Análisis y prueba de escenarios de red en el sistema operativo RIOT para dispositivos IoT

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Iván Álvarez Belotto
Tutores:
Rafael Marín López
Gabriel López Millán

2 de Julio de 2025

Facultad Informática Universidad Murcia

UNIVERSIDAD DE MURCIA

Facultad de Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Facultad de Informática - Universidad de Murcia

Análisis y prueba de escenarios de red en el sistema operativo RIOT para dispositivos IoT

Autor

Iván Álvarez Belotto

Tutores

Rafael Marín López

Departamento de Ingeniería de la Información y las Comunicaciones

Gabriel López Millán

Departamento de Ingeniería de la Información y las Comunicaciones

Curso 2024/2025

Murcia, 2 de Julio de 2025

Declaración firmada sobre originalidad del trabajo

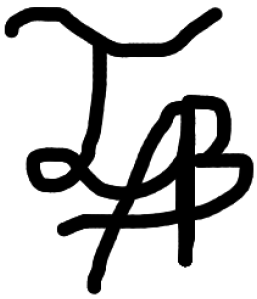
D. **Iván Álvarez Belotto**, con DNI [redacted] estudiante de la titulación de **Grado en Ingeniería Informática** de la Universidad de Murcia y autor del TFG titulado "**Análisis y prueba de escenarios de red en el sistema operativo RIOT para dispositivos IoT**".

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016 y 28-09-2018), así como la normativa interna para la oferta, asignación, elaboración y defensa de los Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015)

DECLARO:

Que el Trabajo Fin de Grado presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declara que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial.

Murcia, a 2 de Julio de 2025



Fdo.: Iván Álvarez Belotto
Autor del TFG

Resumen

En este trabajo, se ha llevado a cabo un análisis exhaustivo acerca del sistema operativo RIOT, una solución software diseñada para interactuar con dispositivos IoT caracterizados por su bajo consumo de energía, poco gasto en memoria y capacidades de cómputo limitadas. Entre las propiedades fundamentales de RIOT, conviene resaltar que permite la gestión de tareas concurrentes, el establecimiento de comunicaciones en redes de bajo consumo y la monitorización de actividades en tiempo real.

Más allá del análisis anterior, el siguiente TFG se centra en el desarrollo y la prueba de escenarios de red en RIOT, los cuales sean acordes a la pila de red que este sistema operativo soporta por defecto. Para ello, definiremos un modelo OSI particular sujeto a esta pila y estudiaremos en profundidad cada uno de los protocolos que lo componen, siendo estos IEEE 802.15.4, 6LoWPAN, IPv6, ICMPv6, RPL y CoAP.

Respecto a los escenarios a implementar, distinguiremos dos: el primero consistirá en la configuración y puesta en marcha de comunicaciones a nivel de red y aplicación entre dos dispositivos o motas IoT que comparten un enlace inalámbrico, donde este define una red de baja potencia y reducida transmisión de datos. Con el fin de probar que hay conectividad en las capas indicadas, utilizaremos mensajes ping y solicitudes CoAP de tipo GET, construidas por un cliente para obtener un recurso alojado en un servidor. Por otro lado, el segundo escenario trasladará el anterior a un intercambio de información análogo, pero en el que los extremos de la comunicación sean una mota IoT y un PC, con este último situado en una red IPv6 externa. De este modo, empezaremos diseñando un escenario dentro de una red con recursos restringidos y, como siguiente paso, veremos que es posible que los mensajes procedentes de una red inalámbrica puedan enrutarse hacia una red cableada, y viceversa, gracias a un componente intermedio denominado Border Router.

Por consiguiente, el objetivo del trabajo se basa en analizar, montar y probar el establecimiento de comunicaciones de red en RIOT. Nótese que este tema sigue siendo objeto de estudio y debate dentro de la comunidad de desarrolladores asociada a este sistema operativo. Sin embargo, la información existente sobre él es muy extensa y se encuentra dispersa. A modo de ejemplo, es bien sabido que RIOT ha sufrido diversas actualizaciones desde su lanzamiento, lo que ha provocado que un amplio número de sus aplicaciones software dejen de estar mantenidas, sin que ello implique que ya no tengan interés para fines de investigación.

En consecuencia, este TFG también pretende reunir toda la documentación de referencia acerca del despliegue de escenarios de red en RIOT y extraer de ella una guía que sea clara, concisa y permita llevar a cabo este proceso con éxito. Para ello, nos ayudaremos de una explicación teórica previa de los dos escenarios comentados y haremos uso de aplicaciones de RIOT ya programadas, centrándonos en la funcionalidad mínima que necesitamos de estas para completar los objetivos. Recordemos que dichos objetivos son tanto la correcta transmisión como recepción de mensajes ping y solicitudes CoAP.

Así, una vez diseñados ambos escenarios, procederemos a identificar las tecnologías, herramientas y, en síntesis, todo el material necesario para comenzar su implementación dentro de RIOT. Entonces será cuando describiremos los dispositivos de Nordic Semiconductor con los que representaremos las motas IoT y el Border Router, las aplicaciones software de RIOT que nos permitirán instalar las funcionalidades correspondientes a un cliente CoAP, servidor CoAP y un Border Router sobre los dispositivos anteriores, así como aquellas utilidades que se requiere incorporar en el PC con la finalidad de que la compilación y ejecución de las aplicaciones se complete satisfactoriamente.

La fase de desarrollo anterior puede resumirse en que, para el escenario entre dos motas IoT, utilizaremos dos placas nRF52840 DK sobre las que cargaremos el firmware de la aplicación "gcoap" de RIOT. En concreto, una de ellas hará el rol de cliente CoAP y la otra será el servidor. Mientras, en el segundo escenario, tendremos de nuevo una mota IoT representada por un nRF52840 DK, que será el servidor CoAP, y el cliente será el PC en el que habremos instalado RIOT y con el que estemos trabajando en todo momento. Por otra parte, el Border Router se implementará dentro de un dispositivo nRF52840 Dongle, al que se le cargará en memoria el firmware de la aplicación "gnrc_border_router" de RIOT.

Otro aspecto relevante para comprobar que los escenarios en RIOT se comportan tal y como se puede considerar desde un punto de vista teórico será la captura de los paquetes intercambiados en cada caso. Para el escenario entre la mota IoT y el PC, este proceso se realizará desde la interfaz cableada por la que se comunicará el ordenador personal, mientras que en el escenario entre dos motas IoT, aprovecharemos el nRF52840 Dongle para instalar sobre él la funcionalidad de un nRF Sniffer, con el que obtendremos una traza de las tramas transmitidas en la red inalámbrica.

Tras analizar cómo manejar correctamente los dispositivos anteriores y cargar aplicaciones software en ellos, estaremos en disposición de montar los escenarios de red de acuerdo al diseño teórico estudiado y, finalmente, desarrollaremos una serie de pruebas de concepto para verificar que los objetivos se satisfacen. Como parte de este último procedimiento, gracias a las trazas obtenidas podremos analizar la estructura de los paquetes intercambiados.

Por tanto, este trabajo parte de un enfoque teórico, centrado en explicar las características principales de RIOT, los protocolos utilizados y los escenarios de red considerados (que sirven como base para cualquier desarrollador que pretenda desplegar comunicaciones a nivel de red y aplicación en este sistema operativo), para posteriormente detallar cómo llevar tales escenarios a la práctica, trabajando con dispositivos IoT específicos dentro de las terminales ofrecidas por las aplicaciones de RIOT empleadas y consiguiendo capturar todo el tráfico generado para estudiar el flujo resultante de las comunicaciones.

Extended Abstract

The purpose of this work is to analyze, develop and test a solution for establishing network and application level communications within RIOT, one of the newest operating systems designed to be deployed on IoT devices. Among its features, RIOT adopts a modular structure with a minimalist kernel and has a unique network stack, called GNRC, that supports protocols for network communications such as IPv6, ICMPv6 or 6LoWPAN.

In particular, 6LoWPAN is characterized by compressing IPv6 packets so that they can be transported in energy efficient wireless networks with low data transmission rates, on which IoT devices tend to operate and exchange information between them. This protocol is based on the IEEE 802.15.4 standard that only defines the physical and media access control layers of the OSI model for this type of network. Hence, 6LoWPAN is used to scale to the network layer. Nevertheless, it is not able to provide packet routing on its own, and this issue led to the definition of additional technologies, the most popular being the RPL protocol.

It is significant to notice that RIOT constitutes an open source collaborative project with a wide variety of example applications, whose code is written in the C programming language. Moreover, the modules that are declared in the "Makefile" of these applications, which can have network, transport, system or other functionalities, are added at compile time so the applications are executed properly and resource use remains low. All things considered, RIOT is just one of the many existing options for IoT dedicated operating systems. Some alternatives are Tiny OS, Contiki OS or Zephyr, the latter being a system with very similar characteristics but more oriented towards industrial projects. However, the choice of RIOT is a perfect fit to the context of this work as it is commonly used in academic or partnership environments. Besides, since its first appearance in 2013, RIOT's development has continued to grow, giving rise to multiple tasks that create a challenging roadmap for potential or current contributors.

In this document, we analyze the design and deployment in RIOT of connectivity scenarios with support for IEEE 802.15.4, 6LoWPAN and the CoAP application layer protocol so they can serve as a basis for future RIOT works, or as a starting point for studies on the quality of communications between IoT devices within this operating system, in terms of latency, packet transmission or failure rates. Specifically, communications are developed according to the OSI layer layout for wireless networks we describe hereafter:

The upper layer will correspond to CoAP, the constrained application protocol that is oriented towards HTTP type communications for IoT devices, and that embraces the REST model, where a server hosts a series of resources and clients can perform certain operations on these. In the context of this work, we will only address GET requests from clients in order to retrieve a certain resource. Furthermore, it is well known that CoAP packets are transported over UDP, which will constitute the transport layer.

If we continue in descending order, we will have IPv6 alongside ICMPv6 for network support and, regarding wireless networks, we will also need to consider both 6LoWPAN and RPL protocols. Finally, in the bottom layer, with respect to wired communications we will work with Ethernet technologies, while in wireless networks we will adopt the IEEE 802.15.4 standard, which transmits by radio the messages on the link layer.

To fully understand the standards and protocols used in the layers of our OSI model, we will previously highlight the state of the art for each of them, just as we will briefly describe the features related to the other operating systems for IoT devices we already mentioned and what differentiates them from RIOT.

Although there is extensive documentation about RIOT, the process for deploying network scenarios on this operating system is not directly specified. In other words, the information is scattered and, therefore, in this work we bring order to all the available information and explain in a clear way how to set up this kind of communication scenario. In particular, we will develop a scenario in which two IoT motes can exchange IPv6 and CoAP packets over a shared wireless network, and another one that sets an analogous communication among an IoT mote and a PC. In the last case, messages from one end to the other go through an IPv6 external public network, where the PC is located.

In order to test communications at the network level, we use ping messages (equivalent to *ICMPv6 Echo Request* packet transmission) that ensure that two nodes on the network can discover each other. Concerning the application plane, verification will consist of the correct sending of queries for a specific resource that a CoAP client builds to a CoAP server, leading up to the expected response from the server.

Before studying how to model and integrate both scenarios into the RIOT operating system, we will delve into how they are designed in a theoretical way, describing the communication flow, as well as analyzing the format that both ICMPv6 and CoAP messages will have to follow. For this purpose, we will work with the values of IPv6 addresses and with the names of interfaces that will then be used in the deployment in RIOT, thus both perspectives can be correlated immediately.

Apart from the above specifications, the bulk of this work will be to set up the scenarios in practice in the RIOT environment. To this end, we will take into account different aspects. First of all, as part of the hardware material used, we will have at our disposal Nordic Semiconductor nRF52840 Development Kits, with which we will represent an IoT mote. In other words, on these boards we will load the firmware of "gcoap" RIOT application so that they act as a CoAP server or a CoAP client, depending on the role of each one. In addition, the PC will run a desktop image of the Ubuntu operating system, on which RIOT will be installed.

One of the key points about the second scenario is that, when an IoT device tries to communicate with the PC, the networks in which each of them operates are of different nature, this is, the mote's is wireless whereas the PC's is wired. Consequently, packets originated by the mote and destined to the PC, or vice versa, cannot reach their destination without the presence of an intermediate component, more precisely a Border Router. This device must be connected to both networks through a separated interface and it is essential for routing messages between the mote and the PC. To be able to route messages on the 6LoWPAN wireless network, which does not have this behavior by default, the Border Router will support RPL in it. Therefore, packets from or to the IoT mote shall be transmitted without any problem. Since the three components involved (namely the PC, the Border Router and the mote) do not share the same network segment, each will have a global IPv6 address, allowing them to make contact with one another based on certain static routes that we will define during the implementation in RIOT.

The functionality related to the Border Router will be implemented on a nRF52840 Dongle device. Although in the scenario corresponding to IoT motes that live together in a wireless network, the communication is set up immediately without any need for routing, we can also take advantage of this USB component by configuring it as a Sniffer thanks to the *nRF Sniffer* tool from Nordic, and thus capture the packets exchanged by the IoT devices. By doing so, we will easily check that the packets follow the layers' structure from the considered OSI model.

On the other hand, the capture of packets in the second scenario will be conducted from the wired interface through which the PC communicates. These methods for tracing will be enough for us to discern differences when messages are sent at the link level by Ethernet compared to when transmitted by IEEE 802.15.4, as well as to dig deeper into their content.

Another relevant part in order to make the scenarios work within RIOT will be to identify those RIOT applications which can help us to establish the desired communications. We will see that the application "gcoap" supports a CoAP client and server at the same time, allowing us to deploy the exchange at the network and application levels, due to the fact that this application facilitates the discovery of motes or other devices through ping messages and the building of GET requests as a CoAP client.

Meanwhile, like its name suggests, the "gnrc_border_router" application will implement a Border Router on the nRF52840 Dongle. Despite the fact that in this application's official documentation there are different ways to configure the IoT device used as a Border Router, we will focus on the changes that the "Makefile" needs so that the execution of "gnrc_border_router" results in the creation of a virtual network interface on the PC, with which this computer can talk to the router and, henceforth, to the mote. Going into the technical details, this is achieved by setting a USB link between the PC and the Dongle, where we will have loaded the program of the Border Router, according to the CDC-ECM model that ensures connectivity among these devices through the wired IPv6 network.

Given the growing importance of security in terms of the IoT scope, we will equally consider integrating in the scenario between IoT motes an alternative implementation of the CoAP protocol known as libCoAP. Even though this version was dropped from RIOT a few years ago, looking at alternative branches of the project, functional examples of a libCoAP client and server can be found. Nonetheless, the main interest in this version of CoAP is that it was designed to implement OSCORE, a protocol that provides security at object level in order to give integrity, authentication and confidentiality to CoAP messages.

In spite of some applications trying to support the integration of OSCORE along with libCoAP in RIOT, such as libOSCORE, unfortunately the solutions presented so far are not uniform and fail in a considerable number of tests. Moreover, a disadvantage for this issue to be addressed in RIOT is that, by supporting the native GNRC network stack, the development of this operating system is oriented to take care and advance that approach, then priming CoAP implementations as "gcoap", which do turn out to be much more operational and efficient.

In order to summarise the software and hardware material used, on the one hand, for the scenario between IoT motes, we have two nRF52840 DKs representing those motes, and on them will be loaded either the"gcoap" software application from RIOT or libCoAP client and server applications from an alternative branch of the RIOT repository, depending on the chosen CoAP implementation. Meanwhile, the Dongle will have a plugin loaded to act as a Sniffer. On the other hand, in the scenario between an IoT mote and the PC, the latter will have the CoAP client functionality, the IoT mote will again be an nRF52840 DK that has "gcoap" application program recorded and the Dongle, connected to the PC, will have the "gnrc_border_router" application flashed on it so it acts as a Border Router.

Once we have an idea of the objectives we have discussed; we have theoretically designed everything necessary to understand how communication works in both scenarios at network and application levels; specified the material used (and the role of each board in each scenario) and what applications from RIOT are the ones that make the nodes behave as we want, we will have to group all these remarks and, having them in mind, deploy the scenarios in RIOT, working from command line with the applications' shells for the purpose of consulting the IPv6 addresses of the devices or the available commands; initializing protocols or configuring routes between different network nodes. Prior to this process, we will explain how to install the technologies required for RIOT applications to load on the Nordic boards, the mandatory dependencies to work with RIOT, and what it takes to set up the Sniffer properly so it has a linked interface in *Wireshark*.

After the above is completed, we will carry out tests on each of the scenarios based on the sending of ICMPv6 and CoAP messages, and we will describe in detail the content of these messages according to information that we extract from *Wireshark* traces (that's why we do the packet captures with the help of the Sniffer and the virtual interface of the PC). The correct performance of the above tests completes the objectives of this document and proves that the scenarios designed and implemented work within RIOT.

To sum up, thanks to the scenarios exposed and deployed, we will have studied the RIOT GNRC network stack, the protocols considered in this work such as CoAP, 6LoWPAN and IEEE 802.15.4, we will also have made a guide for first-time users to configure RIOT and interact with its applications to load them on Nordic devices, we will have gathered scattered information on how to shape network scenarios in this operating system, and in general we will have addressed the features related to communications between low power IoT devices within RIOT and how we can make these not only be restricted to a shared wireless network, but they can go out to other IPv6 networks as well. All the contributions presented in this work will set a precedent for other IoT developers who choose to work on the RIOT operating system to set up ambitious related projects on a larger scale.

Índice de contenidos

Índice de figuras

1. Introducción

A finales del siglo XX, se acuñó el término Internet of Things (IoT) para referirse a la conexión de entidades físicas al mundo digital. Desde entonces, la expansión de Internet se ha visto acentuada por la aparición de millones de dispositivos inteligentes, caracterizados por ser objetos físicos con tecnologías integradas que les permiten intercambiar datos con otros equipos a través de una infraestructura de red compartida. Formalmente, estos dispositivos se conocen como dispositivos IoT. Muchos de ellos aparecen con frecuencia en la vida cotidiana, tal es el caso de sensores, sistemas de frenado o electrodomésticos.

Los dispositivos IoT de bajo nivel se caracterizan por su heterogeneidad, bajo consumo de energía, poco gasto en memoria y capacidades de cómputo limitadas. Estos factores hacen que no cuenten con los recursos suficientes para ejecutar sistemas operativos convencionales como Linux o Windows, lo que dio lugar al diseño de soluciones software con el fin de manejar correctamente los recursos hardware disponibles. Así, se están desarrollando una amplia gama de sistemas operativos, la mayoría de código abierto, adaptados para este tipo de dispositivos.

