de red que quieren ahorrar tiempo de procesamiento porque tienen pocas unidades de datos para intercambiar (y por lo tanto, poco re-ensamblado de mensajes) podrían preferir UDP en lugar de TCP.

La implementación UDP de Contiki está en core/net/ip. A continuación nos enfocaremos en la descripción de las funciones UDP disponibles.

La API de UDP

Necesitamos crear un *socket* para la conexión. Esto se hace usando la estructura udp_socket que tiene los elementos siguientes:

```
struct udp_socket {
  udp_socket_input_callback_t input_callback;
  void *ptr;
  struct process *p;
  struct uip_udp_conn *udp_conn;
};
```

Después de crear la estructura del *socket* del UDP, tenemos que registrar el *socket*, lo que se hace con `udp_socket_register`.

```
/**
* \brief      Register a UDP socket
```

```
* \param c    A pointer to the struct udp_socket that should be
registered
* \param ptr  An opaque pointer that will be passed to callbacks
* \param receive_callback A function pointer to the callback function
that will be called when data arrives
* \retval -1  The registration failed
* \retval 1   The registration succeeded
*/
int udp_socket_register(struct udp_socket *c,
                        void *ptr,
                    udp_socket_input_callback_treceive_callback);
```

Una vez creado y registrado el socket UDP, vamos a escuchar en un puerto dado. La función udp_socket_bind acopla el socket del UDP a un puerto local de manera que comenzará a recibir los datos que llegan a ese puerto específico. Un socket UDP recibirá los datos dirigidos al número de puerto especificado en cualquier dirección IP del host. Un socket de UDP acoplado a un puerto local usará el número de este puerto para los mensajes UDP salientes.

```
* \brief      Bind a UDP socket to a local port
* \param c    A pointer to the struct udp_socket that should be bound to
a local port
* \param local_port The UDP port number, in host byte order, to bind the
UDP socket to
* \retval -1  Binding the UDP socket to the local port failed
* \retval 1   Binding the UDP socket to the local port succeeded
*/
int udp_socket_bind(struct udp_socket *c,
                    uint16_t local_port);
```

La función udp_socket_connect conecta el socket UDP a un puerto remoto específico y a una dirección IP remota opcional. Cuando un socket UDP se conecta a un puerto y una dirección remotos recibirá sólo paquetes que sean enviados desde esa dirección y ese puerto. Cuando se mandan datos por un socket UDP conectado, los datos se enviarán a la dirección remota conectada.

Un socket UDP puede conectarse a un puerto remoto, sin una dirección IP remota en particular cuando le damos NULL como parámetro remot_addr. Esto permite que el socket del UDP reciba datos desde cualquier dirección IP en el puerto especificado.

```
/**
  * \brief      Bind a UDP socket to a remote address and port
  * \param c    A pointer to the struct udp_socket that should be
  connected
  * \param remote_addr The IP address of the remote host, or NULL if the
  UDP socket should only be connected to a specific port
  * \param remote_port The UDP port number, in host byte order, to which
  the UDP socket should be connected
  * \retval -1  Connecting the UDP socket failed
  * \retval 1   Connecting the UDP socket succeeded
  */
int udp_socket_connect(struct udp_socket *c,
                       uip_ipaddr_t *remote_addr,
                         uint16_t remote_port);
```

Para mandar datos por un socket UDP a una dirección y un puerto remotos, debemos estar conectados usando `udp_socket_connect`.

```
/**
  * \brief      Send data on a UDP socket
  * \param c    A pointer to the struct udp_socket on which the data
  should be sent
```

```
  * \param data A pointer to the data that should be sent
  * \param datalen The length of the data to be sent
  * \return     The number of bytes sent, or -1 if an error occurred
  */
int udp_socket_send(struct udp_socket *c,
                    const void *data, uint16_t datalen);
```

Para mandar datos por un socket UDP sin estar conectados, usamos, en cambio, la función udp_socket_sendto.

```
/**
     * \brief      Send data on a UDP socket to a specific address and
     port
     * \param c    A pointer to the struct udp_socket on which the data
     should be sent
     * \param data A pointer to the data that should be sent
     * \param datalen The length of the data to be sent
     * \param addr The IP address to which the data should be sent
     * \param port The UDP port number, in host byte order, to which
     the data should be sent
     * \return     The number of bytes sent, or -1 if an error occurred
     */
     int udp_socket_sendto(struct udp_socket *c,
                           const void *data, uint16_t datalen,
                           const uip_ipaddr_t *addr, uint16_t port);
```

Para cerrar un socket UDP previamente registrado con udp_socket_register se usa la función siguiente. Todos los sockets UDP deben cerrarse antes de salir del proceso que los registró. De no hacerlo, podemos obtener resultados indefinidos.

```
/**
* \brief      Close a UDP socket
* \param c    A pointer to the struct udp_socket to be closed
* \retval -1  If closing the UDP socket failed
* \retval 1   If closing the UDP socket succeeded
*/
int udp_socket_close(struct udp_socket *c);
```

Cada socket UDP tiene una función de retorno que se registra como parte de la llamada a `udp_socket_register`. La función de retorno se invoca cada vez que se recibe un paquete UDP.

```
/**
   * \brief      A UDP socket callback function
   * \param c    A pointer to the struct udp_socket that received the data
   * \param ptr  An opaque pointer that was specified when the UDP socket was
   registered with udp_socket_register()
   * \param source_addr The IP address from which the datagram was sent  *
   \param source_port The UDP port number, in host byte order, from which the
   datagram was sent
   * \param dest_addr The IP address that this datagram was sent to
   * \param dest_port The UDP port number, in host byte order, that the
   datagram was sent to
   * \param data A pointer to the data contents of the UDP datagram
   * \param datalen The length of the data being pointed to by the data pointer
   */
typedef void (* udp_socket_input_callback_t)(struct udp_socket *c,
                             void *ptr,
                           const uip_ipaddr_t*source_addr,
                                uint16_t source_port,
                                const uip_ipaddr_t *dest_addr,
                                uint16_t dest_port,
                                const uint8_t *data,
                                uint16_t datalen);
```

Como alternativa, hay otra librería UDP llamada simple-udp que simplifica la API del UDP utilizando menos funciones. La librería está en core/net/ip/simple-udp.c. Usaremos simple-udp para mostrar cómo se crea un primer ejemplo básico de broadcast. Posteriormente regresaremos al API de UDP completo.

Práctica: Ejemplo de UDP

El objetivo de este ejemplo es comprender los conceptos que se han mostrado en las secciones anteriores. Crearemos una aplicación para difusión con UDP usando simple-udp.

Hay un ejemplo de difusión con UDP usando RPL en:

```
cd examples/ipv6/simple-udp-rpl
```

Abre el broadcast-example.c y `Makefile`. Veamos el contenido de `Makefile`.

```
UIP_CONF_IPV6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
```

Lo anterior añade la pila IPv6 y el protocolo de enrutamiento RPL a nuestra aplicación.

El broadcast-example.c contiene:

```
#include "net/ip/uip.h"
```

Esta es la librería principal uIP.

```
/* Network interface and stateless autoconfiguration */
#include "net/ipv6/uip-ds6.h"

/* Use simple-udp library, at core/net/ip/ */
/* The simple-udp module provides a significantly simpler API. */
#include "simple-udp.h"
static struct simple_udp_connection broadcast_connection;
```

Esta estructura permite almacenar la información de las conexiones UDP y las retrollamadas mapeadas en las cuales se procesa cualquier mensaje recibido. Se inicializa en el siguiente llamado:

```
simple_udp_register(&broadcast_connection, UDP_PORT, NULL, UDP_PORT,
receiver);
```

Esto pasa a la aplicación **simple-udp** los puertos desde/hacia los cuales se maneja la difusión, y la función de retorno que se encarga de las difusiones recibidas. Pasamos el parámetro NULL como la dirección de destino para permitir paquetes desde cualquier dirección.

La función de retorno del receptor se muestra a continuación:

```
receiver(struct simple_udp_connection *c,
       const uip_ipaddr_t *sender_addr,
       uint16_t sender_port,
       const uip_ipaddr_t *receiver_addr,
       uint16_t receiver_port,
       const uint8_t *data,
```

```
        uint16_t datalen);
```

Esta aplicación, primero arranca un temporizador, y cuando expira, fija un nuevo intervalo para el temporizador generado al azar (entre 1 y el intervalo de envío), para evitar inundar la red. Luego establece la dirección IP para *multicast* de todos los nodos del enlace local de esta manera:

```
uip_create_linklocal_allnodes_mcast(&addr);
```

Y luego usamos la estructura broadcast_connection (con los valores pasados al registro) y enviamos nuestros datos con UDP.

```
simple_udp_sendto(&broadcast_connection, "Test", 4, &addr);
```

Para extender la información disponible sobre la dirección hay una librería que permite imprimir las direcciones IPv6 de manera mas amable. Añade esto al comienzo del archivo:

```
#include "debug.h"
#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"
```

Para poder imprimir la dirección multicast, añade lo siguiente antes del llamado simple_udp_sendto(…)

```
PRINT6ADDR(&addr);
printf("\n");
```

Ahora modifiquemos nuestra función de retorno e imprimamos más información sobre el mensaje entrante. Reemplacemos el código del receptor existente con el siguiente:

```
static void
receiver(struct simple_udp_connection *c,
        const uip_ipaddr_t *sender_addr,
        uint16_t sender_port,
        const uip_ipaddr_t *receiver_addr,
        uint16_t receiver_port,
        const uint8_t *data,
        uint16_t datalen)
 {
   /* Modified to print extended information */
   printf("\nData received from: ");
   PRINT6ADDR(sender_addr);
   printf("\nAt port %d from port %d with length %d\n",
         receiver_port, sender_port, datalen);
   printf("Data Rx: %s\n", data);
 }
```

Antes de cargar tu código, remplaza el *target* por defecto escribiendo:

```
make TARGET=remote savetarget
```

Recuerda que puedes usar la **mota Z1** como *target.*
Ahora limpia cualquier código compilado previamente, compila y carga tu propio código y luego reinicia la mota e imprime la salida serial en la pantalla (¡todo en un comando!):

```
make clean && make broadcast-example.upload && make login
```

	Carga este código en por lo menos dos motas y envía/recibe mensajes de sus vecinos. Si tienes

Dato	más de una mota conectada en tu PC, recuerda usar el argumento PORT=/dev/ttyUSBx en los comandos de carga, reinicio y registro.