Entre los numerosos sistemas operativos para dispositivos IoT de bajo nivel que existen en la actualidad, destacan, de mayor a menor antigüedad, Tiny OS [1], Contiki OS [2], RIOT [3] y Zephyr [4]. Todos ellos pueden utilizarse sobre microcontroladores que no dispongan de Memory Management Unit (MMU) y ofrecen requisitos mínimos en términos de uso de Random Access Memory (RAM) y Read Only Memory (ROM) de cara al desarrollo de una aplicación básica.

En este trabajo, nos centramos en el sistema operativo RIOT, orientado en particular a entornos académicos y a la puesta en marcha de proyectos colaborativos. Algunas de las razones que condujeron a su elección son que RIOT tiene soporte para sistemas en tiempo real, sigue una estructura modular que mantiene el gasto en memoria a un nivel bajo y, además, es compatible con un gran número de protocolos o estándares de red como Internet Protocol version 6 (IPv6) [5], IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) [6], Routing Protocol for Low Power and Lossy Networks (RPL) [7] y Constrained Application Protocol (CoAP) [8].

El punto de partida que ha servido de referencia para los escenarios que se modelizan en este TFG se encuentra en un proyecto práctico [9] realizado por uno de los tutores, donde se documenta cómo construir una red Thread [10] en el sistema operativo Zephyr por medio de dispositivos hardware de Nordic Semiconductor. El presente trabajo tiene como objetivo trasladar el escenario de conectividad a nivel de red y aplicación ahí descrito al entorno del sistema operativo RIOT. Para ello, se trabajará con la pila de red Generic (GNRC) Network Stack [11], que constituye la pila de red por defecto que utiliza RIOT. En lugar del protocolo Thread, GNRC soporta 6LoWPAN para establecer comunicaciones entre dispositivos IoT en redes de bajo consumo.

Según los dispositivos involucrados, desarrollaremos dos escenarios de red que ejerzan como base para el despliegue de IEEE 802.15.4, 6LoWPAN y CoAP en RIOT. Conviene resaltar que, en el momento en el que se publica este trabajo, la información relativa al montaje de escenarios de red en RIOT se encuentra de forma muy distribuida y, en ocasiones, esta es poco clara por la existencia de diferentes configuraciones incompatibles o de aplicaciones que ya están en desuso.

Por consiguiente, una de las aportaciones fundamentales de este TFG es dar claridad y ordenar toda la información disponible para desplegar escenarios de este tipo, agrupando fuentes de distinto origen y ofreciendo así una documentación autocontenida por la que se puedan llevar a cabo comunicaciones a niveles de red y aplicación basadas en protocolos conocidos.

En concreto, el primer escenario consistirá en un par de motas IoT, representadas por dispositivos nRF52840 Development Kit (nRF52840 DK) [12], que intercambiarán paquetes IPv6 y CoAP de forma inalámbrica entre sí por medio de un enlace local compartido. El segundo llevará el intercambio anterior al contexto de una comunicación entre una mota IoT y un Personal Computer (PC) a través de una red IPv6 pública. En esta situación, resultará indispensable incluir un router que encamine los paquetes desde un extremo al otro, el cual se implementará en un nRF52840 Dongle [13].

Por tanto, el fin último que persigue este trabajo es analizar y desplegar sobre RIOT un escenario en el que dos motas IoT puedan intercambiar solicitudes y respuestas en una red inalámbrica compartida gracias al protocolo CoAP, y que esta comunicación también pueda establecerse entre una mota IoT y un PC a través de una red externa.

Cabe resaltar que, para referirnos al PC que ejerce de sistema anfitrión sobre el cual se instala el sistema operativo RIOT, utilizaremos indistintamente los términos PC y sistema operativo anfitrión en adelante.

Aparte de explicar en detalle sendos escenarios desde un punto de vista teórico y general, la principal aportación de este trabajo reside en el estudio acerca de cómo llevar los escenarios especificados al ámbito del sistema operativo RIOT. Este proceso no solo conlleva la explicación de las herramientas y tecnologías necesarias para ello, sino que requiere identificar las aplicaciones existentes en RIOT que se pueden utilizar para cumplir el diseño ideado, así como las modificaciones que se deban incorporar en ellas.

Una vez se monten los escenarios sobre RIOT, se realizarán una serie de pruebas para verificar que las comunicaciones a nivel de red y aplicación funcionan correctamente. Además, en el caso particular del escenario entre motas IoT, dado que el nRF52840 Dongle no se utiliza como router, lo aprovecharemos para instalar sobre él la funcionalidad de un Sniffer, con el que analizaremos el tráfico capturado en esta situación. Mientras, para el segundo escenario, la captura del tráfico se llevará a cabo por una interfaz concreta del PC, que será con la que este se comunique. Tras completar todas las pruebas y estudiar el contenido de las trazas resultantes, al final del trabajo destacaremos la utilidad detrás de las aportaciones desarrolladas con el propósito de establecer comunicaciones entre dispositivos IoT dentro del sistema operativo RIOT.

Por consiguiente, en los siguientes capítulos se describirá de forma teórica el estado de los protocolos y estándares a utilizar (capítulo 2), se especificarán los objetivos y la metodología a seguir para cumplirlos (capítulo 3), se detallarán las aportaciones de este trabajo junto a las tecnologías empleadas (capítulo 4), se explicarán y analizarán las distintas pruebas llevadas a cabo para la validación de los objetivos (capítulo 5) y, finalmente, se concluirá el trabajo teniendo en cuenta posibles vías futuras de desarrollo.

2. Estado del arte

Como paso previo a la definición de los objetivos de este trabajo, es necesario contextualizar el estado actual de los siguientes protocolos y estándares que utilizaremos para el despliegue de los escenarios, así como describir brevemente las características de los sistemas operativos dedicados a IoT alternativos a RIOT que mencionamos en la introducción.

2.1. IEEE 802.15.4

El estándar Institute of Electrical and Electronics Engineeers (IEEE) 802.15.4 [14] describe las capas física y de control de acceso al medio pertenecientes al modelo Open Systems Interconnection (OSI) para redes Wireless Personal Area Network (WPAN), caracterizadas por su escasa potencia y baja velocidad de transmisión de datos. Además, gestiona la velocidad y calidad del enlace y ofrece soporte para comunicaciones seguras.

Especificaciones como Zigbee [15], 6LoWPAN [6] o Thread [10] amplían este estándar mediante el desarrollo de las capas superiores. Los dispositivos compatibles utilizan una de las bandas Industrial, Scientific and Medical (ISM) disponibles y reservadas mundialmente para su funcionamiento.

En particular, 6LoWPAN es un estándar de red diseñado para recursos con limitaciones de energía y orientado a redes WPAN, que define cómo comprimir y transportar paquetes IPv6 en tramas 802.15.4. Al favorecer la conectividad entre dispositivos con recursos restringidos y la salida de estos a Internet mediante gateways, supone una alternativa popular para el desarrollo IoT. Entre sus características fundamentales, se integra fácilmente con otras tecnologías inalámbricas o con redes IP, habilita la autoconfiguración de direcciones IPv6, soporta redes que adoptan una topología en forma de malla y puede incorporar mecanismos de seguridad IPv6 para garantizar la protección de los datos que circulan por la red.

2.2. IPv6

IPv6 [5] es el protocolo de Internet diseñado para resolver muchos de los problemas encontrados en el uso de Internet Protocol version 4 (IPv4), especialmente relacionados con el agotamiento de direcciones, aunque también relativos a la seguridad o extensibilidad. Constituye la opción idónea para el desarrollo de redes IoT por dos razones fundamentales:

- *Amplio espacio de direcciones*: al trabajar con direcciones de 128 bits, el espacio de direcciones de IPv6 es prácticamente ilimitado, lo cual simplifica la arquitectura de la red. Como cada dispositivo IoT va a tener su propia dirección IP enrutable de manera global, esto permite el establecimiento de comunicaciones extremo a extremo sin necesidad de aplicar Network Address Translation (NAT).

- *Mejora en seguridad*: dado que la información intercambiada entre dispositivos IoT suele ser de naturaleza sensible, IPv6 incluye para su protección herramientas de monitorización que aseguran la integridad del tráfico IP.

  Algunos ejemplos son Safe Neighbor Discovery Protocol (SEND) [16], que evita ataques de suplantación de identidad y de envenenamiento de caché, o Router Advertisement-Guard (RA-Guard) [17], que asegura que solo los anuncios de routers legítimos son procesados y aceptados por los nodos de la red.

En redes IPv6, el protocolo Internet Control Message Protocol version 6 (ICMPv6) [18] resulta fundamental para la administración de la congestión, la configuración automática de direcciones IPv6, el descubrimiento de vecinos o la detección de errores en la transmisión de mensajes. Por todas estas funciones, se dice que es un protocolo orientado al informe de problemas en la red, así como a tareas de mantenimiento y diagnóstico.

2.3. RPL

RPL [7] es un protocolo de enrutamiento que opera sobre IEEE 802.15.4 y está destinado a redes inalámbricas de bajo consumo de energía y susceptibles a pérdidas de paquetes. Admite comunicaciones del tipo muchos a uno y uno a uno.

El enrutamiento se basa en vectores de distancia. Cada nodo de la red tiene un rango asignado, que aumenta a medida que los dispositivos se alejan de los nodos raíces (también llamados receptores). Entonces, el reenvío de paquetes se lleva a cabo de acuerdo al criterio de seleccionar aquella ruta con el rango más bajo.

Por otro lado, la topología que subyace en RPL es la de un Directed Acyclic Graph (DAG), que se divide en uno o más Destination Oriented DAG (DODAG), uno por cada nodo raíz. Este nodo es el que ejerce de gateway hacia Internet para el resto de dispositivos de la red, y envía de forma periódica mensajes DIO (DODAG Information Object) para informar a los nodos vecinos de su existencia. Estos, tras escuchar el DIO, pueden decidir comunicarse con el nodo raíz mediante el envío de un mensaje DAO (Destination Advertisement Object) [19].

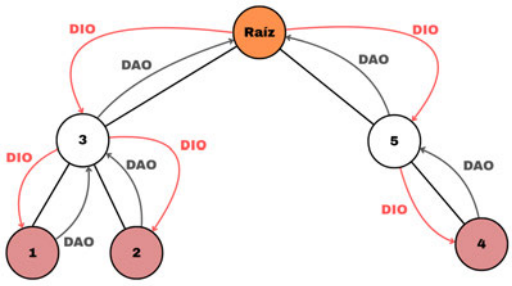


Figura 2.1: Construcción de un árbol DODAG en RPL con un solo nodo raíz

A modo de ejemplo, en la figura anterior, mostramos el proceso que se va repitiendo hasta que todos los nodos de la red RPL están conectados. En primer lugar, el nodo raíz empieza enviando mensajes DIO a sus vecinos, los nodos 3 y 5, que los escuchan y se conectan a él mediante el envío de DAOs. Después, estos dos nodos mandan sus propios DIOs a sus hijos, que también los escuchan y envían DAOs para conectarse cada uno a su padre, y eventualmente, al nodo raíz. De esta manera, se va construyendo el árbol DODAG con todos los nodos siguiendo una organización jerárquica.

2.4. CoAP

CoAP [8] es un protocolo que opera en la capa de aplicación y que fue diseñado para trasladar el modelo de solicitud-respuesta de Hypertext Transfer Protocol (HTTP) [20] a dispositivos y redes con recursos limitados, permitiendo la comunicación a través de Internet. Por este motivo, su uso en el IoT está muy extendido. De hecho, entre las características que ofrece para el desarrollo de aplicaciones en este ámbito, encontramos el soporte de multicast o la baja sobrecarga de datos en cada paquete [21].

HTTP y CoAP se basan en el modelo Representational State Transfer (REST), que describe la interacción entre los componentes de un sistema con el fin de transferir información entre ellos de la siguiente manera: una serie de servidores ponen a disposición de los clientes recursos bajo un Uniform Resource Locator (URL), y entonces los clientes acceden a tales recursos utilizando métodos como GET, PUT, POST o DELETE.

De manera nativa, CoAP es capaz de observar si se producen cambios que afecten a algún recurso en particular. Asimismo, detecta los dispositivos que se unen a la red y permite el intercambio de mensajes de forma asíncrona.

A diferencia de HTTP, que se ejecuta sobre Transmission Control Protocol (TCP) [22], CoAP lo hace a través de User Datagram Protocol (UDP) [23], lo que ahorra memoria y disminuye el número de operaciones de procesamiento. Esta es una de las razones por las que se considera a CoAP un protocolo "Constrained", aunque no es la única. La más relevante es que la cabecera CoAP tiene un tamaño mínimo de 4 bytes, lo que supone una reducción significativa en comparación con los cientos de bytes que puede tener una cabecera HTTP. Así, el mensaje tiene mucha menos sobrecarga y puede ser transmitido sin problema en tramas IEEE 802.15.4. En concreto, un mensaje CoAP está formado por [24]:

- *Cabecera fija (4 B)*: comprende la versión del protocolo (2 bits), el tipo de mensaje (2 bits), que puede ser Confirmable, Non-confirmable, Acknowledgement o Reset, el *Token Length* (TKL) de 4 bits, el *Code* (8 bits), que es similar al campo *Status Code* de HTTP, y, finalmente, el *Message ID* (16 bits).
- *Token*: opcional, de tamaño máximo de 8 B.
- *Options*: opciones que proporcionan parámetros para rellenar peticiones.
- *Payload*: carga útil del mensaje.

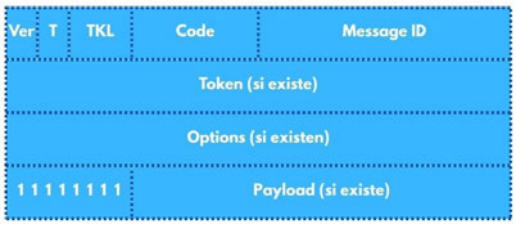


Figura 2.2: Formato de la trama de un mensaje CoAP

2.5. Tiny OS

Tiny OS [1] es el primer sistema operativo dedicado a IoT, que aparece en el año 2000 [25]. Las aplicaciones de Tiny OS están escritas en el lenguaje de programación nesC, un dialecto del lenguaje C optimizado para las limitaciones de memoria propias de las redes de sensores.

Los programas de Tiny OS están compuestos por componentes software, algunos de los cuales presentan abstracciones del hardware. Estos componentes se conectan entre sí mediante interfaces. Al igual que Contiki OS, ofrece funcionalidad pseudo-multihilo, es decir, alterna rápidamente entre múltiples hilos, dando la sensación de que se están ejecutando de forma simultánea. Dos limitaciones fundamentales de este sistema operativo es que no ofrece capacidades en tiempo real ni da soporte al protocolo IPv6 [26].

2.6. Contiki OS

Contiki OS [2] es un sistema operativo cuyo origen se remonta al año 2002. Sigue una estructura modular, basada en un modelo por capas que, además de soportar los protocolos IPv4, IPv6, TCP y CoAP, incluye una pila TCP/IP ligera que contiene los protocolos RPL y 6LoWPAN.

La planificación en Contiki está orientada a eventos, de manera que los nodos se despiertan para realizar mediciones únicamente debido a las interrupciones provocadas por ciertos eventos. La comunicación entre procesos, que se lleva a cabo mediante la técnica de paso de mensajes, también se implementa a través de este sistema [27]. A pesar de utilizar un subconjunto del lenguaje de programación C, Contiki no ofrece programación estándar en él, como tampoco proporciona soporte total para multi-threading o para funcionalidad en tiempo real.

2.7. Zephyr

Zephyr [4] es, junto a RIOT, el sistema operativo para IoT más popular en la actualidad [25]. Ambos son sistemas operativos en tiempo real escalables, de código abierto, que admiten diversas arquitecturas hardware. A diferencia de RIOT, el núcleo de Zephyr es modular, y en él encontramos una amplia gama de servicios, por ejemplo, el reparto de tiempo entre hilos de igual prioridad o servicios de gestión de energía [28].

Zephyr fue creado teniendo en cuenta la seguridad con el fin de ser el mejor ecosistema existente para dispositivos IoT. Entre sus características, ofrece múltiples algoritmos de planificación, mecanismos de protección de memoria y una pila de red nativa soportando protocolos IoT como CoAP u OpenThread [29]. Además, su ámbito de aplicación se orienta en particular al desarrollo industrial.

3. Objetivos y metodología

En este capítulo, se explicarán de forma precisa los objetivos a lograr con la realización de este trabajo, así como la metodología seguida para la consecución de los mismos. En primer lugar, los objetivos de este TFG son los siguientes:

- Analizar las características del sistema operativo RIOT, junto a la estructura de su pila de red interna y la de sus aplicaciones software de cara al despliegue de escenarios de red.
- Diseñar, montar y probar un escenario de red IEEE 802.15.4/6LoWPAN/CoAP en RIOT que asegure la conectividad entre dos motas IoT que compartan una red inalámbrica.
- Diseñar, montar y probar un escenario análogo al anterior, pero donde la comunicación se desarrolle desde una mota, ubicada en una red 6LoWPAN, a un elemento externo (en concreto, un PC), situado en una red IPv6 estándar, o viceversa. Para conectar ambas redes, será necesario desplegar un Border Router que actúe de componente intermedio.
- Capturar el tráfico generado y analizar así que, en cada escenario, las cabeceras de los paquetes intercambiados son acordes a los protocolos de un modelo OSI por capas que definiremos para redes inalámbricas.

El interés detrás del despliegue de los dos escenarios mencionados reside en poder simular un entorno IoT realista, en estudiar cómo funciona la conectividad IPv6 y CoAP sobre redes de bajo consumo e, igualmente, en validar la implementación de la pila de red GNRC de RIOT. Así, este TFG establece un modelo para el desarrollo y la prueba de escenarios de red IEEE 802.15.4/6LoWPAN/CoAP dentro de RIOT, de manera que estas aportaciones sirvan como base para futuros trabajos relacionados con este sistema operativo.

En lo que respecta a la metodología llevada a cabo, se organiza en los siguientes puntos:

- Primeramente, se procedió a la lectura de artículos relacionados con el sistema operativo RIOT para entender su origen, características y diseño (en particular [25], [26], [30] y [31]). El enfoque se centró especialmente en estudiar las razones que motivaron el despliegue de RIOT, al igual que la estructura hardware/software que adopta.
- Después, se instaló RIOT de manera local sobre un PC con un sistema anfitrión Ubuntu. Esta opción fue elegida porque los sistemas operativos de tipo Linux dan soporte nativo a todas las herramientas necesarias para RIOT y son los más recomendados [3].
- A partir de los cursos de aprendizaje de RIOT [32], se trabajó en tener una idea de cómo construir una aplicación y se probó el ejemplo "hello_world" [33] en modo nativo.
- Posteriormente, se conectaron los dispositivos IoT proporcionados por los tutores (dos placas nRF52840 DK) al PC y se intentaron ejecutar las aplicaciones "hello_world" y "blinky" sobre ellos. Este proceso no fue inmediato, y a lo largo de su realización, se descubrieron e instalaron las tecnologías necesarias para conseguir el funcionamiento previsto, con la ayuda de [34].

- Por otro lado, se recabó información para conocer la pila de red GNRC de RIOT y se diseñaron los escenarios a desarrollar de forma teórica, tal que estuvieran alineados con los objetivos planteados.

- Para los dos escenarios resultantes, se identificaron aplicaciones en RIOT cuya funcionalidad se correspondiera al comportamiento que se quería representar, considerando al mismo tiempo los cambios que hiciera falta incorporar en ellas.

  Además, en lo que respecta al segundo escenario, la aplicación con la funcionalidad asociada al router se instaló en un nRF52840 Dongle, también proporcionado por los tutores de este trabajo. Como paso previo, se ejecutó la aplicación "blinky" sobre el Dongle, lo que ayudó a saber cómo cargar una aplicación de RIOT correctamente en este dispositivo.

- Una vez completado el paso anterior, se montaron sobre RIOT los dos escenarios sin involucrar la capa de aplicación, es decir, tales escenarios se desplegaron sobre un contexto que no tuviera en cuenta CoAP y se centrara únicamente en el soporte de IPv6. Para ello, se utilizó la aplicación de RIOT "gnrc_networking" [35] en las motas, la cual inicia la pila de red GNRC y permite el intercambio de paquetes ICMPv6 y UDP a través de línea de comandos.

- Tras realizar pruebas en los escenarios con "gnrc_networking", se reemplazó su uso en las motas por el de las aplicaciones escogidas para integrar CoAP en RIOT. En concreto, se hizo uso de las implementaciones denominadas GCoAP [36] (exclusiva de RIOT) y libCoAP [37]. Entonces, con la experiencia del trabajo anterior, se montaron los escenarios finales en RIOT con CoAP incorporado. Como parte de este proceso, se configuraron las herramientas necesarias para capturar el tráfico generado en cada escenario.

- Para llevar a cabo las pruebas relativas a CoAP, se analizó la sintaxis con la que se construye una petición como cliente en las aplicaciones de RIOT elegidas y, seguidamente, se desarrollaron las pruebas de concepto a nivel de red y aplicación pertinentes en los escenarios finales.

- Completadas las pruebas, se estudiaron en profundidad las trazas capturadas, inspeccionando el contenido de los mensajes transmitidos en cada uno de los escenarios.

4. Estudio y despliegue de una red IEEE 802.15.4/6LoWPAN/CoAP en RIOT

A lo largo del presente capítulo, profundizaremos en las diferentes aportaciones de este trabajo. En concreto, analizaremos los conceptos más relevantes acerca del sistema operativo RIOT: su estructura modular, su pila de red y cómo se componen y ejecutan sus aplicaciones software. Seguidamente, estudiaremos el diseño de los dos escenarios introducidos con base en los objetivos planteados, así como el flujo de las comunicaciones que nos aseguran la conectividad a nivel de red y aplicación que buscamos. Una vez se tenga una perspectiva teórica de los escenarios, explicaremos el proceso detrás de su montaje dentro del sistema operativo RIOT, de manera que se tenga toda la funcionalidad requerida para la posterior ejecución de las pruebas. Dicho proceso conllevará también la explicación del material software utilizado, de las tecnologías necesarias para instalar RIOT y cargar las aplicaciones en las placas de las que se ha hecho uso, y de los ejemplos de aplicaciones de RIOT que nos ayudan a reflejar el escenario teórico en la práctica.

A la hora de describir los escenarios, trabajaremos en la medida de lo posible con direcciones de red o interfaces específicas sin perder generalidad. Así, se podrá establecer una correlación directa entre el estudio teórico y las pruebas. También adoptaremos un enfoque deductivo, partiendo de lo general hacia lo particular.

4.1. Análisis del sistema operativo RIOT

RIOT [3] es un sistema operativo de código abierto para microcontroladores, adaptado a arquitecturas de 8, 16 y 32 bits, cuyo diseño se vio motivado por el objetivo de satisfacer las necesidades de los dispositivos IoT. Aparte de las comentadas en la introducción, cabe destacar el soporte de pilas de comunicación para redes inalámbricas y cableadas, así como el ofrecimiento de capacidades en tiempo real de cara a que los dispositivos IoT respondan de inmediato ante los estímulos del entorno.

Por otro lado, RIOT permite la programación de aplicaciones bajo los lenguajes C y C++. Además, proporciona múltiples pilas de red y utilidades, que incluyen bibliotecas criptográficas, estructuras de datos o un shell, de forma independiente del proveedor y de la tecnología del dispositivo IoT.

Desde su aparición en el año 2013, RIOT está sustentado por una comunidad de usuarios abierta a todo el mundo. En el momento en el que se publica este trabajo, se estima que más de 290 personas han contribuido al desarrollo de este sistema operativo [25].

4.1.1. Estructura modular

RIOT adopta una arquitectura hardware/software que sigue una estructura modular, con un núcleo minimalista que proporciona funciones básicas multihilo como Inter-Process Communication (IPC), temporizadores o primitivas de sincronización.

Como los módulos se añaden en tiempo de compilación, para cada aplicación se construye solo lo que realmente se necesita y, por consiguiente, el número de recursos empleados resulta ser bajo. En particular, podemos distinguir los ocho módulos que se observan en la siguiente figura:

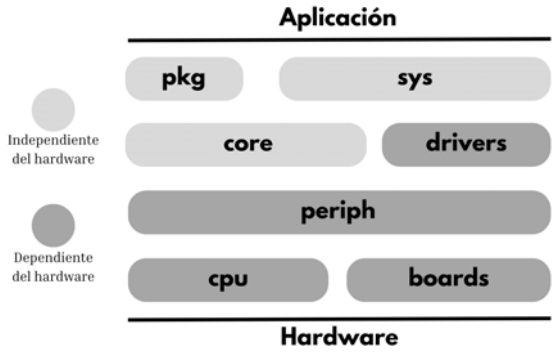


Figura 4.1: Estructura modular de RIOT (figura adaptada de [30])

Es importante destacar que la configuración del sistema al menos debe incluir el *core*, el módulo *cpu* y un *board*, siendo el resto de componentes opcionales. Los elementos de la figura 4.1 que no dependen del hardware del sistema son:

• *pkg*: engloba las librerías externas y aplicaciones que no forman parte del código principal del repositorio de RIOT, pero sí se pueden integrar si se añaden en el fichero "Makefile" correspondiente.

• *sys*: contiene las funcionalidades que hacen de RIOT un sistema operativo más allá de las responsabilidades del núcleo. Es decir, en *sys* encontramos las librerías de sistema que constituyen herramientas para networking, intérprete de comandos...

• *core*: este módulo implementa el núcleo del sistema operativo, así como funciones de IPC o gestión de hilos.

• *drivers*: aquí se agrupan los controladores de dispositivos de alto nivel, como pueden ser sensores, actuadores, radios...

• *periph*: este es el módulo que se utiliza para que los dispositivos accedan a los periféricos agregados al Microcontroller Unit (MCU).

• *cpu*: aparte de aportar inteligencia al sistema, da soporte al MCU y define puntos de entrada que podrían ser usados de cara al manejo de interrupciones externas.

• *boards*: este módulo contiene toda la funcionalidad asociada a una placa, que por lo general incorpora un controlador y algún tipo de dispositivo externo, como puede ser un sensor. Según el board que se especifique en la compilación de una aplicación en RIOT, la ejecución es:

- En modo nativo: el board *native* ejecuta el programa como si se tratara de un proceso Linux, asumiendo que el sistema operativo anfitrión es de este tipo. Esta opción es la que se utiliza por defecto si no se indica en su lugar alguna otra placa soportada.

- En hardware: en este caso, el programa de RIOT se carga en la memoria del MCU de la placa, es decir, se flashea en el board. Posteriormente, trabajaremos con este modo de ejecución para dos tipos de dispositivos nRF52840 de Nordic, el DK y el Dongle.

4.1.2. Pila de red GNRC

La pila de red modular que se utiliza dentro de RIOT para la comunicación a nivel de red se denomina GNRC [11]. Cada capa de la pila se ejecuta en su propio hilo, y cada hilo de una capa inferior tiene mayor prioridad que cualquier hilo de una capa superior. De este modo, el hilo de la implementación de la capa Media Access Control (MAC) tiene la prioridad más alta, mientras que los hilos de la capa de aplicación tienen la prioridad más baja. Los hilos se comunican gracias a la funcionalidad IPC del núcleo y a la interfaz de comunicación de la pila GNRC.

Por otro lado, para la comunicación entre componentes, se utilizan tres interfaces: *netapi* entre módulos internos de la pila de red, *sock* entre la aplicación y el módulo gnrc_sock, y *netdev* entre MAC y los drivers físicos. En concreto, la pila GNRC se organiza del siguiente modo:

- **Aplicación**: representa el código de usuario del que hace uso la red. Por ejemplo, una aplicación que abre un socket UDP para la transmisión de datos.
- **gnrc_sock**: es el módulo que implementa la interfaz de sockets, traduciendo las llamadas de sockets a operaciones internas que se utilizan dentro de la pila.
- **gnrc_tcp** y **gnrc_udp**: son las implementaciones de los protocolos de transporte TCP y UDP respectivamente, que enrutan el tráfico cuando la aplicación abre un socket TCP o UDP.
- **gnrc_ipv6**: implementa IPv6, entregando a las capas inferiores los datos procedentes de la capa de transporte. Se encarga del direccionamiento, la fragmentación y el enrutamiento de paquetes. Además, gnrc_sixlowpan da soporte para 6LoWPAN.
- **MAC**: implementa el control de acceso al medio. Usualmente, hay una instancia MAC por interfaz.
- **Driver**: engloba los controladores específicos del hardware de radio, que se ocupan de enviar y recibir tramas físicas.

La representación estándar de esta pila se muestra a continuación:

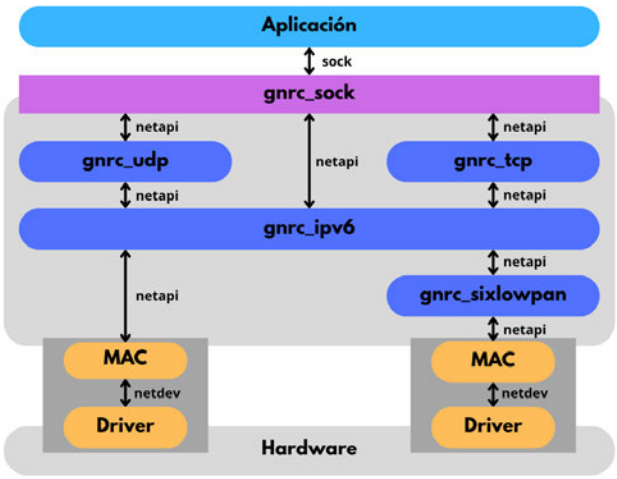


Figura 4.2: Pila de red GNRC (figura adaptada de [11])

4.1.3. Aplicaciones en RIOT

Uno de los aspectos que distinguió a RIOT de otros sistemas operativos dedicados a IoT fue la programación de aplicaciones software en lenguajes como C o Rust, motivando la portabilidad del código, principios de mínima duplicación o el modelo "Keep it simple, stupid" (KISS) que promueve una curva de aprendizaje suave para empezar a programar en RIOT.

Las aplicaciones en RIOT están formadas por dos ficheros fundamentales: "main.c", que contiene la funcionalidad principal del programa, y "Makefile". Cuando se completa la compilación de una aplicación en una placa, RIOT inicia por defecto dos hilos: el *main* y el *idle*. El primero en ejecutarse es el *main*, que llama a la función del mismo nombre. Cuando ningún otro hilo está listo para su ejecución, entonces el *idle* se inicia y automáticamente emplea el modo de energía más bajo posible para minimizar el consumo en la placa o en el propio sistema operativo anfitrión.

Por otro lado, el "Makefile" siempre define al menos dos macros: `APPLICATION`, la cual establece el nombre de la aplicación que se está desarrollando, y `RIOTBASE`, que se refiere a la ruta dentro del PC donde se aloja el repositorio con el código software de RIOT. También es indispensable incluir *Makefile.include* desde el `RIOTBASE`.

Para realizar aquellas tareas que aumentan la complejidad de una aplicación, la solución pasa por agregar macros adicionales en el "Makefile". Por ejemplo, `USEMODULE` sirve para agregar módulos con funcionalidad de red, de conexión, ..., `USEPKG` incluye paquetes externos o `FEATURES_REQUIRED` interactúa con controladores de periféricos.

De cara a entender más en profundidad cómo se define y funciona una aplicación en RIOT, ilustramos un ejemplo sencillo: la aplicación "blinky" [38]. Esta consiste en hacer parpadear un Light Emitting Diode (LED) de una placa, si hay alguno disponible.

```
# name of your application
APPLICATION = blinky

# If no BOARD is found in the environment, use this default:
BOARD ?= native

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../../..

# Comment this out to disable code in RIOT that does safety checking
# which is not needed in a production environment but helps in the
# development process:
DEVELHELP ?= 1

# Change this to 0 show compiler invocation lines by default:
QUIET ?= 1

# Use a peripheral timer for the delay, if available
FEATURES_OPTIONAL += periph_timer

include $(RIOTBASE)/Makefile.include
```

Figura 4.3: Fichero "Makefile" de la aplicación "blinky"

```
#include <stdio.h>

#include "clk.h"
#include "board.h"
#include "periph_conf.h"
#include "timex.h"
#include "ztimer.h"

static void delay(void)
{
    if (IS_USED(MODULE_ZTIMER)) {
        ztimer_sleep(ZTIMER_USEC, 1 * US_PER_SEC);
    }
    else {
        uint32_t loops = coreclk() / 20;
        for (volatile uint32_t i = 0; i < loops; i++) { }
    }
}

int main(void)
{
    while (1) {
        delay();
#ifdef LED0_TOGGLE
        LED0_TOGGLE;
#else
        puts("Blink! (No LED present or configured...)");
#endif
    }

    return 0;
}
```

Figura 4.4: Fichero "main.c" de la aplicación "blinky"

En el "Makefile", se definen los parámetros necesarios para la compilación y ejecución de esta aplicación en RIOT. En concreto, de acuerdo a la figura 4.3, encontramos de arriba abajo el nombre de la aplicación, la placa que se utiliza por defecto (si no se especifica ninguna otra), la ruta base hacia el sistema operativo RIOT, la habilitación de ayudas tanto para el desarrollo como para una salida limpia en el compilador, el uso opcional de periféricos y, por último, se importa el sistema "Makefile" genérico de RIOT.

Por otra parte, el fichero "main.c" está formado por la inclusión de las bibliotecas necesarias para manejar el sistema, el temporizador y los periféricos; la función *delay(void)*, que introduce un retardo de 1 segundo por medio de un temporizador o de una espera activa basada en el reloj de la Central Processing Unit (CPU); y la función *main(void)* que, en cada iteración de un bucle infinito, llama a *delay()* y, si la macro `LED0_TOGGLE` está definida, cambia el estado del LED (en caso contrario, se imprime un mensaje por consola que indica la ausencia de LED).

Para compilar e interactuar con esta aplicación en RIOT, dentro del directorio `/RIOT/examples/blinky`, el comando que se introduce en la terminal del PC que aloja este sistema operativo para IoT es:

```
make BOARD=nrf52840dk flash
```

Así, el código de "main.c" se compila y se carga el binario resultante en la placa indicada en la directiva `BOARD`, que hemos considerado que es el nRF52840 DK. Tras este comando, el programa se ejecuta en la placa automáticamente, por lo que el LED parpadeará al instante.

El procedimiento que acabamos de describir es la base para compilar la mayoría de aplicaciones en RIOT, no obstante, para aquellas que incorporan un shell como parte del programa, se complementa con el uso posterior del siguiente comando:

```
make BOARD=nrf52840dk term
```

Este abre la terminal de RIOT asociada al programa que se carga en la placa a través de un puerto serie Universal Serial Bus (USB), que si no se especifica en la directiva `PORT` es el "/dev/ttyACM0" por defecto. De este modo, a través de la terminal se puede interactuar con el dispositivo IoT mediante la entrada de comandos, lo que facilita el uso y despliegue de un gran número de aplicaciones en RIOT.

4.2. Modelo OSI para redes inalámbricas

Una vez hemos analizado en profundidad la estructura de la pila de red de RIOT y la de sus aplicaciones software, el último paso previo a la explicación de los escenarios que desarrollaremos es describir el modelo OSI por capas que tomaremos como referencia para las redes WPAN. Lo mostramos a continuación:

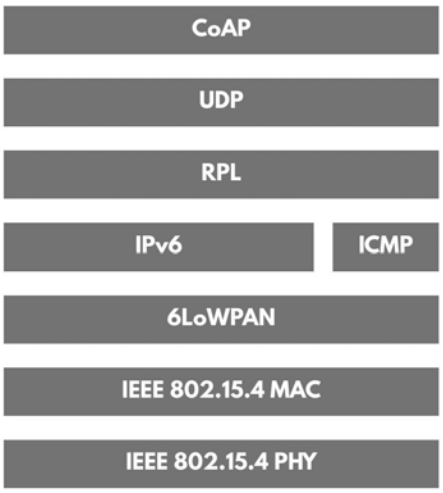


Figura 4.5: Modelo de capa OSI para redes inalámbricas considerado en este trabajo

Observemos que las dos primeras capas adoptan el estándar IEEE 802.15.4. Por encima de ellas, 6LoWPAN da soporte al uso de IPv6 (junto a ICMPv6) para la conectividad entre dispositivos a través de Internet, mientras que el protocolo RPL se encarga de que los mensajes que atraviesan una red inalámbrica puedan enrutarse a nodos específicos de dicha red o a destinos que pertenezcan a una red externa. Por consiguiente, por medio de RPL una mota IoT ubicada en una red 6LoWPAN puede descubrir el camino hacia un gateway a una red IPv6, lo que se alinea con el comportamiento del segundo de los escenarios descritos en el capítulo anterior. Por último, para las solicitudes y respuestas que intercambien los dispositivos en el plano de aplicación, usaremos el protocolo CoAP, que se transporta sobre UDP.