Verás el siguiente resultado:

```
Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 3301
Contiki-2.6-1803-g03f57ae started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address
fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'UDP broadcast example process'
Sending broadcast to -> ff02::1

Data received from: fe80::c30c:0:0:309
At port 1234 from port 1234 with length 4
Data Rx: Test
Sending broadcast to -> ff02::1
```

Ejercicio	Apunta la ID del nodo de las otras motas. Esto puede ser útil más tarde. En este punto deberías también usar sniffer y capturar los datos con wireshark

.
Para cambiar el intervalo de envío puedes modificar los valores en:

```
#define SEND_INTERVAL  (20 * CLOCK_SECOND)
#define SEND_TIME      (random_rand() % (SEND_INTERVAL))
```

Práctica: Conecta una red UDP IPv6 a nuestro host

El archivo udp-client.c en examples/ipv6/rpl-udp. establece la dirección del servidor como aaaa::1(dirección del servidor). Reemplaza las opciones que hay (Mode 2 está por defecto), y añade:

```
uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
```

Para verificar que hemos puesto correctamente la dirección vamos a imprimir la dirección del server; en la función print_local_addresses añade esto al final:

```
PRINTF("Server address: ");
PRINT6ADDR(&server_ipaddr);
PRINTF("\n");
```

La conexión UDP se crea en el bloque siguiente:

```
/* new connection with remote host */
 client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
 if(client_conn == NULL) {
   PRINTF("No UDP connection available, exiting the process!\n");
   PROCESS_EXIT();
 }
udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

Y en cuanto se recibe un mensaje, se llama el tcpip_handler para procesar los datos entrantes:

```
static void
tcpip_handler(void)
 {
```

```
      char *str;

       if(uip_newdata()) {
        str = uip_appdata;
        str[uip_datalen()] = '\0';
        printf("DATA recv '%s'\n", str);
      }
    }
```

Compila y programa la mota:

```
cd examples/ipv6/rpl-udp
make TARGET=z1 savetarget
make udp-client.upload && make z1-reset && make login

Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 158
Contiki-2.6-2071-gc169b3e started. Node id is set to 158.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'UDP client process'
UDP client process started
Client IPv6 addresses: aaaa::c30c:0:0:9e fe80::c30c:0:0:9e
Server address: aaaa::1
Created a connection with the server :: local/remote port 8765/5678 DATA
send to 1 'Hello 1'
DATA send to 1 'Hello 2'
DATA send to 1 'Hello 3'
DATA send to 1 'Hello 4'
```

Recuerda que también puedes compilar para la plataforma **RE-Mote**.

Servidor UDP

El servidor UDP es un script python que reenvía cualquier dato entrante al cliente, útil para comprobar la comunicación bidireccional entre el host y la red.

El script UDP6.py puede ejecutarse como un cliente UDP de ejecución única (*single-shot*), o como un servidor UDP acoplado a una dirección y puerto específicos. Para este ejemplo, vamos a conectar a la dirección aaaa::1 y al puerto 5678.

A continuación, el contenido del script:

```
#! /usr/bin/env python
import sys
from socket import *
from socket import error

PORT      = 5678
BUFSIZE   = 1024

#------------------------------------------------------------#
# Start a client or server application for testing
#------------------------------------------------------------#
def main():
        if len(sys.argv) < 2:
                usage()
        if sys.argv[1] == '-s':
                server()
```

```
        elif sys.argv[1] == '-c':
                client()
        else:
                usage()
#--------------------------------------------------------------#
# Prints the instructions
#--------------------------------------------------------------#
def usage():
        sys.stdout = sys.stderr
        print 'Usage: udpecho -s [port]            (server)'
        print 'or:    udpecho -c host [port] <file (client)'
        sys.exit(2)

#--------------------------------------------------------------#
# Creates a server, echoes the message back to the client
#--------------------------------------------------------------#
def server():
     if len(sys.argv) > 2:
         port = eval(sys.argv[2])
     else:         port = PORT

      try:
         s = socket(AF_INET6, SOCK_DGRAM)
         s.bind(('aaaa::1', port))
     except Exception:
         print "ERROR: Server Port Binding Failed"
         return
     print 'udp echo server ready: %s' % port
     while 1:
         data, addr = s.recvfrom(BUFSIZE)
         print 'server received', `data`, 'from', `addr`
         s.sendto(data, addr)

#--------------------------------------------------------------#
# Creates a client that sends an UDP message to a server
#--------------------------------------------------------------#
def client():
     if len(sys.argv) < 3:
         usage()
     host = sys.argv[2]
     if len(sys.argv) > 3:
         port = eval(sys.argv[3])
     else:
         port = PORT
     addr = host, port
     s = socket(AF_INET6, SOCK_DGRAM)
     s.bind(('', 0))
         print 'udp echo client ready, reading stdin'
     try:
         s.sendto("hello", addr)
     except error as msg:
         print msg
     data, fromaddr = s.recvfrom(BUFSIZE)
     print 'client received', `data`, 'from', `fromaddr`
  #--------------------------------------------------------------#
# MAIN APP
#--------------------------------------------------------------#
main()
```

Para lanzar el script UDP6.py ejecuta:

```
python UDP6.py -s 5678:
```

Este es el resultado esperado cuando se ejecuta y recibe un paquete UDP:

```
udp echo server ready: 5678
server received 'Hello 198 from the client' from ('aaaa::c30c:0:0:9e',
8765, 0, 0)
```

El servidor entonces reenvía el mensaje al cliente UDP al puerto dado 8765. El resultado esperado de la mota es el siguiente:

```
DATA send to 1 'Hello 198'
DATA recv 'Hello 198 from the client'
```

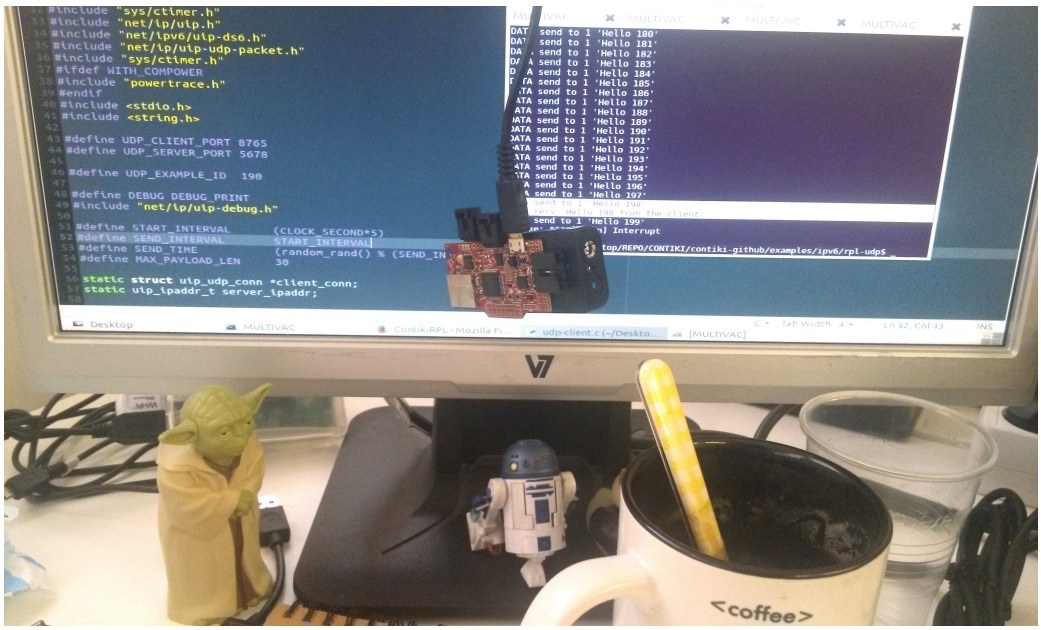

Figura 57. Comunicación UDP entre la mota Z1 y el host. Imagen propia

¿Qué es TCP?

El Protocolo de Control de Transmisión o TCP (*Transmission Control Protocol*) es un protocolo central de la pila de Protocolos de Internet (IP).

TCP es un servicio confiable de transmisión que asegura que todos los bytes recibidos estén en el orden correcto. Usa una técnica llamada reconocimiento positivo (ACK) con retransmisión para garantizar la confiabilidad de la transferencia de paquetes. TCP maneja los fragmentos recibidos y reordena los datos.

Las aplicaciones que no necesitan servicio confiable de transmisión de datos pueden usar el Protocolo de Datagramas del Usuario o UDP (User Datagram Protocol) que presta un servicio de datagrama sin conexión que privilegia la reducción de latencia sobre la confiabilidad.

TCP es comúnmente usado por http, FTP, email y otros servicios orientados a la conexión.

La implementación en Contiki se encuentra en core/net/ip. En el resto de la sección describiremos las funciones disponibles en TCP.

La API TCP

Necesitamos crear un *socket* para la conexión. Esto se hace usando la estructura tcp_socket que tiene los elementos siguientes:

```
struct tcp_socket {
   struct tcp_socket *next;

   tcp_socket_data_callback_t input_callback;
   tcp_socket_event_callback_t event_callback;
   void *ptr;

   struct process *p;

   uint8_t *input_data_ptr;
   uint8_t *output_data_ptr;

   uint16_t input_data_maxlen;
   uint16_t input_data_len;
   uint16_t output_data_maxlen;
   uint16_t output_data_len;
   uint16_t output_data_send_nxt;
   uint16_t output_senddata_len;
   uint16_t output_data_max_seg;

   uint8_t flags;
   uint16_t listen_port;
   struct uip_conn *c;
 };
```

Estatus del socket:

```
enum {
   TCP_SOCKET_FLAGS_NONE      = 0x00,
   TCP_SOCKET_FLAGS_LISTENING = 0x01,
   TCP_SOCKET_FLAGS_CLOSING   = 0x02,
 };
```

Después de crear la estructura del *socket* del TCP, tenemos que registrar el *socket*, lo que hace con `tcp_socket_register`, que toma como argumentos el *socket* del TCP, y los buffers input/output que usa para enviar y recibir datos. Asegúrate de calcular la dimensión de estos buffers de acuerdo con los datos que se espera enviar y recibir.

```
/**
  * \brief      Register a TCP socket
  * \param s    A pointer to a TCP socket
  * \param ptr  A user-defined pointer that will be sent to callbacks
for this socket
  * \param input_databuf A pointer to a memory area this socket will use
for input data
  * \param input_databuf_len The size of the input data buffer
  * \param output_databuf A pointer to a memory area this socket will
use for outgoing data
```

```
  * \param output_databuf_len The size of the output data buffer
  * \param data_callback A pointer to the data callback function for
this socket
  * \param event_callback A pointer to the event callback function for
this socket
  * \retval -1  If an error occurs
  * \retval 1   If the operation succeeds.
  */
int tcp_socket_register(struct tcp_socket *s, void *ptr,
                        uint8_t *input_databuf,int input_databuf_len,
                        uint8_t *output_databuf, int
output_databuf_len,
                        tcp_socket_data_callback_t data_callback,
                        tcp_socket_event_callback_t event_callback);
```

Ya que el *socket* del TCP ha sido creado y registrado, vamos a escuchar en un puerto determinado. Cuando un host remoto se conecta a un puerto determinado, el evento retorno (*callback*) será invocado usando el evento TCP_SOCKET_CONNECTED. Cuando la conexión se cierra, el *socket* regresa a la escucha.

```
/**
     * \brief      Start listening on a specific port
     * \param s    A pointer to a TCP socket that must have been
   previously registered with tcp_socket_register()
     * \param port The TCP port number, in host byte order, of the
   remote host
     * \retval -1  If an error occurs
     * \retval 1   If the operation succeeds.
     */
   int tcp_socket_listen(struct tcp_socket *s,
                         uint16_t port);
```

Para dejar de escuchar en un puerto TCP dado, invoca la siguiente función:

```
/**
  * \brief      Stop listening for new connections
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \retval -1  If an error occurs
  * \retval 1   If the operation succeeds.
  */
int tcp_socket_unlisten(struct tcp_socket *s);
```

Podemos conectar el *socket* del TCP a un host remoto. Cuando el *socket* se ha conectado, la función de retorno será invocada con el evento TCP_SOCKET_CONNECTED. Si el host remoto no acepta la conexión, se envía TCP_SOCKET_ABORTED a la función de retorno. Si el tiempo de la conexión expira antes de conectarse al host remoto, el evento TCP_SOCKET_TIMEDOUT es enviado a la función de retorno.

```
/**
  * \brief      Connect a TCP socket to a remote host
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \param ipaddr The IP address of the remote host
  * \param port The TCP port number, in host byte order, of the remote
host
  * \retval -1  If an error occurs
  * \retval 1   If the operation succeeds.
  */
int tcp_socket_connect(struct tcp_socket *s,
                       const uip_ipaddr_t *ipaddr,
                       uint16_t port);
```

Como estamos usando un buffer en la salida para enviar datos por el *socket* del TCP, sería una buena práctica consultar el *socket* del TCP y chequear el número de bytes disponibles.

```
/**
  * \brief      The maximum amount of data that could currently be sent
  * \param s    A pointer to a TCP socket
  * \return     The number of bytes available in the output buffer
  */
int tcp_socket_max_sendlen(struct tcp_socket *s);
```

Para enviar datos por un *socket* del TCP conectado, estos se colocan en el output buffer. Cuando los datos han sido reconocidos por el host remoto, se invoca la función de retorno con el evento TCP_SOCKET_DATA_SENT.

```
/**
  * \brief      Send data on a connected TCP socket
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \param dataptr A pointer to the data to be sent
  * \param datalen The length of the data to be sent
  * \retval -1  If an error occurs
  * \return     The number of bytes that were successfully sent
  */
int tcp_socket_send(struct tcp_socket *s,
                    const uint8_t *dataptr,
                    int datalen);
```

Alternativamente, podemos mandar una cadena por un *socket* del TCP de esta manera:

```
/**
  * \brief      Send a string on a connected TCP socket
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \param strptr A pointer to the string to be sent
  * \retval -1  If an error occurs
  * \return     The number of bytes that were successfully sent
  */
int tcp_socket_send_str(struct tcp_socket *s,
                        const char *strptr);
```

Para cerrar un *socket* de TCP es usa la función que sigue. La función de retorno es llamada con el evento TCP_SOCKET_CLOSED.

```
/**
* \brief      Close a connected TCP socket
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \retval -1  If an error occurs
  * \retval 1   If the operation succeeds.
  */ int tcp_socket_close(struct tcp_socket *s);
```

Y para desincorporar un *socket* TCP se usa la función tpc_socket_unregister. Esta también puede usarse para reiniciar un *socket* TCP conectado.

```
/**
  * \brief      Unregister a registered socket
  * \param s    A pointer to a TCP socket that must have been previously
registered with tcp_socket_register()
  * \retval -1  If an error occurs
  * \retval 1   If the operation succeeds.
  *
  *             This function unregisters a previously registered
  *             socket. This must be done if the process will be
  *             unloaded from memory. If the TCP socket is connected,
  *             the connection will be reset.
  *
*/
int tcp_socket_unregister(struct tcp_socket *s);
```

La función de retorno del *socket* TCP es invocada cada vez que hay un evento en un *socket*, como cuando este se cierra o se conecta.

```
/**
  * \brief      TCP event callback function
  * \param s    A pointer to a TCP socket
  * \param ptr  A user-defined pointer
  * \param event The event number
  */
typedef void (* tcp_socket_event_callback_t)(struct tcp_socket *s,
                                             void *ptr,
                                            tcp_socket_event_event);
```

La función de retorno de datos del TCP tiene que añadirse a la aplicación. Será invocada cada vez que haya datos nuevos en el *socket*.

```
/**
  * \brief      TCP data callback function
  * \param s    A pointer to a TCP socket
  * \param ptr  A user-defined pointer
  * \param input_data_ptr A pointer to the incoming data
  * \param input_data_len The length of the incoming data
  * \return     The function should return the number of bytes to leave in
the input buffer
  */
typedef int (* tcp_socket_data_callback_t)(struct tcp_socket *s,
                                           void *ptr,
                                           const uint8_t *input_data_ptr,

                                           int input_data_len);
```

Práctica: ejemplo con TCP

Ahora pongamos en práctica la aplicación TCP IP antes descrita. El ejemplo tcp-socket está en examples/tcp-socket. El servidor TCP simplemente reenvía por el puerto 80 la solicitud recibida.

El `Makefile` habilita por defecto la pila IPv4. Cámbiala a IPv6.

```
UIP_CONF_IPV6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
```

Abramos el ejemplo tcp-server.c y exploremos la implementación:

El puerto 80 es usado por el servidor TPC para recibir conexiones remotas. Como se mostró antes, tenemos que crear una estructura tcp_socket, y usar dos buffers input/output separados para el envío y recepción de datos.

```
#define SERVER_PORT 80

static struct tcp_socket socket;

#define INPUTBUFSIZE 400
static uint8_t inputbuf[INPUTBUFSIZE];

#define OUTPUTBUFSIZE 400
static uint8_t outputbuf[OUTPUTBUFSIZE];
```

Estas dos variables se usarán para contar el número de bytes recibidos y los que se enviarán.

```
static uint8_t get_received;
static int bytes_to_send;
```

Como hemos hecho antes, necesitaremos incluir un tcp_socket_event_callback_t para manejar los eventos.

```
static void
event(struct tcp_socket *s, void *ptr,
       tcp_socket_event_t ev)
 {
   printf("event %d\n", ev);
 }
```

Registramos el socket TCP y pasamos como puntero la estructura `tcp_socket`, los buffers de los datos y los punteros a nuestras funciones de retorno.

Ahora, comencemos a oír las conexiones en el puerto 80.

```
tcp_socket_register(&socket, NULL,
                inputbuf, sizeof(inputbuf),
                outputbuf, sizeof(outputbuf),
                input, event);

tcp_socket_listen(&socket, SERVER_PORT);
```

El manejador de la función de retorno *input* recibe los datos, imprime la cadena y su longitud, y luego, si la cadena recibida es una solicitud completa, almacena el número de bytes recibidos en bytes_to_send (la función 'atoi' convierte cadenas de números en números enteros). Si la cadena recibida no está completa, devolvemos el número de bytes recibido al driver para mantener los datos en el buffer de entrada.

```
static int
input(struct tcp_socket *s, void *ptr,
       const uint8_t *inputptr, int inputdatalen)
 {
   printf("input %d bytes '%s'\n", inputdatalen, inputptr);
   if(!get_received) {
     /* See if we have a full GET request in the buffer. */
     if(strncmp((char *)inputptr, "GET /", 5) == 0 &&
        atoi((char *)&inputptr[5]) != 0) {
```

```
        bytes_to_send = atoi((char *)&inputptr[5]);
        printf("bytes_to_send %d\n", bytes_to_send);
        return 0;
      }
      printf("inputptr '%.*s'\n", inputdatalen, inputptr);
      /* Return the number of data bytes we received, to keep them all in
the buffer. */
      return inputdatalen;
    } else {
      /* Discard everything */
      return 0; /* all data consumed */
    }
 }
```

La aplicación esperará que ocurra un evento, en este caso la conexión entrante. Después de que el evento es manejado, se ejecutará el código que está dentro del bucle **while()** y después del PROCESS_PAUSE().

```
while(1) {
   PROCESS_PAUSE();
```

Si hemos recibido previamente una solicitud completa, la reenviamos por el *socket* TCP. Usamos la función tcp_socket_send_str para mandar la cabecera de la respuesta como una cadena. Los datos restantes se enviarán hasta que el contador bytes_to_send se vacíe.

```
if(bytes_to_send > 0) {
       /* Send header */
       printf("sending header\n");
       tcp_socket_send_str(&socket, "HTTP/1.0 200 ok\r\nServer: Contiki
tcp-socket example\r\n\r\n");

        /* Send data */
       printf("sending data\n");
       while(bytes_to_send > 0) {
         PROCESS_PAUSE();
         int len, tosend;
         tosend = MIN(bytes_to_send, sizeof(outputbuf));
         len = tcp_socket_send(&socket, (uint8_t *)"", tosend);
         bytes_to_send -= len;
       }
       tcp_socket_close(&socket);