4.3. Diseño del escenario entre dos motas IoT

Una perspectiva del escenario objetivo que pretendemos desplegar en RIOT para obtener la conectividad IPv6 entre dos dispositivos IoT (que, en adelante, denotaremos por M1 y M2) es la siguiente:

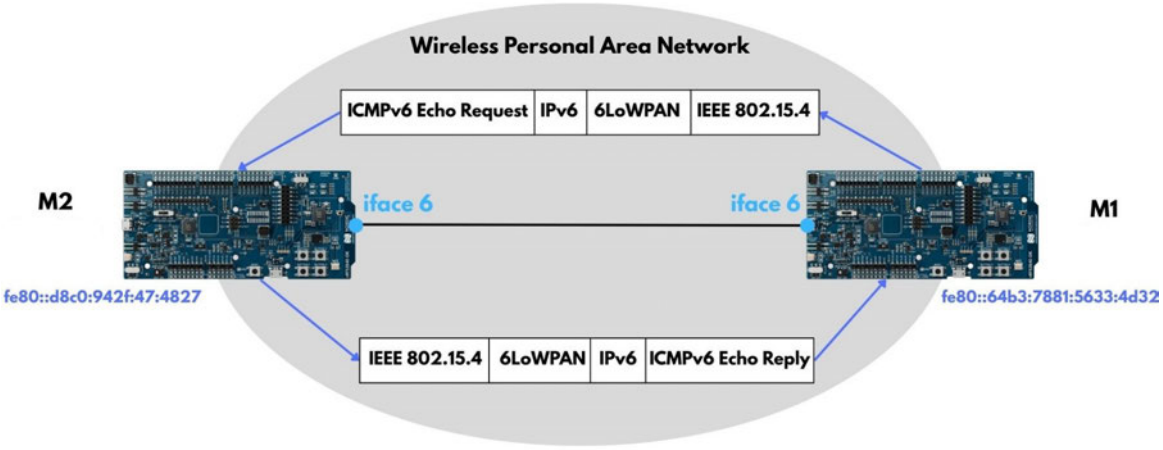


Figura 4.6: Escenario de conectividad IPv6 entre dos motas IoT

Esta figura refleja que, en primer lugar, la mota M1 construye un paquete *ICMPv6 Echo Request* con destino M2. Una vez recibe el mensaje anterior, M2 responde con un *ICMPv6 Echo Reply*, lo que garantiza que M1 descubre a M2, verificando que la mota está disponible. El esquema sería análogo en sentido contrario, si M2 construyera la solicitud y M1 la respuesta. Juntando ambos casos, resulta entonces que las dos motas IoT se conocen de forma activa entre sí gracias a la transmisión de paquetes ICMPv6, lo que nos da la conectividad a nivel de red entre ellas.

Entrando en detalle acerca de este primer escenario, de la figura 4.6 queda claro que los dos dispositivos IoT están conectados al mismo enlace inalámbrico que define una Local Area Network (LAN) inalámbrica, por lo que resulta que las comunicaciones a nivel de red y aplicación no requieren el uso de RPL en este caso. En virtud del método Extended Unique Identifier-64 (EUI-64), a partir de su dirección MAC, cada mota genera una dirección IPv6 link-local en la red, con prefijo "fe80::/64", que queda asignada de forma automática. Estas direcciones solamente son válidas dentro del enlace compartido.

Recordemos que estamos suponiendo que la capa de enlace sigue el estándar IEEE 802.15.4, mientras que el uso de IPv6 en la capa de red viene soportado por 6LoWPAN. Por otra parte, no es restrictivo suponer la siguiente asignación de direcciones link-local, ya indicada en la figura asociada al escenario:

	Dirección IPv6 link-local
M1	fe80::64b3:7881:5633:4d32
M2	fe80::d8c0:942f:47:4827

Cuadro 4.1: Direcciones IPv6 link-local de las motas M1 y M2

En vista de lo anterior, y bajo la hipótesis de que la dirección MAC de cada mota es conocida por la otra gracias al proceso de Neighbor Discovery [39], el flujo relativo a la construcción y transmisión de los paquetes ICMPv6, derivado de la figura 4.6, consta de los siguientes pasos:

1. **Preparación del paquete**: M1 crea un paquete *ICMPv6 Echo Request* con dirección IPv6 origen su link-local y dirección IPv6 destino la link-local de M2.

2. **Participación de 6LoWPAN**: 6LoWPAN encapsula el paquete, comprimiendo las cabeceras ICMPv6 e IPv6, y en caso de exceder los 127 bytes permitidos para un frame en el estándar IEEE 802.15.4, lo fragmenta.

3. **Transmisión radio**: el paquete 6LoWPAN resultante se transmite vía radio a través de la capa MAC que adopta IEEE 802.15.4.

4. **Recepción**: el paquete llega a M2, que lo reensambla si hubo fragmentación, y lo descomprime. Finalmente, reconstruye el paquete IPv6 e ICMPv6 y, como se trata de un mensaje *ICMPv6 Echo Request*, M2 envía un *ICMPv6 Echo Reply* en respuesta, que sigue los mismos pasos que la solicitud pero en sentido inverso, y cuya dirección IPv6 origen es la link-local de M2 y la destino es la link-local de M1.

Para satisfacer otro de nuestros objetivos e incorporar una comunicación a nivel de aplicación a través de CoAP, si suponemos que el cliente CoAP es la mota M1 y el servidor CoAP es la mota M2, el escenario de la figura 4.6 se extiende al siguiente:

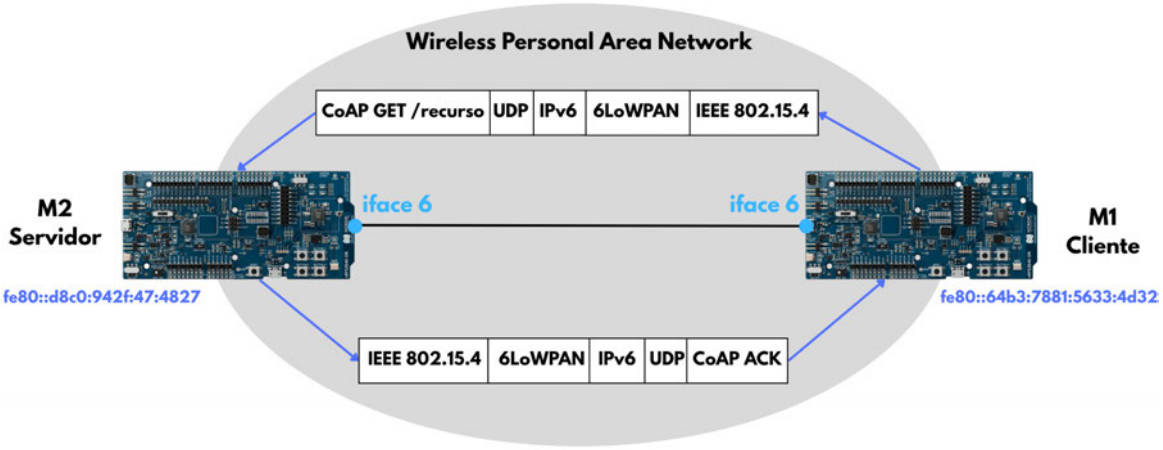


Figura 4.7: Escenario de conectividad IPv6 entre dos motas IoT extendido al uso de CoAP

Notemos que los mensajes CoAP se transportan con UDP sobre IPv6. Así, completamos el modelo de capas ilustrado en la figura 4.5 con los dos niveles superiores restantes (ahora ICMPv6 no entra en juego, como tampoco necesitábamos hacer uso de RPL desde el principio).

En lo que respecta a la aplicación CoAP que muestra la figura 4.7, su funcionamiento se basa en que, en primer lugar, se pone en marcha el servidor, el cual aloja una serie de recursos. Entonces, el cliente construye un mensaje CoAP, de tipo Confirmable, que resulta ser un GET por un cierto recurso, y lo envía al servidor.

Después, el servidor procesa la petición recibida, detectando que se trata de un GET por uno de los recursos disponibles, y responde con un CoAP ACK para confirmar que la solicitud se recibió con éxito. De hecho, este último mensaje tiene el mismo *Message ID* que la solicitud, e incluye además el código de respuesta y el payload del recurso pedido.

Por consiguiente, el flujo correspondiente al envío de una petición CoAP de tipo GET por un cierto recurso desde M1 al servidor en M2, que responde con un asentimiento, es:

1. **Envío de la solicitud CoAP**: M1 construye un paquete UDP con puerto destino 5683, el estándar de CoAP para un servidor en escucha. En el payload, se añade el tipo de solicitud y la ruta al recurso solicitado. A continuación, 6LoWPAN comprime las cabeceras IPv6 (donde la dirección IPv6 origen es la link-local de M1 y la destino es la link-local de M2), UDP y CoAP.

2. **Transmisión radio**: de nuevo, el paquete encapsulado se transmite por radio.

3. **Respuesta a la solicitud CoAP**: M2 descomprime el paquete, lo vuelve a construir y lo envía al servidor CoAP para que lo procese. El mensaje de respuesta que construye el servidor incluye el recurso pedido dentro del payload y, para su transmisión, este vuelve a encapsularse como la solicitud, permutando las direcciones IPv6 origen y destino en la cabecera IPv6.

Con estas explicaciones, hemos completado las consideraciones teóricas fundamentales que conciernen a este primer escenario.

4.4. Diseño del escenario entre una mota IoT y el PC

En el escenario anterior, dado que las motas compartían un mismo enlace de red, podían establecer una comunicación inalámbrica directamente a través de la red 6LoWPAN según el modelo de la figura 4.6.

Sin embargo, para que una mota y un PC hablen entre sí, el enfoque a adoptar cambia drásticamente, pues por un lado tenemos que la mota se comunica de forma inalámbrica y pertenece a una red WPAN, pero por otra parte el PC se sitúa en una red IPv6 y se comunica en la capa de enlace mediante Ethernet vía cable. Por consiguiente, como se recalcó al presentar los objetivos, resulta indispensable que aparezca un tercer dispositivo que actúe de router, en particular de Border Router.

Un Border Router es un equipo cuya función principal consiste en encaminar los paquetes entre una Personal Area Network (PAN) inalámbrica y una red IPv6 estándar (por ejemplo, Internet). Para ello, requiere dos interfaces: una interna, de baja potencia, que lo conecte con la mota a través de la red 6LoWPAN, y otra externa, que sea una interfaz Ethernet que lo conecte a la red IPv6 principal donde se encuentra el PC. Observemos que este router actúa de gateway para el dispositivo IoT hacia la red IPv6 pública donde se encuentra el PC.

Por tanto, el escenario objetivo por el cual obtenemos la conectividad a nivel de red entre la mota IoT y el PC se ilustra a partir del siguiente diagrama:

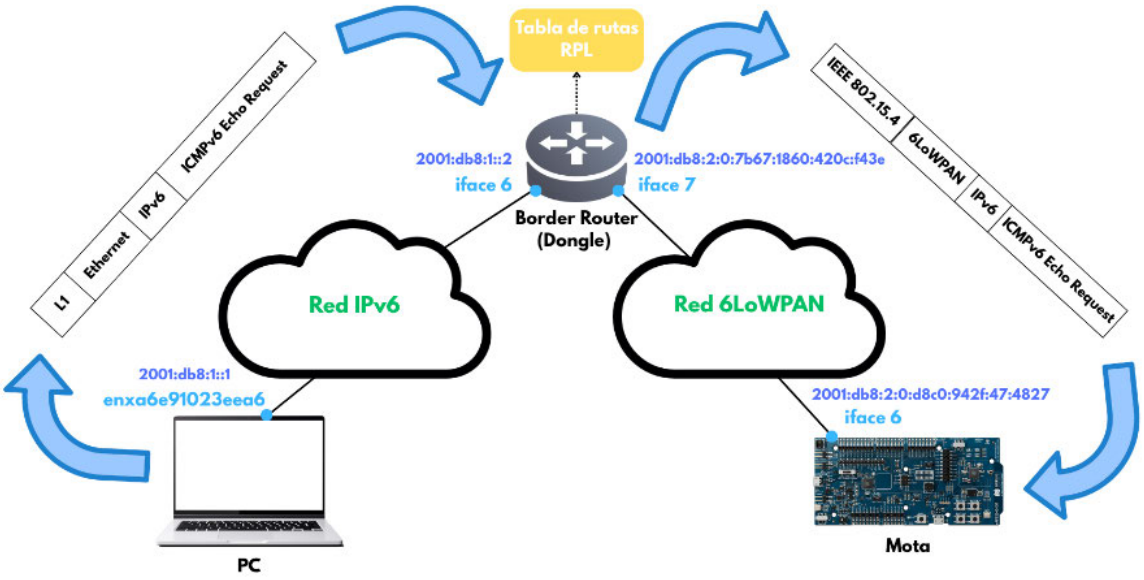


Figura 4.8: Escenario de conectividad IPv6 entre una mota IoT y el PC

Notemos que los componentes se disponen con el PC y la mota IoT como extremos de la comunicación, mientras que el Border Router es el elemento intermedio que tiene conectividad por medio de una interfaz con el PC a través de Ethernet y por otra con la mota vía radio. Si consideramos el Border Router como otra mota, observemos que este segundo enlace refleja el escenario de la figura 4.6, pues tenemos dos dispositivos que comparten una PAN (una red 6LoWPAN) y se comunican a través de sus interfaces inalámbricas.

Puesto que la comunicación va más allá del enlace local entre la mota y el Border Router, es necesario hacer uso de direcciones IPv6 globales, utilizadas por el router para reenviar los paquetes entre las distintas redes. Supondremos que cada red tiene dos prefijos IPv6 distintos, 2001:db8:1::/64 para la red IPv6, mientras que 2001:db8:2::/64 para la red 6LoWPAN inalámbrica.

Además, sin pérdida de generalidad, podemos considerar los siguientes valores para las interfaces y direcciones de cada dispositivo, que se muestran en la figura 4.8:

- Interfaz cableada del PC: *enxa6e91023eea6*, con dirección IPv6 global 2001:db8:1::1.
- Interfaz cableada del Border Router: *iface 6*, con dirección IPv6 global 2001:db8:1::2.
- Interfaz inalámbrica del Border Router: *iface 7*, con dirección IPv6 global 2001:db8:2:0:7b67:1860:420c:f43e.
- Interfaz inalámbrica de la mota: *iface 6*, con dirección IPv6 global 2001:db8:2:0:d8c0:942f:47:4827.

Sabemos que los paquetes IPv6 originados por la mota con destino el PC, o los construidos por el PC hacia la mota, deben ser enrutados por el Border Router. Este equipo traslada la comunicación a través de IEEE 802.15.4 en la red 6LoWPAN a una comunicación vía Ethernet en la red IPv6 o viceversa. No obstante, para que los mensajes transmitidos en este segundo escenario lleguen a su destino, en la interfaz inalámbrica del Border Router se debe inicializar el protocolo RPL, pues 6LoWPAN por sí mismo no define cómo se enrutan los paquetes entre el Border Router y la mota IoT. Por tanto, la integración de RPL en la red 6LoWPAN es vital para que los nodos de esta red (como la mota IoT) sepan establecer una ruta hacia el Border Router.

Así, la lógica detrás del uso de RPL se basa en que la mota sea capaz de calcular su dirección IPv6 global y esta pertenezca al mismo prefijo que la dirección global de la interfaz inalámbrica del Border Router, lo que asegura la comunicación entre estos componentes. En concreto, el dispositivo que ejerce el rol de nodo raíz del árbol DODAG de RPL debe ser el Border Router, pues es el responsable de dar salida hacia fuera y, en consecuencia, de transmitir periódicamente los DIOs. Estos objetos contienen información como el prefijo de la red o la dirección IPv6 del nodo raíz. La mota (o cualquier otro equipo conectado a la red inalámbrica compartida con el Border Router) escucha estos DIOs y se une al árbol (en otras palabras, a la red que se monta con RPL). Entonces, a partir del prefijo indicado en el DIO y del identificador de su interfaz, calcula automáticamente la que será su dirección IPv6 global, dentro del prefijo correspondiente.

En este segundo escenario, otro factor determinante que asegura la correcta recepción de los mensajes es la compresión/descompresión 6LoWPAN por parte del Border Router. Por un lado, la compresión permite que los paquetes que se dirigen a la mota puedan viajar de manera inalámbrica por la red 6LoWPAN, mientras que la descompresión es clave para que los paquetes construidos por la mota lleguen al PC a través de la red IPv6. En ningún momento, el encabezado IP del envío o de la respuesta varían, siguen siendo los mismos (solamente se comprimen o descomprimen). Sin embargo, lo que sí cambia es la dirección MAC origen y destino del paquete en la capa de enlace.

A partir de las consideraciones anteriores, estamos en disposición de explicar el flujo asociado a la figura 4.8, en el que el PC envía un mensaje *ICMPv6 Echo Request* a la mota. Al igual que en el escenario anterior, el envío de los mensajes ICMPv6 de un extremo de la comunicación al otro nos permite verificar la conectividad IPv6 y es análogo en ambos sentidos.

1. **Preparación del paquete**: el PC construye el paquete *ICMPv6 Echo Request* con dirección IPv6 origen la de su interfaz cableada y dirección IPv6 destino la de la interfaz inalámbrica de la mota.

2. **Enrutamiento por el Border Router**: en el paso anterior, el PC detecta que la dirección destino no está en su subred (2001:db8:2::/64 $\neq$ 2001:db8:1::/64), así que decide transmitir el paquete al Border Router vía gateway por una de sus interfaces cableadas.

   Una vez el Border Router recibe el paquete, consulta su tabla de rutas y descubre que hay salida a la subred 2001:db8:2::/64 (donde está la mota) a través de la red RPL, que se establece previamente sobre la red inalámbrica compartida con la mota. Entonces, comprime las cabeceras ICMPv6 e IPv6 con 6LoWPAN y transmite el paquete resultante vía radio según el estándar IEEE 802.15.4.

3. **Recepción**: el paquete llega a la mota por su interfaz inalámbrica y se descomprime. Entonces, la mota procesa el mensaje *ICMPv6 Echo Request*, que tenía de IP origen la dirección de la interfaz cableada del PC, y construye un *ICMPv6 Echo Reply* con dicha dirección como destino y con dirección origen la de su interfaz inalámbrica.

4. **Enrutamiento por el Border Router**: como la dirección destino del paquete de respuesta no pertenece a la subred 6LoWPAN, el mensaje *ICMPv6 Echo Reply* se tiene que transmitir al Border Router, para lo que se aplica de nuevo compresión con 6LoWPAN. Tras recibir dicho mensaje en su interfaz inalámbrica, el Border Router descomprime el paquete 6LoWPAN y reenvía el paquete IPv6 vía cable al PC. Esto conduce al final del proceso, pues da lugar a que el paquete ICMPv6 llegue al PC y este lo procese.

Una forma de asociar los pasos descritos a una imagen visual es a través de un diagrama de flujo para los mensajes *ICMPv6 Echo Request* e *ICMPv6 Echo Reply*.

A continuación, ilustramos un diagrama de ejemplo, donde indicamos las direcciones a nivel de red y de enlace que toma cada paquete en todo momento de la comunicación para reflejar así el comportamiento derivado de la compresión/descompresión 6LoWPAN que mencionamos anteriormente.