     }
   }
   PROCESS_END();
 }
```

Cuando todos los datos han sido reenviados, se cierra el socket TCP.

CoAP, MQTT y Ubidots

En secciones anteriores se tocaron los temas de transmisión inalámbrica y los conceptos generales de Contiki. Esta sección presenta dos protocolos usados extensivamente en la IoT: CoAP y MQTT. Explicaremos sus bases y cerraremos con ejemplos de uso inmediato. También realizaremos un ejemplo utilizando una plataforma IoT de datos (Ubidots), conectándonos así a Internet con una aplicación real.

Ejemplo en CoAP

La implementación de CoAP en Contiki se basa en Erbium (Er), un motor REST de baja potencia para Contiki. El motor REST incluye una implementación de CoAP completa incrustada que es la oficial de Contiki.

Más información sobre su implementación y su autor en la página web del proyecto Erbium
http://people.inf.ethz.ch/mkovatsc/erbium.php

¿Qué son REST y CoAP?

La **Transferencia de Estado Representacional** o **REST** (*Representational State Transfer*) se basa en un protocolo de comunicaciones cliente-servidor sin estado y almacenable en la memoria cache. En casi todos los casos , utiliza HTTP.

La abstracción clave de un servicio web *RESTful* (basado en REST) es el recurso, no el servicio. Los sensores, los actuadores y los sistemas de control, en general pueden representarse de manera elegante como recursos mostrados a través de un servicio web RESTful.

Las aplicaciones RESTful usan solicitudes de tipo HTTP para publicar datos (crear y/o actualizar), leer datos (por ejemplo, hacer preguntas) y borrar datos. Por consiguiente, REST usa HTTP para todo las operaciones CRUD, es decir Crear/Leer/Actualizar/Borrar (*Create/Read/Update/Delete*).

A pesar de ser sencillo, REST es muy completo: prácticamente todo lo que se hace en servicios web puede hacerse con una arquitectura RESTful (REST no es un estándar).

El **Protocolo de Aplicación Restringida, CoAP** (*Constrained Application Protocol*), es un protocolo de software para usar en dispositivos electrónicos muy sencillos que les permite una comunicación interactiva en Internet. En particular, está dirigido a sensores de baja potencia, interruptores, válvulas y componentes similares que deben ser controlados o supervisados de forma remota, a través de redes Internet estándar. CoAP es un protocolo de capa de aplicación destinado para usarse en dispositivos de Internet con recursos limitados, tales como nodos WSN. CoAP está diseñado para fácil traducción a HTTP para su integración en la web, cumple con los requisitos especializados como soporte multicast, muy bajo costo operativo, y simplicidad.

CoAP es compatible con la mayoría de los dispositivos que pueden usar UDP. CoAP usa dos tipos de mensajes, solicitudes y respuestas, con un sencillo formato de cabecera de base. La cabecera básica puede estar seguida por opciones en un formato optimizado Type-Length-Value. Por defecto, CoAP usa UDP y opcionalmente DTLS cuando se requiere un alto nivel de seguridad en las comunicaciones.

Los octetos después de las cabeceras del paquete (si están presentes) son tomados como el cuerpo del mensaje. La longitud del cuerpo del mensaje está implícita en la longitud del datagrama. Cuando usa UDP todo el mensaje debe caber dentro de un solo datagrama. Cuando se utiliza con 6LoWPAN, como se define en el RFC 4944, los mensajes deben caber en una sola trama IEEE 802.15.4 para minimizar la fragmentación.

La API de CoAP

La implementación CoAP en Contiki está ubicada en `apps/er-coap`.
El motor Erbium REST está implementado en `apps/rest-engine`

El motor CoAP (actualmente CoAP-18) está implementado en `er-coap-engine.c`
Su interfaz tiene la estructura siguiente:

```
const struct rest_implementation coap_rest_implementation = {
   coap_init_engine,
   coap_set_service_callback,
   coap_get_header_uri_path,
 (...)
 }
```

CoAP se invoca con:

```
REST.get_query_variable();
```

Los servicios web están considerados como recursos (*resources*) y se identifican distintivamente por sus URL. El diseño básico de REST usa los métodos del protocolo HTTP o CoAP para las operaciones normales (crear, leer, actualizar, borrar):

- POST: crea un recurso
- GET: Recupera un recurso
- PUT: actualiza un recurso
- DELETE : Borra un recurso

Hay varios recursos disponibles en el servidor. Cada recurso tiene una función de manejador que es llamada por REST para pasar la solicitud del cliente. El servidor REST envía de vuelta una respuesta al cliente con los contenidos del recurso solicitado.

Las siguientes macros están disponibles en `apps/rest-engine`, y se recomiendan cuando creamos un recurso nuevo CoAP.

Un **recurso normal** se define por un Uri-Path estático que está asociado con una función de manejador de recursos. Este es la base de otros tipos de recursos.

```
#define RESOURCE(name, attributes, get_handler, post_handler, put_handler,
delete_handler) \
   resource_t name = { NULL, NULL, NO_FLAGS, attributes, get_handler,
post_handler, put_handler, delete_handler, { NULL } }
```

Un **recurso parental** maneja varios sub-recursos por evaluación del Uri-Path, que puede ser más largo que el recurso parental.

```
#define PARENT_RESOURCE(name, attributes, get_handler, post_handler,
put_handler, delete_handler) \
   resource_t name = { NULL, NULL, HAS_SUB_RESOURCES, attributes, get_handler,
post_handler, put_handler, delete_handler, { NULL } }
```

Si el servidor no es capaz de responder inmediatamente a una solicitud `CON` , simplemente responderá con un mensaje Empty ACK de manera tal que el cliente detenga el proceso de re-transmisión de la solicitud. La siguiente macro permite crear un recurso CoAP con **respuesta separada** (*separate response*):

```
#define SEPARATE_ (name, attributes, get_handler, post_handler, put_handler,
delete_handler, resume_handler) \   resource_t name = { NULL, NULL,
IS_SEPARATE, attributes, get_handler, post_handler, put_handler,
delete_handler, { .resume = resume_handler } }
```

Un **recurso de evento** es semejante a un recurso periódico, la única diferencia es que el segundo manejador es llamado por un evento asíncrono como el oprimir un botón.

```
#define EVENT_RESOURCE(name, attributes, get_handler, post_handler,
put_handler, delete_handler, event_handler) \
   resource_t name = { NULL, NULL, IS_OBSERVABLE, attributes, get_handler,
post_handler, put_handler, delete_handler, { .trigger = event_handler } }
```

Si necesitamos declarar un **recurso periódico**, por ejemplo, sondear un sensor y publicar un cambio en los valores en los clientes suscritos, deberíamos usar:

```
#define PERIODIC_RESOURCE(name, attributes, get_handler, post_handler,
put_handler, delete_handler, period, periodic_handler) \
   periodic_resource_t periodic_##name; \
   resource_t name = { NULL, NULL, IS_OBSERVABLE | IS_PERIODIC, attributes,
get_handler, post_handler, put_handler, delete_handler, { .periodic =
&periodic_##name } }; \
   periodic_resource_t periodic_##name = { NULL, &name, period, { { 0 } },
periodic_handler };
```

Nótese que `PERIODIC_RESOURCE` y `EVENT_RESOURCE` pueden ser observables, lo que significa que se le puede notificar a un cliente sobre cualquier cambio en un recurso dado.

Una vez que declaremos e implementemos los recursos (lo que haremos en la sección práctica), necesitamos inicializar el marco REST y comenzar el proceso HTTP o CoAP. Esto se hace usando:

```
void rest_init_engine(void);
```

Luego, para cada recurso declarado que necesitamos que esté accesible, declararemos:

```
void rest_activate_resource(resource_t *resource, char *path);
```

Supongamos que hemos creado un recurso `hello-world` en `res-hello.c` declarado así:

```
RESOURCE(res_hello,
         "title=\"Hello world: ?len=0..\";rt=\"Text\"",
         res_get_handler,
         NULL,
         NULL,
         NULL);
```

Para activar el recursos haríamos:

```
rest_activate_resource(&res_hello, "test/hello");
```

Lo que significa que el recurso va a estar disponible en el uri-Path `test/hello`
La anterior función almacena los recursos en una lista. Para tener la lista de recursos disponibles se usa la función `rest_get_resources`. Esta nos entregará la lista de recursos usando:

```
rest_get_resources();
```

Recuerda que el puerto obligatorio para CoAP es el `5683`.
Ahora juntemos todos estos conceptos en un ejemplo práctico.

Práctica: servidor CoAP y Copper

En primer lugar, tomemos el usuario-agente CoAP Copper (Cu) de:

https://addons.mozilla.org/en-US/firefox/addon/copper-270430/

Copper es un navegador genérico para la Internet de las Cosas (IoT) basado en el Protocolo de Aplicación Restringida (CoAP). Es una herramienta amigable de manejo para dispositivos integrados en red. Como está incorporada en los navegadores web permite una interacción intuitiva con la capa de presentación por lo que hace fácil la tarea de depurar otros dispositivos CoAP existentes.

Más información en:

http://people.inf.ethz.ch/mkovatsc/copper.php

Para esta práctica usaremos dos motas: un enrutador de frontera y un servidor CoAP.

Advertencia	Si estás usando **motas Z1**, para chequear esto (enrutador de frontera, cliente y servidor), asegúrate de que hayan sido reprogramadas con una ID de Nodo para generar las direcciones MAC/IPv6, como se explicó en secciones anteriores. ¡No te olvides de tomar nota de las direcciones! Otra cosa: si obtienes un error como el siguiente ve a `platform/z1/contiki-conf.h` y cambia `UIP_CONF_BUFFER_SIZE` a 240:

```
#error "UIP_CONF_BUFFER_SIZE too small for REST_MAX_CHUNK_SIZE"
make: *** [obj_z1/er-coap-07-engine.o] Error 1
```

En `Makefile` podemos notar dos cosas: la carpeta `resources` está incluida como un directorio de proyectos, y todos los archivos de recursos se añaden a la compilación.

```
REST_RESOURCES_DIR = ./resources
REST_RESOURCES_FILES = $(notdir $(shell find $(REST_RESOURCES_DIR) -name '*.c' !
-name 'res-plugtest*'))
PROJECTDIRS += $(REST_RESOURCES_DIR)
PROJECT_SOURCEFILES += $(REST_RESOURCES_FILES)
```

Además incluimos las aplicaciones `er-coap` y `rest-engine`

```
# REST Engine shall use Erbium CoAP implementation
APPS += er-coap
APPS += rest-engine
```

Elimina lo siguiente, ya que queremos evitar en lo posible las colisiones:

```
#undef NETSTACK_CONF_MAC
#define NETSTACK_CONF_MAC     nullmac_driver
```

Ahora vamos a examinar la configuración relevante `project-conf.h`. Primero asegurémonos de que TCP esté deshabilitado, ya que CoAP se basa en UDP.

```
/* Disabling TCP on CoAP nodes. */
 #undef UIP_CONF_TCP
 #define UIP_CONF_TCP                    0
```

El `REST_MAX_CHUNK_SIZE` es el tamaño máximo de buffer que se tiene para repuestas de recursos. Si la cantidad de datos es grande deberían ser manejados por el recurso y enviarse en bloques CoAP. El `COAP_MAX_OPEN_TRANSACTIONS` es el número máximo de transacciones abiertas que el nodo es capaz de manejar.

```
/* Increase rpl-border-router IP-buffer when using more than 64. */
 #undef REST_MAX_CHUNK_SIZE
 #define REST_MAX_CHUNK_SIZE             48

/* Multiplies with chunk size, be aware of memory constraints. */
 #undef COAP_MAX_OPEN_TRANSACTIONS
 #define COAP_MAX_OPEN_TRANSACTIONS      4

/* Filtering .well-known/core per query can be disabled to save space. */
 #undef COAP_LINK_FORMAT_FILTERING
 #define COAP_LINK_FORMAT_FILTERING      0
 #undef COAP_PROXY_OPTION_PROCESSING
 #define COAP_PROXY_OPTION_PROCESSING    0

/* Enable client-side support for COAP observe */
 #define COAP_OBSERVE_CLIENT 1
```

Servidor CoAP

Vamos a explorar el ejemplo `er-example-server.c` y a entender su implementación. La primera cosa que notamos es una carpeta que se llama *resources* en la carpeta de ejemplo, porque los

recursos van a estar implementados en un archivo diferente. Esto hace que sea más fácil depurar y mantener los ejemplos.

Los recursos que se incluirán en el servidor CoAP se definen en la siguiente declaración:

```
extern resource_t
   res_hello,
   res_mirror,
   res_chunks,
   res_separate,
   res_push,
   res_event,
   res_sub,
   res_b1_sep_b2;
 #if PLATFORM_HAS_LEDS
 extern resource_t res_leds, res_toggle;
 #endif
 #if PLATFORM_HAS_BATTERY
 #include "dev/battery-sensor.h"
 extern resource_t res_battery;
 #endif
 #if PLATFORM_HAS_RADIO
 #include "dev/radio-sensor.h"
 extern resource_t res_radio;
 #endif
```

Los recursos envueltos en `PLATFORM_HAS_X` son dependientes de la plataforma y serán incorporado si la plataforma los ha habilitado.

Luego el motor REST se inicia con el llamado `rest_init_engine()`, y los recursos habilitados se vinculan:

```
/* Initialize the REST engine. */
 rest_init_engine();

  /*
  * Bind the  s to their Uri-Path.
  * WARNING: Activating twice only means alternate path, not two instances!
  * All static variables are the same for each URI path.
  */
 rest_activate_resource(&res_hello, "test/hello");
 rest_activate_resource(&res_push, "test/push");
 rest_activate_resource(&res_event, "sensors/button"); */
 #if PLATFORM_HAS_LEDS
   rest_activate_resource(&res_toggle, "actuators/toggle");
 #endif
(...)
```

Ahora veamos el recurso `res-hello.c` que implementa un recurso “hello world”.

Como se mostró antes, los recursos se definen usando la macro `RESOURCE`. Para esta implementación particular especificamos como nombre `res_hello`, los atributos formateados del enlace y el manejador de funciones de retorno `GET`. Los métodos POST, PUT Y DELETE no son compatibles con este recurso, así que como argumento se usa el parámetro NULL.

```
RESOURCE(res_hello,
          "title=\"Hello world: ?len=0..\";rt=\"Text\"",
          res_get_handler,
```

```
            NULL,
            NULL,
            NULL);
```

El `res_get_handler` es el evento de retorno para las solicitudes `GET`. Su implementación es la siguiente:

```
static void
 res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t
preferred_size, int32_t *offset)
 {
   const char *len = NULL;
   /* Some data that has the length up to REST_MAX_CHUNK_SIZE. For more, see the
chunk resource. */
   char const *const message = "Hello World!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy";
   int length = 12;                                                    (1)

/* The query string can be retrieved by rest_get_query() or parsed for its key-
value pairs. */
   if(REST.get_query_variable(request, "len", &len)) {
                      (2)
     length = atoi(len);
    if(length < 0) {
                                                        (3)
     length = 0;
    }
    if(length > REST_MAX_CHUNK_SIZE) {
                                     (4)
      length = REST_MAX_CHUNK_SIZE;
    }
    memcpy(buffer, message, length);
   } else {
    memcpy(buffer, message, length);
                                      (5)
/* text/plain is the default, hence this option could be omitted. */
   } REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
        (6)
    REST.set_header_etag(response, (uint8_t *)&length, 1);
                  (7)
REST.set_response_payload(response, buffer, length);
                    (8)
 }
```

1) La longitud por defecto de la respuesta. En este caso, de la cadena completa se enviará solamente Hello World!

2) Si se especifica la opción **len** , se enviará un número de bytes len de la cadena.

3) Si el valor es negativo, manda una cadena vacía.

4) Si **len** es mayor que lo máximo permitido, enviamos sólo el valor de la longitud máxima permitida .

5) Copia lo que está por defecto

6) Especifica el tipo de respuesta de contenido como Content-Type:text/plain

7) Añade la cabecera a la respuesta; especifica la longitud del campo carga útil (payload).

8) Añade la carga útil a la respuesta

Advertencia	Asegúrate de que los ajustes sean coherentes con los del enrutador de frontera

Para los propósitos de este test, añade lo siguiente al archivo `project-conf.h`

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC     nullrdc_driver
```

Luego compila y carga:

```
cd examples/er-rest-example/
make TARGET=remote savetarget
make er-example-server.upload && make login
```

Anota la dirección IPv6 del servidor, desconecta la mota y conecta otra para usarla como cliente.

Desconecta la mota, conecta otra para usarla como enrutador de frontera.

Enrutador de frontera

```
cd ../ipv6/rpl-border-router/
make TARGET=z1 savetarget
make border-router.upload && make connect-router
```

¡No cierres esta ventana! Deja la mota conectada. Ahora verás algo como lo que sigue:

```
SLIP started on ``/dev/ttyUSB0''
opened tun device ``/dev/tun0''
ifconfig tun0 inet `hostname` up
ifconfig tun0 add aaaa::1/64
ifconfig tun0 add fe80::0:0:0:1/64 ifconfig tun0

tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-
00-00
      inet addr:127.0.1.1  P-t-P:127.0.1.1  Mask:255.255.255.255
      inet6 addr: fe80::1/64 Scope:Link
      inet6 addr: aaaa::1/64 Scope:Global
      UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:500
      RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

Rime started with address 193.12.0.0.0.0.3.229
MAC c1:0c:00:00:00:00:03:e5 Contiki-2.5-release-681-gc5e9d68 started. Node id is
set to 997.
CSMA nullrdc, channel check rate 128 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:03e5
Starting 'Border router process' 'Web server'
Address:aaaa::1 => aaaa:0000:0000:0000
Got configuration message of type P
Setting prefix aaaa::
Server IPv6 addresses:
```

```
aaaa::c30c:0:0:3e5
fe80::c30c:0:0:3e5
```

Hagamos un *ping* al enrutador de frontera:

```
ping6 aaaa:0000:0000:0000:c30c:0000:0000:03e5
PING aaaa:0000:0000:0000:c30c:0000:0000:03e5(aaaa::c30c:0:0:3e5) 56 data bytes
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=1 ttl=64 time=21.0 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=2 ttl=64 time=19.8 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=3 ttl=64 time=22.2 ms
64 bytes from aaaa::c30c:0:0:3e5: icmp_seq=4 ttl=64 time=20.7 ms
```

Conectemos la mota del servidor y hagamos *ping*:

```
ping6 aaaa:0000:0000:0000:c30c:0000:0000:0001
PING aaaa:0000:0000:0000:c30c:0000:0000:0001(aaaa::c30c:0:0:1) 56 data bytes
64 bytes from aaaa::c30c:0:0:1: icmp_seq=1 ttl=63 time=40.3 ms
64 bytes from aaaa::c30c:0:0:1: icmp_seq=2 ttl=63 time=34.2 ms
64 bytes from aaaa::c30c:0:0:1: icmp_seq=3 ttl=63 time=35.7 ms
```

Empezamos a descubrir los recursos del servidor. Abre Firefox y escribe la dirección del servidor:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/
```

Ahora podemos comenzar a descubrir los recursos disponibles. Presiona en `DISCOVER` y se te poblará el lado izquierdo de la página.

Si seleccionas el `toggle` y usas `POST` puedes observar cómo el LED rojo de la mota servidor va a destellar.

Si haces lo mismo con el resource `Hello` el servidor te responderá con el consabido mensaje.

Finalmente, si seleccionas el botón `Sensors→ Button` y haces clic en `OBSERVE`,
cada vez que presionas el botón del usuario, se accionará un evento y te será reportado.

Por último, si vas al archivo `er-example-server.c` y habilitas las siguientes definiciones, tendrás a tu disposición más recursos:

```
#define REST_RES_HELLO    1
#define REST_RES_SEPARATE 1
#define REST_RES_PUSHING  1
#define REST_RES_EVENT    1
#define REST_RES_SUB      1
#define REST_RES_LEDS     1
#define REST_RES_TOGGLE   1
#define REST_RES_BATTERY  1
#define REST_RES_RADIO    1
```

Y para obtener el nivel actual de RSSI en el transceptor:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/sensor/radio?p=rssi
```

Haz lo mismo para ver el nivel de la batería:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/sensors/battery
```

Este último caso te da las unidades ADC. Cuando la mota está conectada al USB. El valor en milivoltios debería ser:

```
voltaje [mV] = (units * 5000)/4096
```

Digamos que quieres encender el LED verde. En la URL escribe:

```
coap://[aaaa::c30c:0000:0000:0001]:5683/actuators/leds?color=g
```

Y luego en la carga útil, escribe:

```
mode="on"
```

Presiona POST o PUT (mueve el ratón sobre los actuadores/LED para ver la descripción y los métodos permitidos).

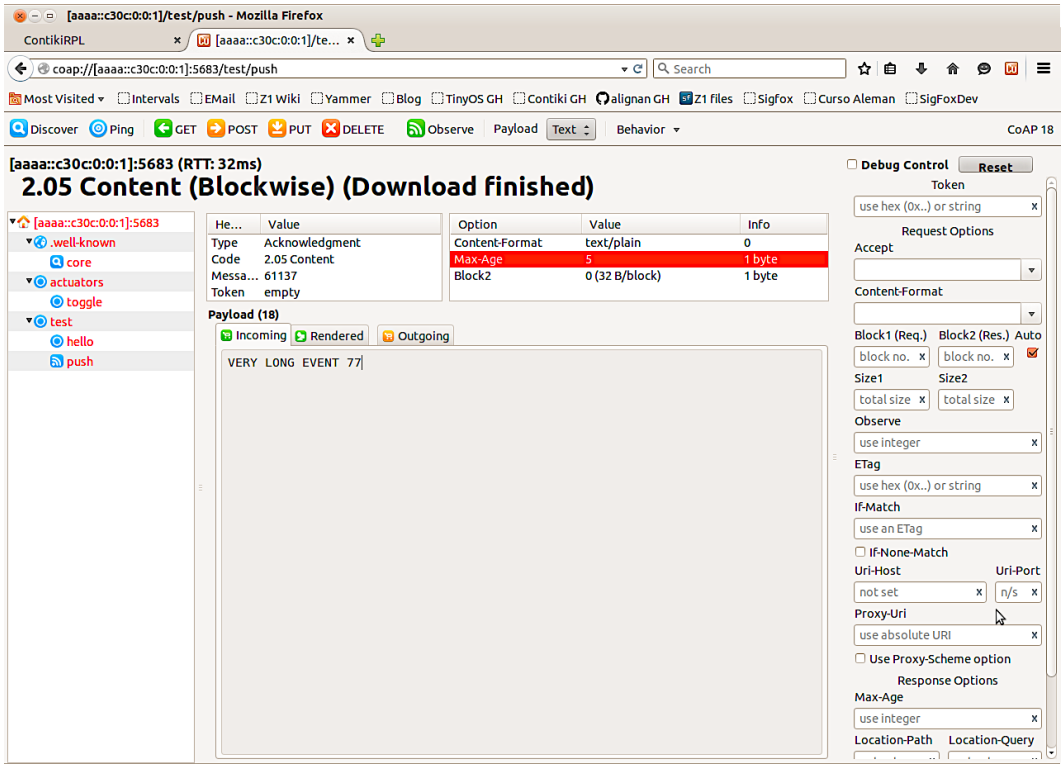

Figura 58. Plugin de Copper (CoAP). Imagen propia.

Ejemplo con MQTT

¿Qué es MQTT?

MQTT (anteriormente *MQ Telemetry Transport*) es un protocolo de mensajería basado en publicación-suscripción (*publish-subscribe*) ubicado encima del protocolo TCP/IP. Está diseñado

para conexiones con lugares remotos donde se requiere un una implementación mínima de código y tamaño, o donde el ancho de banda es limitado.

El patrón de publicación-suscripción de mensajería necesita un *broker* de mensajes. Este es responsable de distribuir los mensajes a los clientes interesados basándose en el tópico de un mensaje.

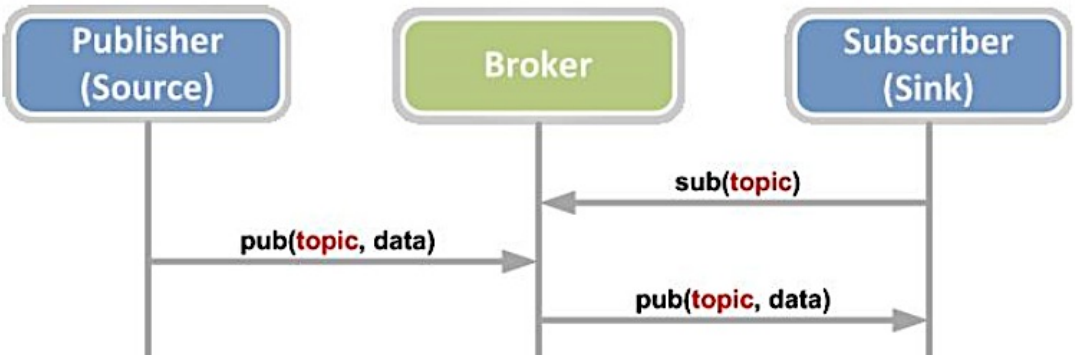

Figura 59. MQTT publish/subscribe protocol. Imagen tomada de:
http://www.eurotech-inc.com/mqtt-protocol-for-data-delivery.asp

MQTT ha definido tres niveles de Calidad de Servicio (QoS):

- QoS 0: El *broker*/cliente enviará el mensaje solamente una vez, sin confirmación (envía y olvida).
- QoS 1: El *broker*/cliente enviará el mensaje por lo menos una vez, con solicitud de confirmación.
- QoS 3: El *broker*/cliente enviará el mensaje exactamente una vez usando un *handshake* de cuatro pasos.

Otros rasgos de MQTT son:

- Mensaje de *keep-alive* (PINGREQ, PINGRESP).
- Un *broker* puede detectar la desconexión de un cliente incluso sin haber un mensaje explícito de `DISCONNECT` .
- Mensaje de "*Last Will and Testament* " ("Última Voluntad y Testamento")*:* especificado en `CONNECT` mensaje con tópico, QoS y *retain*. Cuando ocurre una desconexión inesperada del cliente, el mensaje "*Last Will and Testament"* se envía a los clientes suscritos.
- Mensaje de "*retain"* ("retención") : un mensaje `PUBLISH` sobre un tópico se mantiene en el *broker*, lo que le permite a un nuevo suscriptor conectado en el mismo tópico recibir este mensaje (último mensaje bueno conocido).
- Suscripción "durable": cuando se desconecta el cliente, todas las suscripciones se mantienen en el *broker* y se recuperan con la reconexión del cliente.

MQTT reserva los puertos TCP/IP 1883 y 8883 . Este último para usar MQTT sobre SSL.

Para información más amplia , ver http://mqtt.org/

API de MQTT

La implementación de MQTT en Contiki está en `apps/mqtt` Usa la librería `tcp-socket` que se discutió en secciones anteriores.

La versión actual de MQTT implementada en Contiki admite QoS de niveles 0 y 1.
Las funciones disponibles de MQTT se describen a continuación. Un ejemplo práctico nos ayudará a esclarecer su uso utilizando una máquina de estado para mantener el estado de la conexión y la operación del dispositivo.

La siguiente función inicia el motor MQTT y debe ser invocada antes de cualquier otra:

```
/**
  * \brief Initializes the MQTT engine.
  * \param conn A pointer to the MQTT connection.
  * \param app_process A pointer to the application process handling the MQTT
*        connection.
  * \param client_id A pointer to the MQTT client ID.
  * \param event_callback Callback function responsible for handling the  *
callback from MQTT engine.
  * \param max_segment_size The TCP segment size to use for this MQTT/TCP  *
connection.
  * \return MQTT_STATUS_OK or MQTT_STATUS_INVALID_ARGS_ERROR
  */
mqtt_status_t mqtt_register(struct mqtt_connection *conn,
                            struct process *app_process,
                            char *client_id,
                            mqtt_event_callback_t event_callback,
                            uint16_t max_segment_size);
```

Esta función nos conecta a un *broker* MQTT:

```
/**
  * \brief Connects to a MQTT broker.
  * \param conn A pointer to the MQTT connection.
  * \param host IP address of the broker to connect to.
  * \param port Port of the broker to connect to, default is MQTT port is 1883.
  * \param keep_alive Keep alive timer in seconds. Used by broker to handle
  *        client disc. Defines the maximum time interval between two messages
  *        from the client. Shall be min 1.5 x report interval.
  * \return MQTT_STATUS_OK or an error status
  */
mqtt_status_t mqtt_connect(struct mqtt_connection *conn,
                           char *host,
                           uint16_t port,
                           uint16_t keep_alive);
```

Esta nos desconecta de un *broker* MQTT:

```
/**
  * \brief Disconnects from a MQTT broker.
  * \param conn A pointer to the MQTT connection.
  */ void mqtt_disconnect(struct mqtt_connection *conn);
```

Esta función nos suscribe a un tópico en un *broker* MQTT:

```
/**
  * \brief Subscribes to a MQTT topic.
  * \param conn A pointer to the MQTT connection.
  * \param mid A pointer to message ID.  * \param topic A pointer to the
topic to subscribe to.
  * \param qos_level Quality Of Service level to use. Currently supports 0,
1.
  * \return MQTT_STATUS_OK or some error status
  */
mqtt_status_t mqtt_subscribe(struct mqtt_connection *conn,
                             uint16_t *mid,
                             char *topic,
                             mqtt_qos_level_t qos_level);
```

Esta función nos da de baja (*unsuscribe*) de un tópico en un *broker* MQTT:

```
/**
  * \brief Unsubscribes from a MQTT topic.
  * \param conn A pointer to the MQTT connection.
  * \param mid A pointer to message ID.
  * \param topic A pointer to the topic to unsubscribe from.
  * \return MQTT_STATUS_OK or some error status
  */
mqtt_status_t mqtt_unsubscribe(struct mqtt_connection *conn,
                               uint16_t *mid,
                               char *topic);
```

Esta otra función publica a un tópico en un *broker* MQTT:

```
/**
  * \brief Publish to a MQTT topic.
  * \param conn A pointer to the MQTT connection.
  * \param mid A pointer to message ID.
  * \param topic A pointer to the topic to subscribe to.
  * \param payload A pointer to the topic payload.
  * \param payload_size Payload size.
  * \param qos_level Quality Of Service level to use. Currently supports 0, 1.
  * \param retain If the RETAIN flag is set to 1, in a PUBLISH Packet sent by
a
  *        Client to a Server, the Server MUST store the Application Message
  *        and its QoS, so that it can be delivered to future subscribers
whose
  *        subscriptions match its topic name
  * \return MQTT_STATUS_OK or some error status
  */
mqtt_status_t mqtt_publish(struct mqtt_connection *conn,
                           uint16_t *mid,
                           char *topic,
                           uint8_t *payload,
                           uint32_t payload_size,
                           mqtt_qos_level_t qos_level,
                           mqtt_retain_t retain);
```

Con la función siguiente se especifican los nombres de los usuarios, y sus contraseñas para conectarse a un broker MQTT.

```
/**
  * \brief Set the user name and password for a MQTT client.
  * \param conn A pointer to the MQTT connection.
  * \param username A pointer to the user name.
  * \param password A pointer to the password.
  */
void mqtt_set_username_password(struct mqtt_connection *conn,
                                char *username,
                                char *password);
```

Esta función declara el tópico *Last Will* de los clientes y el mensaje. Si la bandera *Will* está puesta en 1 cuando una solicitud Connect es aceptada, un mensaje *Will* DEBE publicarse cuando la conexión de red se cierre.

La funcionalidad siguiente puede usarse para obtener notificación de cuándo un dispositivo se ha desconectado del *broker*.

```
/**
  * \brief Set the last will topic and message for a MQTT client.
  * \param conn A pointer to the MQTT connection.
  * \param topic A pointer to the Last Will topic.
  * \param message A pointer to the Last Will message (payload).
  * \param qos The desired QoS level.
  */
void mqtt_set_last_will(struct mqtt_connection *conn,
                        char *topic,
                        char *message,
                        mqtt_qos_level_t qos);
```

Las funciones de ayuda siguientes sirven para confirmar el estatus de conexión de MQTT, para comprobar si la mota está conectada al *broker* con `mqtt_connected` y con `mqtt_ready`, si la conexión se establece y si hay espacio de buffer para publicar.