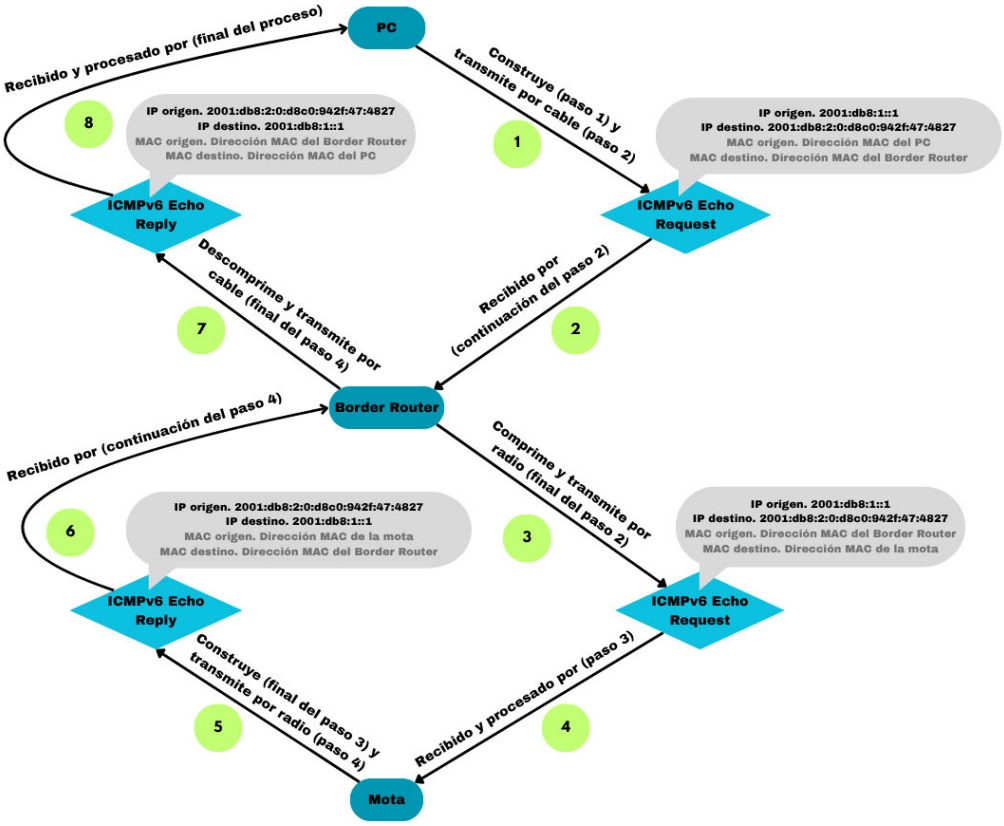


Figura 4.9: Diagrama de flujo de la transmisión de los mensajes *ICMPv6 Echo Request* e *ICMPv6 Echo Reply* entre la mota IoT y el PC

Para lograr la conectividad a nivel de aplicación con CoAP, este proceso extiende el despliegue de la figura 4.8, similarmente a como ocurría en el escenario entre motas IoT. En particular, tenemos:

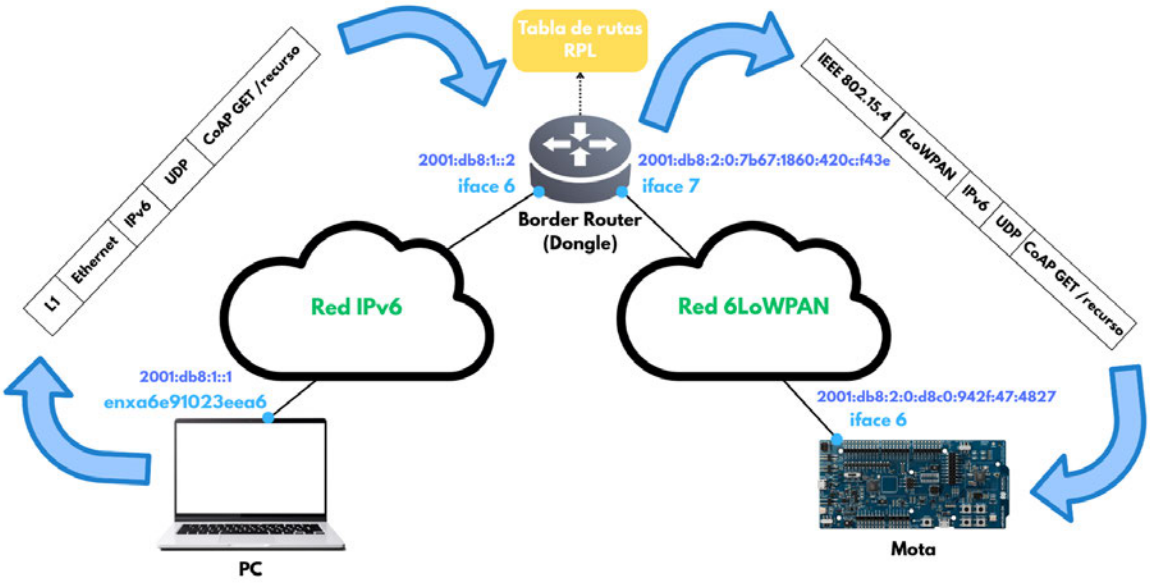


Figura 4.10: Escenario de conectividad IPv6 entre una mota IoT y el PC extendido al uso de CoAP

Sin perder generalidad, podemos considerar para este escenario la misma aplicación CoAP que explicamos para el diagrama de la figura 4.7. La única diferencia reside en que el cliente está ahora situado en el PC. Por tanto, el flujo de mensajes correspondiente al envío de una solicitud CoAP por parte del PC a la mota, que actúa de servidor CoAP, es:

1. **Envío de la solicitud CoAP**: el PC construye la petición por un recurso determinado y esta se encapsula sobre UDP e IPv6, con puerto destino 5683 y dirección IP destino la de la interfaz inalámbrica de la mota.

   De nuevo, las subredes de la dirección origen y destino no coinciden, por lo que el paquete se transmite vía Ethernet hacia la MAC del Border Router.

2. **Enrutamiento por el Border Router**: a continuación, el Border Router aplica compresión 6LoWPAN para las cabeceras IPv6 y UDP. La dirección IP destino sigue siendo la de la interfaz inalámbrica de la mota, pero esta ahora sí es alcanzable pues el Border Router tiene una entrada para ella en su tabla de rutas. Por otro lado, el mensaje CoAP permanece intacto, y el paquete completo se transmite a nivel de enlace por IEEE 802.15.4 vía radio.

3. **Recepción**: la mota contesta al Border Router con un CoAP ACK, y en el payload de la respuesta se encuentra el recurso solicitado. Las cabeceras UDP e IPv6 de este mensaje (donde la IP destino es la de la interfaz cableada del PC) se comprimen de nuevo con 6LoWPAN para que viajen hacia el Border Router por la red inalámbrica que soporta RPL.

4. **Enrutamiento por el Border Router**: finalmente, el Border Router descomprime la cabecera 6LoWPAN y reenvía la respuesta CoAP originada por la mota a la dirección IP destino correspondiente a la interfaz cableada del PC, finalizando el intercambio.

4.5. Montaje preliminar en RIOT

Gracias a las explicaciones anteriores, ya tenemos una idea de cómo funciona la comunicación a nivel de red y aplicación en los dos escenarios planteados. Para probarla en la práctica, describiremos el proceso previo a la implementación de sendos escenarios sobre el sistema operativo RIOT. En concreto, introduciremos el material hardware utilizado y las aplicaciones software de RIOT que se han considerado para satisfacer los objetivos propuestos. Después, especificaremos las tecnologías que han de instalarse en el PC para compilar las aplicaciones de RIOT en los dispositivos Nordic empleados y el despliegue de estas para tener un entorno de partida sobre el que empezar a trabajar con RIOT.

4.5.1. Material hardware utilizado

• El **nRF52840 DK** [12] es un kit de desarrollo compuesto por una sola placa, diseñado por Nordic para redes sujetas a los estándares IEEE 802.15.4. Incorpora un depurador SEGGER *J-Link* que se utiliza para programar y depurar tanto el System-on-a-Chip (SoC) integrado como los dispositivos externos.

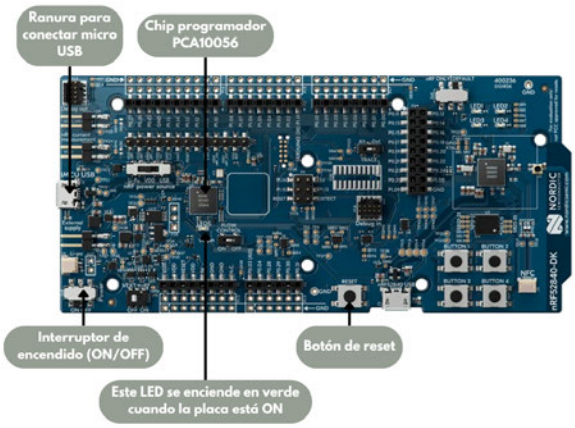


Figura 4.11: Componentes fundamentales del nRF52840 DK

• El **nRF52840 Dongle** [13] es un dispositivo USB compatible con todos los estándares inalámbricos de corto alcance utilizados en los dispositivos Nordic (entre ellos, IEEE 802.15.4), que no ofrece depuración. Fue diseñado para usarse junto con *nRF Connect for Desktop*, aunque también se puede programar mediante *nRF Util*.

Asimismo, cuenta con un LED Red, Green, Blue (RGB) programable por el usuario. Cuando este está en rojo, lo que ocurre al pulsar el botón de RESET, el Dongle se pone en modo Device Firmware Update (DFU), lo cual indica que está listo para ser restaurado y recibir un nuevo firmware.

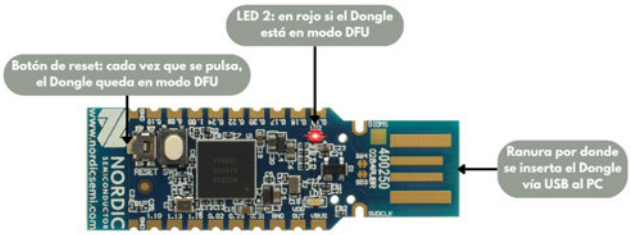


Figura 4.12: Componentes fundamentales del nRF52840 Dongle

• Se han utilizado **cables micro USB 2.0** con el fin de alimentar los nRF52840 DK, cuya fuente de energía se deja por defecto a Voltage Digital Drain (VDD), indicando tensión de alimentación positiva.

4.5.2. Aplicaciones software consideradas

• **gnrc_border_router:** en [40], se describe cómo configurar un Border Router en RIOT por medio de Communication Device Class-Ethernet Control Model (CDC-ECM) [41], una especificación del estándar USB CDC que define cómo se conectan dispositivos de red a máquinas host. En particular, CDC-ECM se centra en redes de tipo Ethernet, permitiendo el intercambio de datos a través de USB independientemente de la aplicación, así como abstrae a los desarrolladores de aplicaciones USB a nivel de red mediante el uso de sockets o HTTP.

Por tanto, con CDC-ECM establecemos una conexión Ethernet entre un dispositivo USB, que en nuestro caso es el Dongle, y el PC, de forma que el primero pueda enviar datos a través de un enlace uplink al segundo. En otras palabras, el Border Router (instalado en el Dongle) posee un enlace CDC-ECM con el PC, y este último simplemente es el nodo al otro extremo de dicho enlace (de modo que el PC puede actuar de gateway o de proveedor de red para el Dongle).

A continuación, describimos las modificaciones más relevantes al "Makefile" del ejemplo "gnrc_border_router" de RIOT [42] que se realizan en [40]. En primer lugar, se añaden dos interfaces GNRC con `GNRC_NETIF_NUMOF := 2`. El "Makefile" original soporta múltiples tipos de enlaces uplink, sin embargo, en nuestro caso solo nos interesa el enlace CDC-ECM. Por ello se añaden los dos módulos indicados con flechas verdes en esta imagen:

```
APPLICATION = usb_cdc_ecm_border_router
BOARD ?= samr21-xpro
RIOTBASE ?= $(CURDIR)/../..
GNRC_NETIF_NUMOF := 2

USEMODULE += usbus_cdc_ecm
USEMODULE += auto_init_usbus

USEMODULE += gnrc_netdev_default
USEMODULE += auto_init_gnrc_netif
USEMODULE += gnrc_icmpv6_error
USEMODULE += gnrc_sixlowpan_border_router_default
USEMODULE += gnrc_udp
USEMODULE += gnrc_rpl
USEMODULE += auto_init_gnrc_rpl
USEMODULE += gnrc_icmpv6_echo
USEMODULE += shell
USEMODULE += shell_commands
# Optional, but might be useful
USEMODULE += ps
USEMODULE += netstats_l2
USEMODULE += netstats_ipv6
USEMODULE += netstats_rpl

USB_VID ?= 1915 # Replace with your vendor ID
USB_PID ?= 521f # Replace with your product ID

CFLAGS += -DUSB_CONFIG_VID=0x$(USB_VID) -DUSB_CONFIG_PID=0x$(USB_PID)

DEVELHELP ?= 1

include $(RIOTBASE)/Makefile.include
```

Figura 4.13: Fichero "Makefile" de la aplicación "gnrc_border_router" considerada

Por otra parte, en la nueva versión se incluyen los siguientes módulos de red: gnrc_udp, gnrc_rpl, auto_init_gnrc_rpl, gnrc_icmpv6_error y netstats_l2, netstats_ipv6, netstats_rpl. Estos implementan UDP, RPL, mensajes de error ICMPv6 y estadísticas de red. Además, se elimina la lógica relativa al soporte de Domain Name System (DNS) [43] y Dynamic Host Configuration Protocol (DHCP) [44]. Al final del "Makefile" (en las flechas rojas de la figura 4.13), se configura el Dongle que se usará como Border Router por medio de su Vendor ID y el Product ID. Esto es necesario para establecer el enlace CDC-ECM vía USB.

Así, el resultado conseguido es el despliegue de un Border Router 6LoWPAN con conectividad global a través de Internet, que actúa de gateway para un dispositivo IoT y permite que este se comunique con el PC. Esta funcionalidad es precisamente la que queremos que tenga el Border Router de acuerdo a lo descrito en el segundo escenario.

- **gcoap:** la aplicación "gcoap" [45] proporciona acceso de línea de comandos a la Application Programming Interface (API) de alto nivel para mensajería CoAP del mismo nombre. Implementa tanto un cliente como un servidor CoAP, este último capaz de recibir solicitudes desde otros equipos como el PC.

Para poder utilizar esta versión de CoAP, en el "Makefile" hemos de incluir el módulo RPL, así la mota que actúe de servidor podrá configurarse una dirección IPv6 global, basada en el prefijo de la red inalámbrica (2001:db8:2::/64) compartido con el Border Router. Tras incluir el módulo `netdev_default`, podemos introducir las líneas:

```
USEMODULE += gnrc_rpl
USEMODULE += auto_init_gnrc_rpl
```

El cliente CoAP se implementa en el fichero *client.c*, donde encontramos el código mediante el cual el usuario puede enviar comandos GET, PUT o POST desde la terminal de RIOT. Por su parte, el servidor se sitúa en *server.c*. Para indicar que la petición por un recurso se ha recibido correctamente, facilitamos la depuración de la aplicación añadiendo una sentencia `printf` en este último fichero:

```
static ssize_t _riot_board_handler(coap_pkt_t *pdu, uint8_t *buf, size_t len,
coap_request_ctx_t *ctx)
{
    (void)ctx;
    printf("Solicitud recibida en /riot/board\n");
    gcoap_resp_init(pdu, buf, len, COAP_CODE_CONTENT);
    coap_opt_add_format(pdu, COAP_FORMAT_TEXT);
    size_t resp_len = coap_opt_finish(pdu, COAP_OPT_FINISH_PAYLOAD);

    /* write the RIOT board name in the response buffer */
    if (pdu->payload_len >= strlen(RIOT_BOARD)) {
        memcpy(pdu->payload, RIOT_BOARD, strlen(RIOT_BOARD));
        return resp_len + strlen(RIOT_BOARD);
    }
```

Figura 4.14: Modificación en el fichero *server.c* de la aplicación "gcoap"

• **libcoap_server y libcoap_client:** la prueba del escenario entre dos motas IoT también se realizará usando la implementación de CoAP conocida como libCoAP. A pesar de que en las versiones recientes de RIOT esté obsoleta, libCoAP todavía se encuentra disponible en una rama alternativa del repositorio [46]. En ella, los ejemplos "libcoap_client" y "libcoap_server" tienen configurados un cliente y servidor libCoAP, de los que podemos hacer uso sin ningún cambio adicional en los ficheros que conforman la aplicación.

Esta implementación de CoAP ha sido incluida porque está diseñada para soportar Object Security for Constrained RESTful Environments (OSCORE) [47], un protocolo que aplica seguridad a nivel de objeto a los datos transmitidos en cada mensaje CoAP mediante cifrado, autenticación y protección de integridad. No obstante, la aplicación de este método queda fuera de los aspectos que cubre este trabajo, dado que la integración de OSCORE en RIOT es un problema que aún no ha alcanzado una solución estándar y funcional en la gran mayoría de casos.

Si bien existen bibliotecas como libOSCORE [48] que pueden integrarse en el proceso de compilación de RIOT, su uso junto a CoAP es propenso a causar errores innecesarios o a aumentar la carga de mantenimiento de la aplicación. Esto ocurre porque cada aplicación en RIOT que depende de OSCORE debe depender a su vez de una implementación personalizada de OSCORE y no puede acceder a la pila OSCORE a través de una interfaz genérica común. En consecuencia, la aplicación necesita manejar todo el código intermedio complejo entre la implementación de OSCORE y la API CoAP de RIOT.

Idealmente, la solución a esta cuestión sería que la integración de OSCORE en RIOT fuera tal que las aplicaciones usen CoAP de manera normal, mientras obtengan soporte de OSCORE automáticamente [49]. De esta manera, no sería difícil actualizar o reemplazar dependencias para obtener nuevas capacidades de seguridad, como algoritmos de cifrado actualizados.

4.5.3. Tecnologías

Con respecto a las tecnologías usadas para llevar a cabo el montaje de los dos escenarios y los procesos de compilación y ejecución de las aplicaciones sobre el material hardware, cabe destacar las siguientes.

• **Dependencias de RIOT:** existe un conjunto de herramientas comunes [50] que son requeridas y/o recomendables para compilar RIOT, independientemente de la arquitectura y la placa sobre las que se trabaje. Una breve descripción de cada una de ellas se encuentra en el anexo I. Un aspecto que es importante señalar es que el sistema operativo anfitrión elegido sobre el que se ha instalado RIOT ha sido una imagen de escritorio de Ubuntu 22.04.5, pues, como mencionamos anteriormente, los sistemas de tipo Linux son los más recomendados para trabajar con RIOT u otros entornos destinados a IoT similares.