```
#define mqtt_connected(conn) \
   ((conn)->state == MQTT_CONN_STATE_CONNECTED_TO_BROKER ? 1 : 0)

#define mqtt_ready(conn) \
   (!(conn)->out_queue_full && mqtt_connected((conn)))
```

Práctica: MQTT y mosquitto

Ahora, construyamos un ejemplo sencillo usando el *broker* MQTT mosquitto [http://mosquitto.org/] v.3.1 que va a funcionar en nuestro host como un *Broker* MQTT, con la siguiente topología:

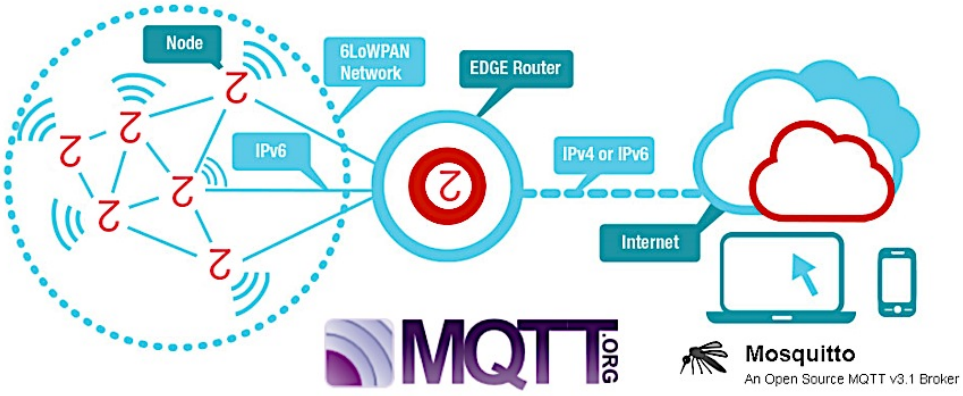

Figura 60. Escenario con MQTT y Mosquitto. Imagen propia.

La instalación de mosquitto es bastante sencilla. Las instrucciones están en el sitio web de mosquitto http://mosquitto.org/2013/01/mosquitto-debian-repository/. Para este ejemplo vamos a usar la configuración predeterminada.

Las instrucciones para compilar el enrutador de frontera están en las secciones anteriores.

Vamos a usar los ejemplos que están en `examples/cc2538dk/mqtt-demo`. Con algunos ajustes menores (como eliminar el código específico a cc2538) debería también funcionar para la **mota Z1.**

El ejemplo está basado en el formato *quickstart* de IBM. Para más información sobre esto, revisa el archivo README, o lee IBM MQTT doc.
https://docs.internetofthings.ibmcloud.com/messaging/applications.html

En el archivo `project-conf.h` la dirección del *broker* IPv6 se define de esta manera:

```
#define MQTT_DEMO_BROKER_IP_ADDR "aaaa::1"
```

Por defecto, el *broker* de `mosquitto` usa las direcciones IPv4/IPv6 del host; si se usa el script `tunslip6` con la dirección `aaaa::1/64` debería coincidir con `MQTT_DEMO_BROKER_IP_ADDR`.

Dato	Recuerda que algunos de los ejemplos son para una plataforma específica, así que hay que chequear si hay un `Makefile.target` predefinido o si el `TARGET` está definido en otra parte.

El ejemplo hace lo siguiente:

- Le publica información a un *broker* MQTT.
- Se subscribe a un tópico y recibe comandos desde un broker MQTT.

Nota también que `PUBLISH_TRIGGER` se ha mapeado al botón del usuario. Puede usarse para activar un evento "publicar".

La estructura de datos para el cliente MQTT se declara así:

```
typedef struct mqtt_client_config {
   char org_id[CONFIG_ORG_ID_LEN];                 (1)
   char type_id[CONFIG_TYPE_ID_LEN];               (2)
   char auth_token[CONFIG_AUTH_TOKEN_LEN];         (3)
   char event_type_id[CONFIG_EVENT_TYPE_ID_LEN];   (4)
   char broker_ip[CONFIG_IP_ADDR_STR_LEN];         (5)
   char cmd_type[CONFIG_CMD_TYPE_LEN];             (6)
   clock_time_t pub_interval;                      (7)
   int def_rt_ping_interval;                       (8)
   uint16_t broker_port;                           (9)
 } mqtt_client_config_t;
```

1) ID única de la organización.
2) Tipo de dispositivo.
3) *Token* de autorización (si es requerido).
4) Tipo de evento por defecto.
5) Dirección IPv6 del *broker.*
6) Tipo de comando por defecto.
7) Periodo del intervalo de publicación.
8) Hace ping al *parent* periódicamente y recupera el valor RSSI.
9) Puerto del *broker* por defecto. Por defecto es el `1883`.

Se definen y agregan los valores:

```
static mqtt_client_config_t conf;

static int
init_config()
{
/* Populate configuration with default values */
memset(&conf, 0, sizeof(mqtt_client_config_t));
memcpy(conf.org_id, DEFAULT_ORG_ID, strlen(DEFAULT_ORG_ID));
memcpy(conf.type_id, DEFAULT_TYPE_ID, strlen(DEFAULT_TYPE_ID));
memcpy(conf.auth_token,DEFAULT_AUTH_TOKEN,strlen(DEFAULT_AUTH_TOKEN));
memcpy(conf.event_type_id, DEFAULT_EVENT_TYPE_ID,
          strlen(DEFAULT_EVENT_TYPE_ID));
memcpy(conf.broker_ip, broker_ip, strlen(broker_ip));   memcpy(conf.cmd_type,
DEFAULT_SUBSCRIBE_CMD_TYPE, 1);   conf.broker_port = DEFAULT_BROKER_PORT;
conf.pub_interval = DEFAULT_PUBLISH_INTERVAL;   conf.def_rt_ping_interval =
DEFAULT_RSSI_MEAS_INTERVAL;
return 1;
}
```

La aplicación del ejemplo puede entenderse como una máquina de estado finito. A pesar de parecer complicado, en verdad es bastante sencillo. El proceso `mqtt_demo_process` comienza así:

```
PROCESS_THREAD(mqtt_demo_process, ev, data)
{
   PROCESS_BEGIN();
    if(init_config() != 1) {                                          (1)
     PROCESS_EXIT();
}
update_config();                                                      (2)

uip_icmp6_echo_reply_callback_add(&echo_reply_notification,           (3)
                                  echo_reply_handler);
etimer_set(&echo_request_timer, conf.def_rt_ping_interval);           (4)

while(1) {
      PROCESS_YIELD();

if(ev == sensors_event && data == PUBLISH_TRIGGER) {                  (5)
       if(state == STATE_ERROR) {
         connect_attempt = 1;
         state = STATE_REGISTERED;
       }
     }

if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
        ev == PROCESS_EVENT_POLL ||
        (ev == sensors_event && data == PUBLISH_TRIGGER)) {           (6)
       state_machine();
     }

if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {  (7)
       ping_parent();
       etimer_set(&echo_request_timer, conf.def_rt_ping_interval);
     }
   }
    PROCESS_END();
 }
```

1) Valores iniciales de la configuración, como se describió antes.

2) Crea la ID del cliente, publica y suscribe tópicos. Se fija el estado inicial `STATE_INIT` y se programa el evento `publish_periodic_timer`.

3) Registra el evento de retorno empleado cuando ocurre un evento *ping.*

4) Inicia el temporizador del *ping* periódico.

5) Permite la recuperación desde `STATE_ERROR` al presionar el botón de usuario.

6) Maneja los eventos `publish_periodic_timer` y botón. Aquí es cuando la aplicación en realidad inicia.

7) Cuando expira el temporizador periódico *ping*, hace *ping* al *parent*.

Cuando el `construct_client_id` es llamado por primera vez con el `STATE_INIT` la `state_machine` es llamada. Una exploración breve de la máquina de estado se muestra a

continuación. Nótese cómo en algunos eventos (como `STATE_INIT`) el driver salta inmediatamente al evento próximo ya que no hay ninguna declaración de `break`.

```
   static void
   state_machine(void)
   {
      switch(state) {

   case STATE_INIT:                  (1)
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
                      MAX_TCP_SEGMENT_SIZE);
        state = STATE_REGISTERED;

   case STATE_REGISTERED:            (2)
        if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
          connect_to_broker();
        } else {
          leds_on(STATUS_LED);
          ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);     }
   etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
        return;
        break;

   case STATE_CONNECTING:            (3)
        leds_on(STATUS_LED);
        ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);     /* Not
connected yet. Wait */
   DBG("Connecting (%u)\n", connect_attempt);
        break;

   case STATE_CONNECTED:             (4)
   /* Don't subscribe unless we are a registered device */
if(strncasecmp(conf.org_id, QUICKSTART, strlen(conf.org_id)) == 0) {
     DBG("Using 'quickstart': Skipping subscribe\n");
       state = STATE_PUBLISHING;
}

case STATE_PUBLISHING:            (5)
/* If the timer expired, the connection is stable. */
if(timer_expired(&connection_life)) {
    /*
        * Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
        * attempts if we disconnect after a successful connect
 */
       connect_attempt = 0;
}

if(mqtt_ready(&conn) && conn.out_buffer_sent) {
 /* Connected. Publish */
       if(state == STATE_CONNECTED) {
         subscribe();
         state = STATE_PUBLISHING;
       } else {
         leds_on(STATUS_LED);
         ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
         publish();
}
etimer_set(&publish_periodic_timer, conf.pub_interval);
       return;
     } else {
       /*
        * Our publish timer fired, but some MQTT packet is already in flight
        * (either not sent at all, or sent but not fully ACKd).
        *
```

```
        * This can mean that we have lost connectivity to our broker or that
        * simply there is some network delay. In both cases, we refuse to
        * trigger a new message and we wait for TCP to either ACK the entire
* packet after retries, or to timeout and notify us.        */

}
break;

case STATE_DISCONNECTED:          (6)
     DBG("Disconnected\n");
     if(connect_attempt < RECONNECT_ATTEMPTS ||
        RECONNECT_ATTEMPTS == RETRY_FOREVER) {
/* Disconnect and backoff */
       clock_time_t interval;
       mqtt_disconnect(&conn);
       connect_attempt++;

interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
         RECONNECT_INTERVAL << 3;

DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

etimer_set(&publish_periodic_timer, interval);
        state = STATE_REGISTERED;
       return;
} else {
/* Max reconnect attempts reached. Enter error state */
       state = STATE_ERROR;
       DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
}
break;

case STATE_CONFIG_ERROR:        (7)
     /* Idle away. The only way out is a new config */
     printf("Bad configuration.\n");
     return;

case STATE_ERROR:               (8)
default:
     leds_on(STATUS_LED);
     /*
      * 'default' should never happen.
      *
      * If we enter here it's because of some error. Stop timers. The only thing
      * that can bring us out is a new config event
      */
     printf("Default case: State=0x%02x\n", state);
     return;
}

/* If we didn't return so far, reschedule ourselves */
etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
 }
```

1) Punto de entrada; registra la conexión MQTT y sigue hacia el evento `STATE_REGISTERED`

2) Intenta conectar con el *broker*. Si el nodo no se ha unido a la red (no tiene una dirección global IPv6), vuelve a probar más tarde. Si el nodo tiene una dirección válida invoca la función `mqtt_connect` y establece el estado como `STATE_CONNECTING`. Luego fija el `publish_periodic_timer` a un ritmo más rápido.

3) Este evento sólo le informa al usuario sobre los intentos de conexión. Cuando se realiza la conexión MQTT con el *broker*, se acciona `MQTT_EVENT_CONNECTED` en la función de retorno `mqtt_event`

4) Como ahora estamos conectados, procede y publica. Puesto que no deberíamos estar usando el *quickstart* de IBM, nos saltamos cambiar el estado a `STATE_PUBLISHING` y proseguimos.

5) Chequea si la conexión MQTT está OK en `mqtt_ready` luego subscribimos y publicamos.

6) Maneja cualquier evento de desconexión activado desde `MQTT_EVENT_DISCONNECTED`

7) Detiene la aplicación. Sólo permite valores de configuración válidos.

8) Manejador de evento por defecto; detiene el temporizador y no hace nada más.

La función `publish` crea la cadena de datos que se van a publicar. A continuación tenemos un fragmento de la función resaltando solamente las partes más relevantes. El ejemplo publica periódicamente lo siguiente:

- Nombre del dispositivo.
- Una secuencia numérica incremental.
- Tiempo de funcionamiento del dispositivo (en segundos).
- Temperatura del chip.
- Valor de la batería.

```
static void
publish(void) {
   len = snprintf(buf_ptr, remaining,
                  "{"
                  "\"d\":{"
                  "\"myName\":\"%s\","
                  "\"Seq #\":%d,"
                  "\"Uptime (sec)\":%lu",
                  BOARD_STRING, seq_nr_value, clock_seconds());

len = snprintf(buf_ptr, remaining, ",\"Def Route\":\"%s\",\"RSSI (dBm)\":%d",
                  def_rt_str, def_rt_rssi);

len = snprintf(buf_ptr, remaining, ",\"On-Chip Temp (mC)\":%d",
           cc2538_temp_sensor.value(CC2538_SENSORS_VALUE_TYPE_CONVERTED));

len = snprintf(buf_ptr, remaining, ",\"VDD3 (mV)\":%d",
                  vdd3_sensor.value(CC2538_SENSORS_VALUE_TYPE_CONVERTED));

mqtt_publish(&conn, NULL, pub_topic, (uint8_t *)app_buffer,
                strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);

DBG("APP - Publish!\n");
 }
```

El `mqtt_publish` actualiza el *broker* MQTT con los nuevos valores y publica al `pub_topic` específico. El tópico por defecto es `iot-2/evt/status/fmt/json` como en la función `construct_pub_topic.` Este tópico sigue el formato IBM pero puede ser modificado en concordancia.

Cuando se recibe un evento desde un tópico al que estamos suscritos, el evento `MQTT_EVENT_PUBLISH` se activa y `pub_handler` es llamado. El ejemplo permite encender y apagar el LED rojo alternadamente.

El tópico al cual se suscribe el ejemplo por defecto es específicamente `iot-2/cmd/+/fmt/json` Para cambiar el estatus del LED necesitaríamos publicar al tópico `iot-2/cmd/leds/fmt/json` con valor 1 para encender el LED y 0 en otro caso.

```
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
   if(strncmp(&topic[10], "leds", 4) == 0) {
     if(chunk[0] == '1') {
       leds_on(LEDS_RED);
     } else if(chunk[0] == '0') {
       leds_off(LEDS_RED);
     }
     return;
   }
 }
```

Nótese el **+** en el tópico suscrito. Esto permite tener subconjuntos diferentes de tópicos, como por ejemplo `leds` .

Práctica: conectarnos a una plataforma IoT en el mundo real

Preparemos la aplicación Ubidots. Ubidots (www.ubidots.com) es una plataforma web en la nube que permite llevar la información de los dispositivos a Internet y poder controlarlos desde la nube.

Ejemplo de Ubidots IPv6 en Contiki nativo

Este ejemplo mostrará la funcionalidad básica de la librería Ubidots de Contiki:

- Cómo usar la librería para (enviar) POST a una variable.
- Cómo usar la librería para (enviar) POST a una colección.
- Cómo recibir (partes de) la respuesta HTTP.

Las librerías y el ejemplo de Ubidots no son parte oficial de Contki, sin embargo, el ejemplo funcional se encuentra disponible en:
https://github.com/g-oikonomou/contiki/tree/ubidots-demo

La Librería Ubidots de Contiki fue escrita por George Oikonomou.

El ejemplo de Ubidots se encuentra en `examples/ipv6/ubidots`.

La aplicación Ubidots está implementada en `apps/ubidots`.

La aplicación Ubidots usa sockets TCP para conectar con el host `things.ubidots.com` que tiene los siguientes extremos (*endpoints*) IPv4 e IPv6:

Figura 61. Ubidots

Para saber qué está pasando, activa el enunciado para depurar que está en el archivo `ubidots.c` , busca `#define DEBUG DEBUG_NONE` y reemplaza con:

```
#define DEBUG DEBUG_PRINT
```

La aplicación `ubidots` usa por defecto el host remoto `things.ubidots.com`
Sin embargo, si no se dispone del servicio NAT/NAT64 para resolver la dirección del host, el endpoint IPv6 puede definirse explícitamente así:

```
#define UBIDOTS_CONF_REMOTE_HOST "2607:f0d0:2101:39::2"
```

Dato	Si no tienes conexión local IPv6, hay servicios como gogo6 y hurricane electric que suministran túneles IPv6 para conexiones IPv4. Otras opciones como wrapsix permiten tener NAT64 para traducir las direcciones IPv6/IPv4. http://www.gogo6.com/ https://ipv6.he.net/ http://www.wrapsix.org/

La demo de Ubidots publica cada 30 segundos tiempo de operación (*uptime*) y la secuencia de números de la mota Z1. Así que, igual que en las secciones anteriores, necesitamos crear estas dos variables en Ubidots. Crea la fuente de datos y sus variables, y luego abre el archivo `project-conf.h` y reemplaza lo siguiente como corresponde:

```
#define UBIDOTS_DEMO_CONF_UPTIME        "XXXX"
#define UBIDOTS_DEMO_CONF_SEQUENCE      "XXXX"
```

El último paso es asignar un *Short Token* de Ubidots fijo, de manera que no sea necesario solicitar uno cada vez que expira. Selecciona un *Short Token* y añade esto a `Makefile`. El archivo debería ser así:

```
DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
CONTIKI_PROJECT = ubidots-demo
APPS = ubidots
UBIDOTS_WITH_AUTH_TOKEN=XXXXXXXX
ifdef UBIDOTS_WITH_AUTH_TOKEN
   DEFINES+=UBIDOTS_CONF_AUTH_TOKEN=\"$(UBIDOTS_WITH_AUTH_TOKEN)\" endif
all: $(CONTIKI_PROJECT)
CONTIKI_WITH_IPV6 = 1
CONTIKI = ../../..
include $(CONTIKI)/Makefile.include
```

Nótese que se debería reemplazar el `UBIDOTS_WITH_AUTH_TOKEN` **sin usar** comillas **“”.**

Ahora, todo debería estar listo. ¡Compilemos y programemos una mota Z1!

```
make TARGET=z1 savetarget
make clean && make ubidots-demo.upload && make z1-reset && make login
```

Deberías obtener la siguiente salida:

```
connecting to /dev/ttyUSB0 (115200) [OK]
Rime started with address 193.12.0.0.0.0.0.158
MAC c1:0c:00:00:00:00:00:9e Ref ID: 158
Contiki-d368451 started. Node id is set to 158.
nullmac nullrdc, channel check rate 128 Hz, radio channel 26
Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:009e
Starting 'Ubidots demo process'
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_ERROR_NO_NET
Ubidots client: STATE_STARTING
Ubidots client: Checking 64:ff9b::3217:7c44
Ubidots client: 'Host: [64:ff9b::3217:7c44]' (remaining 44)
Ubidots client: STATE_TCP_CONNECT (1)
Ubidots client: Connect 64:ff9b::3217:7c44 port 80 event_callback: connected
Ubidots client: STATE_TCP_CONNECTED
Ubidots client: Prepare POST: Buffer at 199
Ubidots client: Enqueue value: Buffer at 210
Ubidots client: POST: Buffer at 211, content-length 13 (2), at 143 Ubidots
client: POST: Buffer at 208
Ubidots client: STATE_POSTING (176)
Ubidots client: STATE_POSTING (176)
Ubidots client: STATE_POSTING (144)
Ubidots client: STATE_POSTING (112)
Ubidots client: STATE_POSTING (80)
Ubidots client: STATE_POSTING (48)
Ubidots client: STATE_POSTING (16)
Ubidots client: STATE_POSTING (0)
Ubidots client: HTTP Reply 200
HTTP Status: 200
Ubidots client: New header: <Server: nginx>
Ubidots client: New header: <Date: Fri, 13 Mar 2015 09:35:08 GMT> Ubidots
client: New header: <Content-Type: application/json>
Ubidots client: New header: <Transfer-Encoding: chunked>
Ubidots client: New header: <Connection: keep-alive>
Ubidots client: New header: <Vary: Accept-Encoding>
```

```
Ubidots client: Client wants header 'Vary'
H: 'Vary: Accept-Encoding'
Ubidots client: New header: <Vary: Accept>
Ubidots client: Client wants header 'Vary'
H: 'Vary: Accept'
Ubidots client: New header: <Allow: GET, POST, HEAD, OPTIONS>
Ubidots client: Chunk, len 22: <[{"status_code": 201}]> (counter = 22) Ubidots
client: Chunk, len 0: <(End of Reply)> (Payload Length 22 bytes)
P: '[{"status_code": 201}]'
```

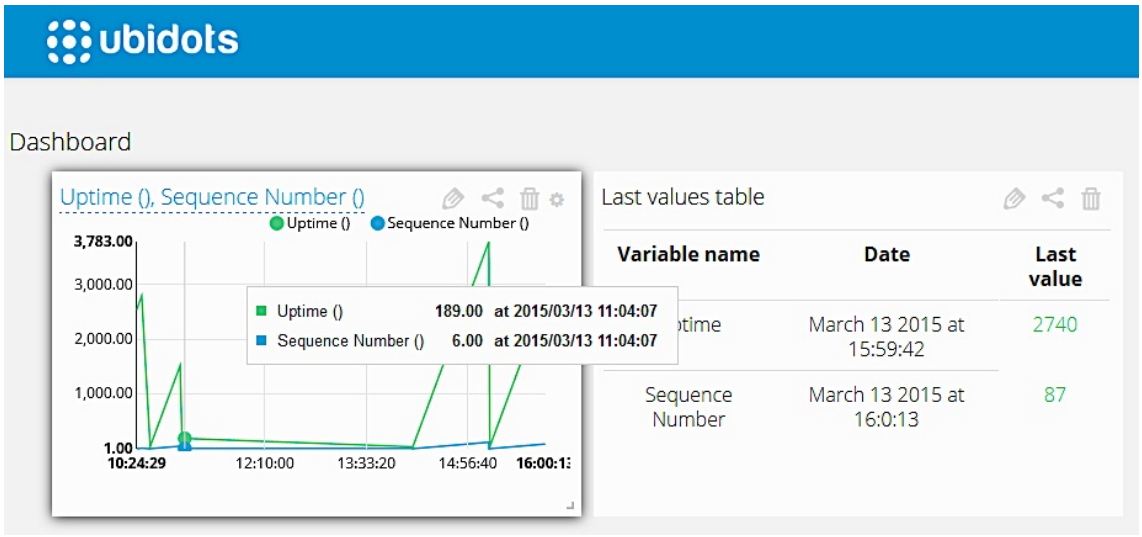

Figura 62. Valores publicados en Ubidots. Imagen propia.

Los valores se presentan usando una gráfica de varias líneas (**Multi-line chart)** y también una tabla de valores (**Table-Values)**.