• **nRF Util:** *nRF Util* [51] es una herramienta de línea de comandos que permite a los usuarios administrar dispositivos Nordic Semiconductor, dando soporte a su descubrimiento, programación, borrado, restablecimiento o recuperación. Entre las diversas operaciones que ofrece, instala y supervisa toolchains para el *nRF Connect Software Development Kit (nRF Connect SDK).*

• **J-Link:** *J-Link* [52] constituye la opción más popular para optimizar la depuración y la programación flash. Funciona sobre una amplia gama de CPUs y arquitecturas, así como sobre los principales Integrated Development Environments (IDEs), como el SEGGER Embedded Studio. Se caracteriza por una rápida velocidad de descarga y por permitir transferencia en tiempo real y puntos de interrupción ilimitados en memoria flash.

• **nRF Connect for Desktop:** *nRF Connect for Desktop* [53] es un framework multiplataforma que facilita el desarrollo en dispositivos nRF. Proporciona funcionalidades destinadas a la prueba, monitorización o programación de aplicaciones. A modo de ejemplo, la aplicación "Programmer" se emplea para programar SoC de Nordic, arrastrando y soltando archivos para leer, escribir o borrar el dispositivo.

• **Wireshark:** *Wireshark* [54] es una herramienta para la captura y rastreo de paquetes que permite la escucha de una conexión en tiempo real desde múltiples interfaces, la visualización de flujos de red completos y el análisis de los datos obtenidos mediante diferentes filtros.

• **nRF Sniffer for 802.15.4:** *nRF Sniffer for 802.15.4* [55] es otra herramienta multiplataforma que utiliza *Wireshark* como interfaz y tiene soporte para sistemas operativos Linux, Windows y macOS. Ofrece visualización en tiempo real de lo que sucede en protocolos que adoptan el estándar IEEE 802.15.4. El único hardware sobre el que se aplica es un nRF52840 DK o un nRF52840 Dongle.

• **PySerial:** *PySerial* [56] es una biblioteca que da soporte al establecimiento de comunicaciones serie en Python, encapsulando el acceso al puerto correspondiente. Maneja APIs de cara al envío y recepción de datos. La diferencia con `python3-serial` consiste en que este último lo gestiona el sistema de paquetes `apt` en lugar de `pip`.

• **libCoAP:** *libCoAP* [37] es una implementación en el lenguaje C del protocolo CoAP para dispositivos con recursos limitados. Es compatible con múltiples plataformas y proporciona las funciones principales para el desarrollo de servidores y clientes CoAP eficientes. Por ejemplo, `coap-client` es una herramienta similar a `wget` que genera solicitudes sencillas para la recuperación y modificación de recursos en un servidor remoto. Mientras, `coap-server` ilustra diversas funciones del lado del servidor libCoAP.

4.5.4. Despliegue previo

El siguiente paso para llevar a cabo la puesta en marcha de los escenarios es que el sistema operativo anfitrión esté preparado para compilar las aplicaciones de RIOT e interactuar con los dispositivos nRF52840 mencionados. Básicamente, este proceso consiste en incorporar las tecnologías que hemos especificado en la sección anterior. Lo ilustramos a continuación:

1. Partiendo de la imagen de Ubuntu 22.04.5 instalada a través de la configuración normal, desde una terminal instalamos primero las dependencias necesarias para RIOT indicadas al inicio de la sección 4.5.3.

   ```
   sudo apt install git gcc-arm-none-eabi make gcc-multilib
   libstdc++-arm-none-eabi-newlib openocd gdb-multiarch doxygen
   wget unzip python3-serial python3-psutil
   ```

2. Seguidamente, clonamos el repositorio de RIOT. Utilizaremos la versión de los desarrolladores Oleg Hahm y Martine Lenders que forma parte de los tutoriales de aprendizaje de RIOT [32], pues es la recomendada para aquellos usuarios que buscan tener una primera toma de contacto con este sistema operativo.

   ```
   git clone --recursive https://github.com/RIOT-OS/Tutorials
   ```

3. Para cargar el firmware de la aplicación "gnrc_border_router" en el Dongle, instalamos *nRF Util* desde el repositorio de Nordic Semiconductor [57]. Antes, nos aseguramos de tener operativos *Python3* y *Python3-pip*.

   ```
   sudo apt install python3 python3-pip -y
   ```

   Entonces, clonamos el repositorio de *nRF Util* y nos movemos a la carpeta donde lo hayamos alojado.

   ```
   git clone https://github.com/NordicSemiconductor/pc-nrfutil
   cd pc-nrfutil
   ```

   Dentro de dicha carpeta, cargamos todas las dependencias requeridas para el correcto funcionamiento de la herramienta.

   ```
   sudo pip3 install -r requirements.txt
   ```

   Finalmente, incorporamos también el paquete en el sistema.

   ```
   sudo python3 setup.py install
   ```

   Tras esto, observemos que *nRF Util* se mueve al directorio `/usr/local/bin` incluido en la variable de entorno PATH.

   ```
   iab@iab:~$ cd /usr/local/bin
   iab@iab:/usr/local/bin$ ls
   bin2hex.py     extcap_ot.py  hex2dump.py  hexinfo.py   nrfutil      pyserial-miniterm  sniffer.py     tqdm
   extcap_ot.bat  hex2bin.py    hexdiff.py   hexmerge.py  __pycache__  pyserial-ports     spinel-cli.py
   ```

   Figura 4.15: Localización de la herramienta *nRF Util* en el PC

4. Otro factor fundamental para interactuar con los dispositivos IoT es añadir nuestro usuario al grupo *dialout*.

   ```
   sudo usermod -a -G dialout $USER
   ```

   De esta manera, tendremos acceso a los puertos serie, en particular a aquellos de la forma "/dev/ttyACM*", con los que vamos a trabajar. Para que los cambios surtan efecto, se debe reiniciar el sistema posteriormente.

   Llegados a este punto, toda la funcionalidad necesaria para programar aplicaciones de RIOT en el nRF52840 Dongle ya ha sido desplegada.

5. Para los escenarios en los que el PC actúa como cliente CoAP, hemos de instalar desde los repositorios de Ubuntu un paquete que ofrezca esta funcionalidad. En particular, se ha optado por las versiones de desarrollo y binaria más recientes de libCoAP.

   ```
   sudo apt install libcoap3-dev libcoap3-bin
   ```

   Igualmente, para configurar las interfaces de red en el PC utilizaremos el comando `ifconfig`, que instalamos como sigue.

   ```
   sudo apt install net-tools
   ```

6. Por último, necesitamos el software de *J-Link* [58] para la programación y depuración de los nRF52840 DK.

La versión escogida de *J-Link* ha sido la 8.12e, aunque alternativamente otras de las versiones más recientes pueden utilizarse. Tras descargar el fichero de extensión ".deb", antes de instalar el paquete en el sistema, le otorgamos permisos de ejecución.

```
chmod +x JLink_Linux_V812e_x86_64.deb
sudo dpkg -i JLink_Linux_V812e_x86_64.deb
```

A pesar de que existen otras herramientas como el comando `nrfjprog` para borrar y recuperar la memoria del firmware que se carga en la placa, esta opción resulta mucho más sencilla pues la configuración se realiza automáticamente por parte del depurador SEGGER *J-Link*.

En este momento, conviene resaltar que, cuando en una de las motas tengamos cargada una aplicación de RIOT y queramos cargar otra diferente, en ocasiones esta nueva escritura se omite y la grabación del programa no se completa correctamente. La mejor práctica para evitar esta situación es hacer uso de *J-Link* y borrar antes la memoria que haya en la placa con el siguiente comando.

```
JLinkExe -device NRF52840_XXAA -if SWD -speed 4000 -autoconnect 1
```

Si tuviéramos dos motas conectadas, se abriría una ventana emergente para decidir cuál usar (pueden distinguirse por su número de serie).

Durante la ejecución del comando anterior, es altamente probable que aparezca la siguiente ventana emergente:

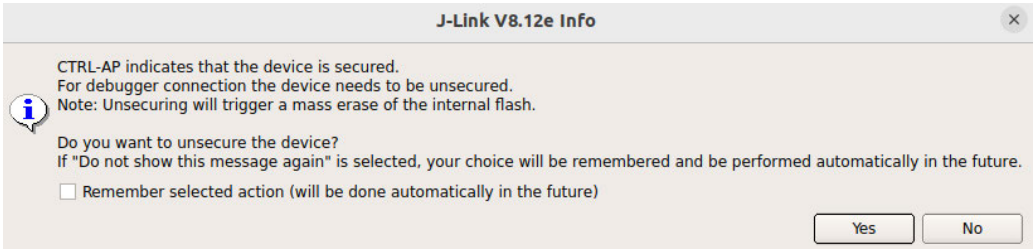


Figura 4.16: Aviso de memoria protegida en *J-Link*

Esta indica que el dispositivo hardware tiene la memoria protegida y solo podemos acceder a él si la desecurizamos (es decir, eliminamos la protección). Entonces, el proceso de borrado se completa satisfactoriamente tras darle a "Yes".

4.6. Montaje de los escenarios en RIOT

Gracias a las tecnologías instaladas en el PC, estamos en disposición de montar, con el material hardware/software descrito, los escenarios presentados y completar toda su configuración a falta de la realización de las pruebas de concepto. Distinguimos en particular:

• Escenario entre dos motas IoT con GCoAP → basado en lo descrito en la sección 4.3, utilizando como aplicación CoAP la que proporciona la aplicación "gcoap" de RIOT.

• Escenario entre una mota IoT y el PC con GCoAP → montado de acuerdo a las especificaciones vistas en la sección 4.4, usando las aplicaciones "gcoap" en el lado de la mota IoT y "gnrc_border_router" para el Dongle que actúa de Border Router.

• Escenario entre dos motas IoT con libCoAP → como el primer escenario, se basa en el diseño explicado en la sección 4.3, pero en este caso se utilizan las aplicaciones "libcoap_client" y "libcoap_server" de [46] para las motas cliente y servidor, respectivamente.

4.6.1. Montaje del escenario entre dos motas IoT con GCoAP

Procedemos entonces a describir paso a paso la forma de montar los tres escenarios anteriores, comenzando con el primero de ellos. En él, vamos a tener dos nRF52840 DK que representen las motas IoT y sobre las que se cargue el firmware de la aplicación "gcoap" de RIOT. A lo largo de este proceso y de las pruebas correspondientes, trabajaremos sobre dos terminales del PC que, tras compilar la aplicación en las placas, quedarán ligadas a las terminales de RIOT donde se desarrolle la funcionalidad de cliente y servidor CoAP, respectivamente.

Sabemos que el objetivo grosso-modo es realizar una petición CoAP por un determinado recurso desde la mota cliente a la mota servidora. Conviene indicar que el ejemplo "gcoap" da soporte al descubrimiento de los dispositivos mediante mensajes ICMPv6, por lo que a partir de él también podremos comprobar que hay conectividad IPv6 entre las motas y así reflejar fielmente los escenarios de las figuras 4.6 y 4.7. Este paso previo resulta intuitivo porque, sin él, no se podría elevar posteriormente la comunicación al nivel de aplicación.

Por tanto, ya podemos detallar todo el proceso de montaje desarrollado:

1. En primer lugar, conectamos los dos nRF52840 DK mediante los cables micro USB 2.0 al PC. Ponemos ambas placas a ON para encenderlas y que reciban alimentación del PC.

2. Cuando se trabaja con dispositivos conectados por USB a un sistema operativo, una buena práctica es consultar los puertos disponibles. Ya comentamos que nos interesa filtrar los de la forma "/dev/ttyACM*". Desde una terminal del PC, mostramos los puertos relativos a las dos motas conectadas:

```
lab@iab:~$ ls /dev/ttyACM*
/dev/ttyACM0  /dev/ttyACM1  /dev/ttyACM2  /dev/ttyACM3
```

Figura 4.17: Puertos disponibles al conectar los dos nRF52840 DK

   En la salida, distinguimos que los dos primeros corresponden a la primera mota que fue conectada y los otros dos a la siguiente.

3. En el sistema Ubuntu sobre el que trabajamos, abrimos una terminal que vamos a ligar a una de las placas sobre la que carguemos "gcoap" y que representará al cliente CoAP. Nos movemos para ello al directorio de la aplicación "gcoap" dentro del repositorio de RIOT instalado (este es `/Tutorials/RIOT/examples/gcoap`) y ejecutamos:

   ```
   make BOARD=nrf52840dk flash
   ```

   Antes de introducir el comando, se recomienda apagar una de las placas (por ejemplo, la segunda en haberse conectado), pues la compilación y carga del firmware de la aplicación solo se lleva a cabo en una de ellas y así es más fácil identificar cuál es. El proceso de compilación genera un binario de extensión ".hex", que se transfiere a la placa mediante el depurador SEGGER de *J-Link*, y la salida resultante es la siguiente:

```
Downloading file [/home/lab/Tutorials/RIOT/examples/gcoap/bin/nrf52840dk/gcoap_example.elf]
Comparing flash   [100%] Done.
Erasing flash     [100%] Done.
Programming flash [100%] Done.
J-Link: Flash download: Bank 0 @ 0x00000000: 1 range affected (98304 bytes)
J-Link: Flash download: Total: 1.162s (Prepare: 0.063s, Compare: 0.010s, Erase: 0.000s, Prc
J-Link: Flash download: Program & Verify speed: 92 KB/s
O.K.
J-Link>r
Reset delay: 0 ms
Reset type: NORMAL (https://wiki.segger.com/J-Link_Reset_Strategies)
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
J-Link>g
Memory map 'after startup completion point' is active
J-Link>exit

Script processing completed.
```

Figura 4.18: Final del proceso de compilación de la aplicación "gcoap" sobre un nRF52840 DK con *J-Link*

Para asociar la otra terminal del PC al shell de "gcoap" que consideraremos para el servidor CoAP en RIOT, ejecutamos de nuevo el comando anterior, pero ahora apagamos la placa que ya está cargada y trabajamos solo con la que antes se apagó. Así, conseguimos flashear el firmware en las dos motas y ya podemos interactuar con ambas encendidas.

4. A continuación, desde la primera terminal asociada al primer DK conectado (que está ligado al puerto "/dev/ttyACM0", aunque también podría usarse "/dev/ttyACM1" de acuerdo a la salida de la figura 4.17), abrimos la terminal serie de RIOT de "gcoap" con *pyterm*.

   ```
   make BOARD=nrf52840dk PORT=/dev/ttyACM0 term
   ```

   Como el puerto indicado es el que se conecta por defecto, podría omitirse la directiva `PORT` del comando anterior. Tras este paso, ya tenemos configurada una de las motas y estamos dentro de una terminal con un shell accesible desde el que podemos interactuar directamente con la aplicación "gcoap" y ejecutar diversos comandos. De cara a acceder a la funcionalidad de la mota que actúa de servidor, basta introducir el siguiente comando:

   ```
   make BOARD=nrf52840dk PORT=/dev/ttyACM2 term
   ```

   Nótese cómo hemos elegido uno de los puertos asociados a la segunda placa, así evitamos conflictos con el DK que ya está operativo.

5. Tanto en la terminal del cliente como en la del servidor, podemos visualizar los comandos que hay disponibles en el shell mediante `help`:

   ```
   help
   2025-03-24 12:01:46,829 # help
   2025-03-24 12:01:46,832 # Command              Description
   2025-03-24 12:01:46,835 # ---------------------------------------
   2025-03-24 12:01:46,839 # coap                 CoAP example
   2025-03-24 12:01:46,843 # 6ctx               6LoWPAN context configuration tool
   2025-03-24 12:01:46,847 # ifconfig           Configure network interfaces
   2025-03-24 12:01:46,852 # nib                Configure neighbor information base
   2025-03-24 12:01:46,856 # ping               Ping via ICMPv6
   2025-03-24 12:01:46,860 # pm                 interact with layered PM subsystem
   2025-03-24 12:01:46,866 # ps                 Prints information about running threads.
   2025-03-24 12:01:46,869 # reboot             Reboot the node
   2025-03-24 12:01:46,876 # rpl                rpl configuration tool ('rpl help' for more information)
   2025-03-24 12:01:46,880 # version            Prints current RIOT_VERSION
   ```

   Figura 4.19: Comandos disponibles en la aplicación "gcoap"

   De la salida resultante, podemos identificar la orden `ifconfig`, utilizada para conocer la configuración de las interfaces de red de las motas; mensajes `ping`, que verifican la conectividad IPv6 equivalentemente al envío de mensajes *ICMPv6 Echo Request* e *ICMPv6 Echo Reply*; y el comando `coap`, gracias al cual podremos realizar consultas al servidor construyendo peticiones GET.

6. Para poder establecer las comunicaciones deseadas, hemos de conocer por medio de `ifconfig` las direcciones IPv6 link-local de las motas, que coinciden a propósito con las del cuadro 4.1. Por ejemplo, en la terminal del cliente tenemos:

   ```
   > ifconfig
   2025-03-24 12:08:02,622 # ifconfig
   2025-03-24 12:08:02,627 # Iface  6  HWaddr: 4D:32  Channel: 26  NID: 0x23  PHY: O-QPSK
   2025-03-24 12:08:02,631 #          Long HWaddr: 66:B3:78:81:56:33:4D:32
   2025-03-24 12:08:02,634 #           State: IDLE
   2025-03-24 12:08:02,639 #          ACK_REQ  L2-PDU:102  MTU:1280  HL:64  6LO
   2025-03-24 12:08:02,640 #          IPHC
   2025-03-24 12:08:02,644 #          Source address length: 8
   2025-03-24 12:08:02,646 #          Link type: wireless
   2025-03-24 12:08:02,652 #          inet6 addr: fe80::64b3:7881:5633:4d32  scope: link  VAL
   2025-03-24 12:08:02,655 #          inet6 group: ff02::1
   2025-03-24 12:08:02,657 #          inet6 group: ff02::1a
   ```

   Figura 4.20: Salida del comando `ifconfig` en la terminal de la mota cliente

Tras completar los pasos anteriores, tenemos todo preparado para que las motas IoT puedan intercambiar mensajes IPv6 y CoAP. De hecho, el servidor CoAP dado por la aplicación "gcoap" está en escucha por defecto desde el principio, como puede comprobarse ejecutando la orden `coap info`.

4.6.2. Montaje del escenario entre una mota IoT y el PC con gCoAP

Ahora es el turno de desplegar el segundo escenario. En primer lugar, recordemos cómo hemos identificado los tres componentes involucrados. Por un lado, el nRF52840 Dongle es el dispositivo USB que hace el rol de Border Router. Para ello, se le carga el firmware de la aplicación "gnrc_border_router" descrita en el apartado 4.5.2.

Al igual que en el montaje anterior, un nRF52840 DK se utiliza para representar una mota IoT y en él se flashea la aplicación "gcoap" de RIOT. En concreto, este dispositivo va a implementar la funcionalidad de servidor CoAP. Por otro lado, el PC ejerce de cliente CoAP gracias a los paquetes instalados en el paso 5 de la sección 4.5.4.

En este caso, necesitaremos trabajar desde tres terminales del PC. Una de ellas siempre se mantendrá ligada a este equipo, mientras que las otras dos, una vez completada la carga de las aplicaciones "gcoap" y "gnrc_border_router", corresponderán a las terminales de RIOT donde interactuaremos con el nRF52840 DK que hace de servidor CoAP y con el Dongle que ejerce de Border Router, respectivamente.

De este modo, de cara a montar comunicaciones a nivel de red con IPv6 y a nivel de aplicación con CoAP entre la mota y el PC, el procedimiento a seguir se resume en:

1. Primeramente, conectamos el Dongle al PC y anotamos el puerto serie al que queda asociado (en nuestro caso, este es el "/dev/ttyACM0", el primero por defecto). Para poder escribir sobre este dispositivo, nos aseguramos de que esté en modo DFU, presionando el botón RESET para ello si fuera necesario.

   Abrimos una terminal del PC que acabará asociada al shell de la aplicación del Border Router. Para ello, nos movemos al directorio dentro del repositorio de RIOT correspondiente a tal aplicación (este es, `/Tutorials/RIOT/examples/gnrc_border_router`) y ejecutamos:

   ```
   make BOARD=nrf52840dongle UPLINK=cdc-ecm flash
   ```

   Este comando compila y flashea el firmware de dicha aplicación sobre el Dongle. El uso de `UPLINK = cdc-ecm` indica que el Border Router utilizará la interfaz USB del Dongle para emular una conexión Ethernet con el PC.

   Así, el PC ve el Dongle como una tarjeta de red, permitiendo que el tráfico IPv6 de la red 6LoWPAN que compartirá el router con la mota IoT sea enrutado a través de esta conexión USB.

```
Zip created at /home/iab/Tutorials/RIOT/examples/gnrc_border_router/bin/nrf52840dongle/usb_cdc_ecm_border_router.hex.zip
stty -F /dev/ttyACM0 raw ispeed 1200 ospeed 1200 cs8 -cstopb ignpar eol 255 eof 255
sleep 1
nrfutil dfu usb-serial --port=/dev/ttyACM0 --package=/home/iab/Tutorials/RIOT/examples/gnrc_border_router/bin/nrf52840dongle/usb_cdc_ecm_border_router.hex.zip
  [####################################]  100%
Device programmed.
```

Figura 4.21: Resultado de la compilación de la aplicación "gnrc_border_router" en el Dongle

2. Como resultado del paso anterior, observamos la creación de una interfaz de red USB entre el Dongle y el PC. En la terminal asociada al PC, si hacemos `ifconfig`, encontramos esta nueva interfaz:

```
enxa6e91023eea6: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::2c5e:6f94:9e11:6bd0  prefixlen 64  scopeid 0x20<link>
        ether a6:e9:10:23:ee:a6  txqueuelen 1000  (Ethernet)
        RX packets 2  bytes 104 (104.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 2031 (2.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figura 4.22: Interfaz de red virtual generada por CDC-ECM entre el PC y el Border Router

Su nombre es precisamente el mismo que le asignamos a la hora de diseñar este escenario, con el fin de que el montaje en la práctica y el diseño teórico compartan no solo los valores específicos de las direcciones, sino también los nombres de las interfaces. Además, cabe destacar que esta es la interfaz por la que capturaremos el tráfico desarrollado en las pruebas desde *Wireshark*. El proceso de captura de paquetes lo comenzamos justo tras finalizar este paso.

3. A continuación, en la primera terminal que abrimos, tras haber flasheado el firmware de "gnrc_border_router", nos conectamos al terminal serie de esta aplicación para interactuar con el Border Router (o el Dongle) desde el shell de RIOT.

```
make BOARD=nrf52840dongle UPLINK=cdc-ecm term
```

La directiva `PORT` no se vuelve a incluir pues por defecto el puerto que se escoge es el "/dev/ttyACM0". Si introducimos la orden `help`, descubrimos los distintos comandos de los que podemos hacer uso en esta aplicación:

```
lab@lab:~/Tutorials/RIOT/examples/gnrc_border_router$ make BOARD=nrf52840dongle UPLINK=cdc-ecm term
Deprecated modules are in use: gnrc_netdev_default shell_commands
/home/iab/Tutorials/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
2025-03-23 16:51:06,860 # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
2025-03-23 16:51:07,864 # ented by driver
2025-03-23 16:51:07,865 # main(): This is RIOT! (Version: 2023.10)
2025-03-23 16:51:07,867 # RIOT border router example application
2025-03-23 16:51:07,868 # All up, running the shell now
> help
2025-03-23 16:51:11,026 # help
2025-03-23 16:51:11,027 # Command              Description
2025-03-23 16:51:11,029 # ---------------------------------------
2025-03-23 16:51:11,031 # 6ctx                 6LoWPAN context configuration tool
2025-03-23 16:51:11,031 # bootloader          Reboot to bootloader
2025-03-23 16:51:11,031 # ifconfig            Configure network interfaces
2025-03-23 16:51:11,032 # nib                 Configure neighbor information base
2025-03-23 16:51:11,032 # ping                Ping via ICMPv6
2025-03-23 16:51:11,032 # pm                  interact with layered PM subsystem
2025-03-23 16:51:11,033 # ps                  Prints information about running threads.
2025-03-23 16:51:11,033 # reboot              Reboot the node
2025-03-23 16:51:11,034 # rpl                 rpl configuration tool ('rpl help' for more information)
2025-03-23 16:51:11,034 # version             Prints current RIOT_VERSION
```

Figura 4.23: Comandos disponibles en la aplicación "gnrc_border_router"

De momento, nos interesa saber que, con `ifconfig`, contemplamos las dos interfaces con las que opera el Border Router: la inalámbrica (`iface 7`), mediante la cual se comunicará con la mota servidora, y la cableada (`iface 6`), por la que intercambiará mensajes con el PC. Ambas tienen preconfiguradas sus direcciones IPv6 link-local, así como un conjunto de grupos multicast a los que están suscritas.

```
ifconfig
2025-03-23 16:53:33,590 # ifconfig
2025-03-23 16:53:33,592 # Iface  6  HWaddr: A6:E9:10:23:F2:A6
2025-03-23 16:53:33,593 #          L2-PDU:1500  MTU:1500  HL:64  RTR
2025-03-23 16:53:33,595 #          Source address length: 6
2025-03-23 16:53:33,595 #          Link type: wired
2025-03-23 16:53:33,596 #          inet6 addr: fe80::a4e9:10ff:fe23:f2a6  scope: link  VAL
2025-03-23 16:53:33,597 #          inet6 group: ff02::2
2025-03-23 16:53:33,597 #          inet6 group: ff02::1
2025-03-23 16:53:33,598 #          inet6 group: ff02::1:ff23:f2a6
2025-03-23 16:53:33,598 #
2025-03-23 16:53:33,598 #          Statistics for Layer 2
2025-03-23 16:53:33,598 #            RX packets 87  bytes 15082
2025-03-23 16:53:33,599 #            TX packets 4 (Multicast: 4)  bytes 0
2025-03-23 16:53:33,599 #            TX succeeded 0 errors 0
2025-03-23 16:53:33,600 #          Statistics for IPv6
2025-03-23 16:53:33,600 #            RX packets 67  bytes 7684
2025-03-23 16:53:33,601 #            TX packets 4 (Multicast: 4)  bytes 224
2025-03-23 16:53:33,601 #            TX succeeded 4 errors 0
2025-03-23 16:53:33,601 #
2025-03-23 16:53:33,602 # Iface  7  HWaddr: 16:A9  Channel: 26  NID: 0x23  PHY: O-QPSK
2025-03-23 16:53:33,602 #          Long HWaddr: A6:E7:E2:FD:D8:F4:16:A9
2025-03-23 16:53:33,603 #           State: IDLE
2025-03-23 16:53:33,603 #          ACK_REQ  L2-PDU:102  MTU:1280  HL:64  RTR
2025-03-23 16:53:33,604 #          6LO  IPHC
2025-03-23 16:53:33,604 #          Source address length: 8
2025-03-23 16:53:33,604 #          Link type: wireless
2025-03-23 16:53:33,605 #          inet6 addr: fe80::a4e7:e2fd:d8f4:16a9  scope: link  VAL
2025-03-23 16:53:33,605 #          inet6 group: ff02::2
2025-03-23 16:53:33,606 #          inet6 group: ff02::1
2025-03-23 16:53:33,606 #          inet6 group: ff02::1:fff4:16a9
2025-03-23 16:53:33,606 #
2025-03-23 16:53:33,607 #          Statistics for Layer 2
2025-03-23 16:53:33,607 #            RX packets 0  bytes 0
2025-03-23 16:53:33,607 #            TX packets 0 (Multicast: 0)  bytes 0
2025-03-23 16:53:33,608 #            TX succeeded 0 errors 0
2025-03-23 16:53:33,608 #          Statistics for IPv6
2025-03-23 16:53:33,609 #            RX packets 0  bytes 0
2025-03-23 16:53:33,609 #            TX packets 0 (Multicast: 0)  bytes 0
2025-03-23 16:53:33,609 #            TX succeeded 0 errors 0
2025-03-23 16:53:33,610 #
```

Figura 4.24: Salida del comando `ifconfig` en la terminal del Border Router

4. Bajo estas condiciones, todavía no tenemos conectividad entre el Dongle y el PC, pues hay que asignar tanto a la interfaz virtual del PC como a la interfaz cableada del router una dirección IPv6 global.

   Para ello, desde la terminal del PC verificamos que la interfaz virtual creada en el paso 2 esté en UP, es decir, activa y preparada para transmitir paquetes, aunque aún no tenga dirección IP:

```
lab@lab:~$ ip link show enxa6e91023eea6
5: enxa6e91023eea6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT group default qlen 1000
    link/ether a6:e9:10:23:ee:a6 brd ff:ff:ff:ff:ff:ff
```

Figura 4.25: Comprobación del estado de la interfaz *enxa6e91023eea6* en la terminal del PC

   Ahora, asignamos como usuario root una dirección IPv6 global a esta interfaz, dentro del primer prefijo de red a utilizar (2001:db8:1::/64), mediante la orden:

   `sudo ip address add 2001:db8:1::1/64 dev enxa6e91023eea6`

   Tras esto, al volver a ejecutar `ifconfig` en la terminal del PC, la etiqueta `inet6` mostrará información de la dirección 2001:db8:1::1 recién configurada.

```
enxa6e91023eea6: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 2001:db8:1::1  prefixlen 64  scopeid 0x0<global>
        ether a6:e9:10:23:ee:a6  txqueuelen 1000  (Ethernet)
        RX packets 3  bytes 160 (160.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 97  bytes 16925 (16.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figura 4.26: Dirección IPv6 global asignada a la interfaz *enxa6e91023eea6* del PC

5. El último paso para tener conectividad a través de Internet entre el Border Router y el PC es añadir una dirección global en la interfaz cableada (`iface 6`) del primer dispositivo, con el mismo prefijo que la dirección establecida anteriormente. Así, desde la terminal de RIOT del Border Router introducimos:

   `ifconfig 6 add 2001:db8:1::2/64`

   De este modo, cuando ejecutemos a continuación `ifconfig 6`, aparecerá la nueva dirección asignada (2001:db8:1::2).

```
ifconfig 6
2025-03-23 17:00:40,334 # ifconfig 6
2025-03-23 17:00:40,335 # Iface  6  HWaddr: A6:E9:10:23:F2:A6
2025-03-23 17:00:40,337 #          L2-PDU:1500  MTU:1500  HL:64  RTR
2025-03-23 17:00:40,338 #          Source address length: 6
2025-03-23 17:00:40,338 #          Link type: wired
2025-03-23 17:00:40,339 #          inet6 addr: fe80::a4e9:10ff:fe23:f2a6  scope: link  VAL
2025-03-23 17:00:40,339 #          inet6 addr: 2001:db8:1::2  scope: global  VAL
2025-03-23 17:00:40,339 #          inet6 group: ff02::2
2025-03-23 17:00:40,340 #          inet6 group: ff02::1
2025-03-23 17:00:40,340 #          inet6 group: ff02::1:ff23:f2a6
2025-03-23 17:00:40,340 #          inet6 group: ff02::1:ff00:2
2025-03-23 17:00:40,340 #
2025-03-23 17:00:40,341 #          Statistics for Layer 2
2025-03-23 17:00:40,341 #            RX packets 99  bytes 17203
2025-03-23 17:00:40,341 #            TX packets 4 (Multicast: 4)  bytes 0
2025-03-23 17:00:40,342 #            TX succeeded 0 errors 0
2025-03-23 17:00:40,342 #          Statistics for IPv6
2025-03-23 17:00:40,342 #            RX packets 79  bytes 9637
2025-03-23 17:00:40,342 #            TX packets 4 (Multicast: 4)  bytes 224
2025-03-23 17:00:40,343 #            TX succeeded 4 errors 0
```

Figura 4.27: Dirección IPv6 global asignada a la interfaz cableada del Border Router

6. Una vez completado lo anterior, incorporamos el nRF52840 DK al montaje del escenario. Lo conectamos primero con el cable micro USB a otro puerto USB del PC. En consecuencia, ahora tenemos disponibles dos puertos serie más ligados a esta mota, “/dev/ttyACM1” y “/dev/ttyACM2”.

Análogamente a como se hizo en el escenario entre dos motas IoT, desde una nueva terminal del PC, que acabará siendo la terminal asociada a la mota que actúa de servidor CoAP, nos dirigimos al directorio de la aplicación "gcoap" de RIOT y ejecutamos:

```
make BOARD=nrf52840dk flash
make BOARD=nrf52840dk PORT=/dev/ttyACM1 term
```

Así, flasheamos en el DK esta aplicación y accedemos al shell que incorpora, haciendo uso de uno de los puertos disponibles para esta placa. Evidentemente, los comandos ofrecidos por el shell de "gcoap" son los mismos que los de la figura 4.19.

7. Para que el Border Router y la mota se comuniquen, el uso de direcciones IPv6 globales ya sabemos que no es necesario. No obstante, como el origen o destino de los mensajes es el PC, las interfaces inalámbricas de ambos equipos deben tener una dirección IPv6 global configurada. Solo de esta manera se podrá establecer la transmisión de mensajes directa entre el PC y la mota representada por el nRF52840 DK.

Por consiguiente, en la terminal del Border Router, asignamos una dirección IPv6 global a su interfaz inalámbrica (`iface 7`), con un prefijo de red distinto al que usamos en la comunicación por cable:

```
ifconfig 7 add 2001:db8:2:0:7b67:1860:420c:f43e/64
```

Después de introducir este comando, si se ejecuta `ifconfig 7`, la salida mostrará la nueva dirección global de esta interfaz (2001:db8:2:0:7b67:1860:420c:f43e):

```
ifconfig 7
2025-03-23 17:08:40,226 # ifconfig 7
2025-03-23 17:08:40,228 # Iface  7  HWaddr: 16:A9  Channel: 26  NID: 0x23  PHY: O-QPSK
2025-03-23 17:08:40,230 #           Long HWaddr: A6:E7:E2:FD:D8:F4:16:A9
2025-03-23 17:08:40,230 #            State: IDLE
2025-03-23 17:08:40,231 #           ACK_REQ  L2-PDU:102  MTU:1280  HL:64  RTR
2025-03-23 17:08:40,231 #           6LO  IPHC
2025-03-23 17:08:40,232 #           Source address length: 8
2025-03-23 17:08:40,232 #           Link type: wireless
2025-03-23 17:08:40,232 #           inet6 addr: fe80::a4e7:e2fd:d8f4:16a9  scope: link  VAL
2025-03-23 17:08:40,233 #           inet6 addr: 2001:db8:2:0:7b67:1860:420c:f43e  scope: global  VAL
2025-03-23 17:08:40,233 #           inet6 group: ff02::2
2025-03-23 17:08:40,233 #           inet6 group: ff02::1
2025-03-23 17:08:40,234 #           inet6 group: ff02::1:fff4:16a9
2025-03-23 17:08:40,234 #           inet6 group: ff02::1:ff0c:f43e
2025-03-23 17:08:40,234 #
2025-03-23 17:08:40,234 #           Statistics for Layer 2
2025-03-23 17:08:40,235 #             RX packets 15  bytes 617
2025-03-23 17:08:40,235 #             TX packets 0 (Multicast: 0)  bytes 0
2025-03-23 17:08:40,235 #             TX succeeded 0 errors 0
2025-03-23 17:08:40,236 #           Statistics for IPv6
2025-03-23 17:08:40,236 #             RX packets 15  bytes 932
2025-03-23 17:08:40,237 #             TX packets 0 (Multicast: 0)  bytes 0
2025-03-23 17:08:40,237 #             TX succeeded 0 errors 0
```

Figura 4.28: Dirección IPv6 global asignada a la interfaz inalámbrica del Border Router

Llegados a este punto, surge la necesidad de inicializar RPL en la interfaz inalámbrica entre el Border Router y la mota. Así, los mensajes intercambiados entre estos dispositivos podrán transmitirse al exterior (en nuestro caso, al sistema anfitrión).

Por tanto, desde la terminal del Border Router, iniciamos RPL en su interfaz inalámbrica gracias al comando:

```
rpl init 7
```

Además, recordemos que hemos de especificar que el Border Router será el nodo raíz del árbol DODAG de RPL. Esto lo conseguimos con la orden:

```
rpl root 0 2001:db8:2:0:7b67:1860:420c:f43e
```

Donde el valor `0` representa el identificador de la instancia RPL creada.

8. Tras montar la red RPL, obtenemos el resultado esperado, este es, la asignación automática de una dirección IPv6 global en la interfaz inalámbrica de la mota gracias a la transmisión de los DIO por parte del Border Router. Así, en la terminal de la mota podemos consultar con `ifconfig` que dicha dirección configurada toma el valor 2001:db8:2:0:d8c0:942f:47:4827, tal y como se muestra en la siguiente imagen:

```
ifconfig
2025-03-23 17:10:48,152 # ifconfig
2025-03-23 17:10:48,157 # Iface  6  HWaddr: 48:27  Channel: 26  NID: 0x23  PHY: O-QPSK
2025-03-23 17:10:48,162 #          Long HWaddr: DA:C0:94:2F:00:47:48:27
2025-03-23 17:10:48,164 #           State: IDLE
2025-03-23 17:10:48,169 #          ACK_REQ  L2-PDU:102  MTU:1280  HL:64  6LO
2025-03-23 17:10:48,170 #          IPHC
2025-03-23 17:10:48,173 #          Source address length: 8
2025-03-23 17:10:48,176 #          Link type: wireless
2025-03-23 17:10:48,182 #          inet6 addr: fe80::d8c0:942f:47:4827  scope: link  VAL
2025-03-23 17:10:48,188 #          inet6 addr: 2001:db8:2:0:d8c0:942f:47:4827  scope: global  VAL
2025-03-23 17:10:48,191 #          inet6 group: ff02::1
2025-03-23 17:10:48,194 #          inet6 group: ff02::1a
```

Figura 4.29: Dirección IPv6 global asignada a la interfaz inalámbrica de la mota

9. Aunque resulte natural pensar que en este momento el Border Router ya sabe cómo enrutar el tráfico entre la mota y el PC, no es así. De hecho, si en la terminal del router consultamos su tabla de rutas con el comando `nib route`, obtenemos:

```
nib route
2025-03-23 17:14:10,226 # nib route
2025-03-23 17:14:10,228 # 2001:db8:1::/64 dev #6
2025-03-23 17:14:10,228 # 2001:db8:2::/64 dev #7
2025-03-23 17:14:10,231 # 2001:db8:2:0:d8c0:942f:47:4827/128 via fe80::d8c0:942f:47:4827 dev #7
```

Figura 4.30: Salida del comando `nib route` en la terminal del Border Router

Por un lado, el prefijo 2001:db8:1::/64 está directamente conectado a través de la interfaz cableada. Sin embargo, la dirección 2001:db8:1::1 no aparece como directamente conectada, lo que se debe a que no corresponde a la dirección local de ninguna interfaz del Border Router. Mientras, en la segunda entrada de la salida observamos que los mensajes destinados al segundo prefijo de red se encaminan por la interfaz inalámbrica del Border Router. Este comportamiento es el esperado, y debido a la red montada con RPL, hay otra ruta disponible que indica por qué salto es alcanzable la mota. Aun así, con toda esta información, el Border Router no es capaz de enrutar tráfico para el cual no sea el destinatario final. Si ejecutamos:

```
nib route add 6 ::/0 2001:db8:1::1
```

Añadimos una ruta por defecto (`::/0`) con la interfaz virtual como gateway, cuya dirección era la 2001:db8:1::1. De esta manera, todo el tráfico que reciba el Border Router y que no tenga una ruta específica se enviará a esta dirección por la interfaz cableada, con lo que alcanzamos nuestro destino, el PC.

```
nib route add 6 ::/0 2001:db8:1::1
2025-03-23 17:14:52,750 # nib route add 6 ::/0 2001:db8:1::1
> nib route
2025-03-23 17:14:55,677 # nib route
2025-03-23 17:14:55,678 # 2001:db8:1::/64 dev #6
2025-03-23 17:14:55,679 # 2001:db8:2::/64 dev #7
2025-03-23 17:14:55,681 # 2001:db8:2:0:d8c0:942f:47:4827/128 via fe80::d8c0:942f:47:4827 dev #7
2025-03-23 17:14:55,682 # default* via 2001:db8:1::1 dev #6
```

Figura 4.31: Salida del comando `nib route` en la terminal del Border Router tras añadir la ruta por defecto

10. Asimismo, el PC tiene que conocer que el prefijo 2001:db8:2::/64 es alcanzable a través de la interfaz cableada del Border Router. Por tanto, desde el PC hemos de configurar la siguiente ruta estática:

```
sudo ip route add 2001:db8:2::/64 via 2001:db8:1::2
```

Gracias a los dos últimos comandos ejecutados, el Border Router sabe cómo encaminar los mensajes al PC, mientras que el PC sabe que el segundo prefijo de red es alcanzable por el Border Router.

De este modo, completamos la implementación del escenario propuesto, a la espera de verificar que la comunicación IPv6 y CoAP funciona con éxito. Como ocurría en la situación entre dos motas IoT, desde el lado del servidor CoAP podemos comprobar que este equipo está en escucha mediante la orden `coap info`:

```
> coap info
2025-03-23 17:18:54,275 # coap info
2025-03-23 17:18:54,278 # CoAP server is listening on port 5683
2025-03-23 17:18:54,281 #  CLI requests sent: 0
2025-03-23 17:18:54,282 # CoAP open requests: 0
2025-03-23 17:18:54,284 # Configured Proxy: None
```

Figura 4.32: Salida del comando `coap info` en la terminal de la mota servidora

4.6.3. Montaje del escenario entre dos motas IoT con libCoAP

Finalmente, detallamos los pasos seguidos para replicar el escenario entre dos motas IoT integrando libCoAP en RIOT, en lugar de la implementación nativa "gcoap" soportada hoy en día por este sistema operativo. A pesar de que el proceso sea equivalente al caso de "gcoap", sí conviene señalar ciertas diferencias en la fase de despliegue.

Como este caso particular del escenario entre dos motas IoT es el que aprovecharemos para analizar el tráfico generado en él desde *Wireshark*, en primer lugar describimos cómo instalar el programa asociado a la herramienta *nRF Sniffer* en el nRF52840 Dongle. Recordemos que este dispositivo en el segundo escenario se utiliza para cargar el firmware de la aplicación "gnrc_border_router", y ahora reemplazamos esa funcionalidad para programarlo como Sniffer.

1. Instalamos *Wireshark* y añadimos el usuario con el que estemos operando al grupo.

   ```
   sudo apt-get install wireshark
   sudo usermod -aG wireshark $USER
   ```

2. Instalamos también el módulo *PySerial*.

   ```
   sudo apt install python3-serial
   ```

3. Descargamos el plugin del *nRF Sniffer* desde el repositorio correspondiente:

   ```
   git clone https://github.com/NordicSemiconductor/nRF-Sniffer-for-802.15.4
   ```

4. Para añadir el plugin a *Wireshark*, abrimos la aplicación y, en "Ayuda → Acerca de Wireshark", nos dirigimos a "Carpetas". Dentro de "Ruta global de Extcap", añadimos en la columna de ubicación el fichero "nrf802154_sniffer.py" del directorio `nRF-Sniffer-for-802.15.4/nrf802154_sniffer`.

Figura 4.33: Vista de la pestaña "Carpetas" en "Acerca de Wireshark"

Opcionalmente, podemos asegurarnos de que el Sniffer funciona correctamente mediante el siguiente comando, con el que se ordena al script de Python que enumere y describa las interfaces de captura disponibles que puede proporcionar a *Wireshark*:

```
lab@iab:/usr/lib/x86_64-linux-gnu/wireshark/extcap$ ./nrf802154_sniffer.py --extcap-interfaces
extcap {version=0.8.0}{help=https://github.com/NordicSemiconductor/nRF-Sniffer-for-802.15.4}{display=nRF Sniffer for 802.15.4}
control {number=6}{type=button}{role=logger}{display=Log}{tooltip=Show capture log}
```

Figura 4.34: Salida del comando `./nrf802154_sniffer.py --extcap-interfaces`

5. En este momento es precisamente cuando necesitamos descargar la aplicación *nRF Connect for Desktop*. Esta nos servirá para cargar el firmware del Sniffer en el Dongle. Asumiendo que tenemos la versión más reciente de dicha aplicación y hemos permitido que se ejecute como un programa, procedemos como sigue:

   a) Dentro de *nRF Connect for Desktop*, instalamos la herramienta "Programmer" y la abrimos. Conectamos el Dongle por USB al equipo anfitrión y lo ponemos en modo DFU, es decir, en un estado listo para recibir una nueva actualización de firmware.

   b) Tras clicar en "Select Device", la herramienta reconoce el Dongle y resulta la siguiente panorámica:

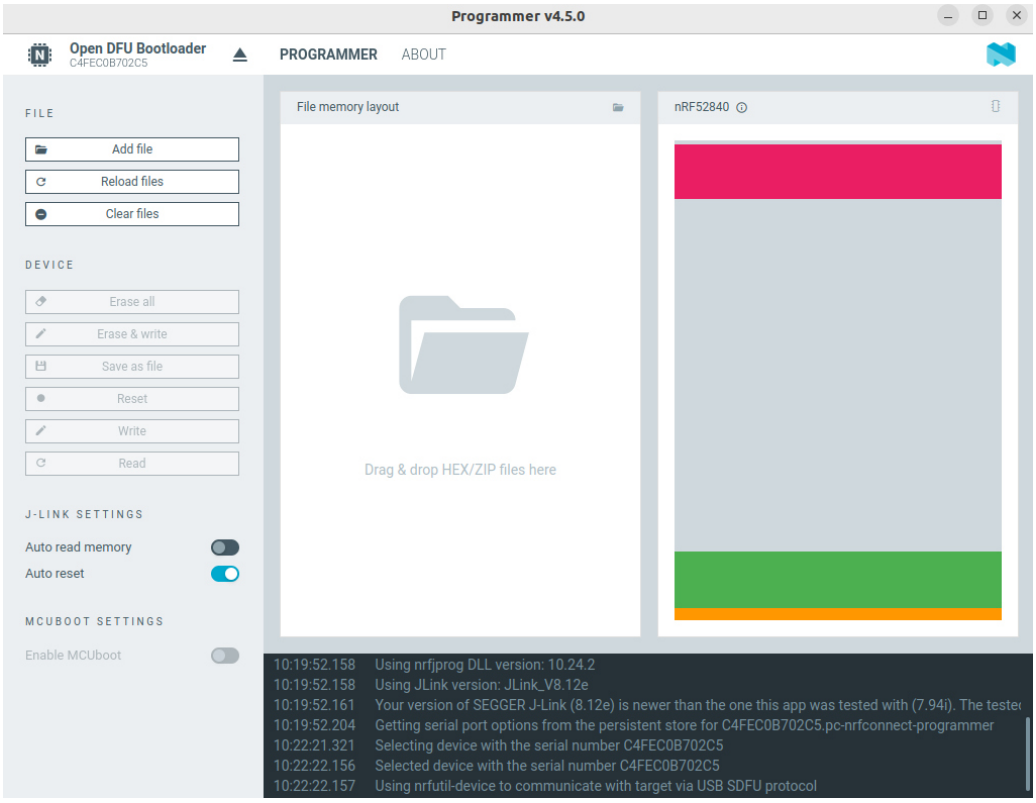


Figura 4.35: Panorámica de "Programmer" en *nRF Connect for Desktop* tras seleccionar el Dongle

   c) En "Add file", seleccionamos el fichero "nrf802154_sniffer_nrf52840dongle.hex", que contiene el firmware del Sniffer y se encuentra en el repositorio que descargamos con el plugin. Entonces, al darle a "Write", el firmware se escribe en el Dongle, cuyo LED programable cambia de parpadear en rojo a verde. Esto indica que se ha conseguido grabar con éxito el programa del Sniffer en el Dongle.

6. Una vez cargado el firmware en el Dongle y refrescado *Wireshark*, observamos cómo aparece una nueva interfaz asociada al Sniffer:

Figura 4.36: Interfaz del *nRF Sniffer* en *Wireshark*

Finalmente, es importante configurar el canal en el que va a escuchar el Dongle que actúa como Sniffer, aparte de comprobar el puerto en el que está (en adelante, supondremos que es el de la siguiente imagen, el "/dev/ttyACM0").

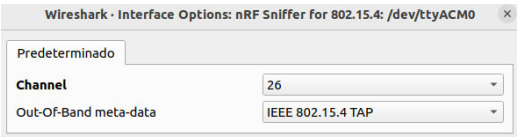


Figura 4.37: Configuración de la interfaz del *nRF Sniffer* en *Wireshark*

Ahora ya tenemos operativo el Dongle como Sniffer capturando los paquetes que se transmiten por IEEE 802.15.4. Para completar el montaje del escenario, el procedimiento consta de los siguientes pasos:

7. Como mencionamos en la sección 4.5.3, necesitamos hacer uso de una versión particular de RIOT [46]. Para ello, incorporamos la rama "libcoap_add":

```
git clone https://github.com/mrdeep1/RIOT
cd RIOT
git checkout libcoap_add
```

Así, recuperamos toda la funcionalidad necesaria para integrar el paquete libCoAP en RIOT, y las aplicaciones de ejemplo "libcoap_server" y "libcoap_client".

8. Conectamos uno de los nRF52840 DK vía USB y, desde una terminal del PC, una vez dentro del directorio `/examples/networking/libcoap/libcoap_server`, cargamos el firmware de la aplicación "libcoap_server" con:

```
make BOARD=nrf52840dk flash
```

A continuación, desconectamos el DK anterior, conectamos otro diferente y, desde otra terminal del PC que vamos a asociar al cliente CoAP (la primera por tanto quedará ligada al servidor), en el directorio de la aplicación "libcoap_client", ejecutamos el mismo comando. Así, ya tenemos los programas flasheados en los dos dispositivos empleados.

9. En este momento, podemos conectar ambas placas y acceder a la terminal de las aplicaciones con:

```
make BOARD=nrf52840dk PORT=/dev/ttyACM1 term
make BOARD=nrf52840dk PORT=/dev/ttyACM3 term
```

Puesto que tenemos el Dongle conectado como Sniffer en el puerto "/dev/ttyACM0" (véase la figura 4.37), hemos de especificar para las dos motas sus puertos correspondientes con el fin de evitar conflictos, al igual que en los escenarios anteriores.

10. Mediante la orden `help`, podemos observar los comandos disponibles en ambas aplicaciones, donde la única diferencia reside en que "libcoap_client" dispone del comando `coapc` para la construcción de solicitudes como cliente, mientras que "libcoap_server" ofrece en su lugar la orden `coaps` para iniciar o detener el servidor.

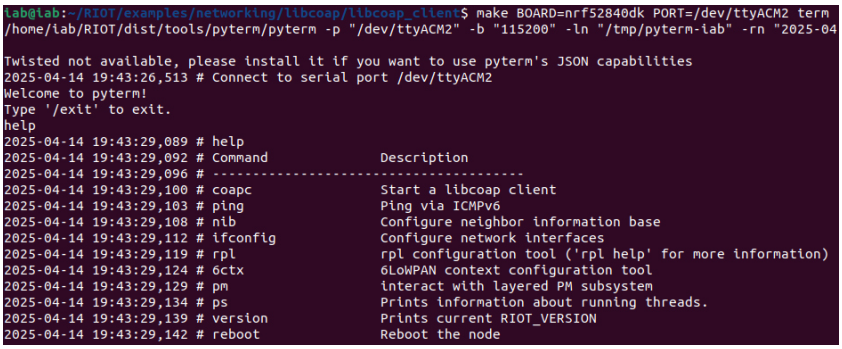


Figura 4.38: Comandos disponibles en la aplicación "libcoap_client"

Igualmente, con `ifconfig` descubrimos las direcciones IPv6 link-local de ambas motas. En este caso, el servidor y el cliente poseen las direcciones fe80::64b3:7881:5633:4d32 y fe80::d8c0:942f:47:4827, respectivamente. Por ejemplo, para el servidor vemos:

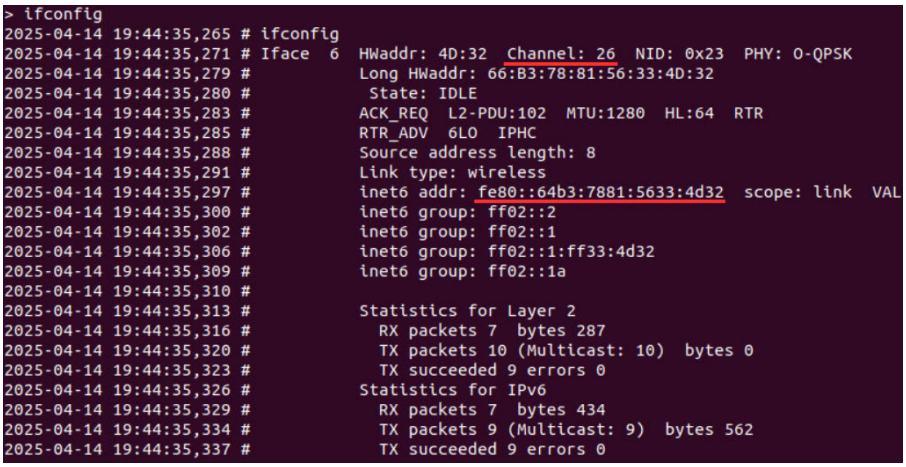


Figura 4.39: Salida del comando `ifconfig` en la terminal de la mota servidora

Una información de interés que nos devuelve `ifconfig` es "Channel: 26". Este es el canal donde el Dongle, que está configurado como Sniffer, se tiene que poner a escuchar el tráfico desde *Wireshark*. Notemos que, en la figura 4.37, aparece precisamente este valor para que la captura de los mensajes en las pruebas se lleve a cabo en el canal correcto.

11. El último paso previo a llevar a cabo las pruebas consiste en iniciar el servidor CoAP, lo que conseguimos a través del comando `coaps start`:

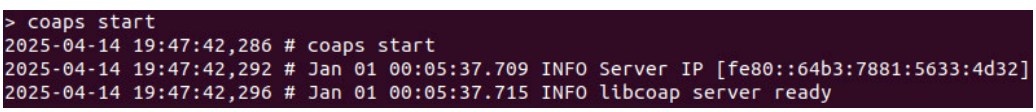


Figura 4.40: Salida del comando `coaps start` en la terminal de la mota servidora

Con esta serie de pasos, concluimos el despliegue de los tres escenarios en el sistema operativo RIOT. Es evidente que, tras el estudio teórico de la comunicación a nivel de red y aplicación, el principal desafío encontrado para completar estos montajes es el de identificar las aplicaciones de RIOT a utilizar, pues una vez descubiertas, los comandos que proporcionan nos son conocidos y es fácil interactuar con ellas de cara a configurar los diferentes enlaces entre componentes.

5. Pruebas de concepto y análisis de resultados

En este capítulo, describiremos las pruebas realizadas con el fin de validar los objetivos planteados en los dos escenarios inicialmente considerados, que a la hora de trasladarlos a RIOT han pasado a ser tres por la distinción entre las implementaciones de CoAP. Para ello, partiremos de donde hemos concluido el montaje de cada escenario y demostraremos que las aportaciones de este trabajo conducen al establecimiento de comunicaciones IPv6 y CoAP entre dispositivos IoT a través de una red inalámbrica y también hacia una red IPv6 pública.

Sin pérdida de generalidad, de cara a probar el funcionamiento de GCoAP, consideraremos que el recurso que el cliente va a solicitar es "/riot/board", uno de los que el servidor tiene disponibles en la aplicación "gcoap", y que nos devuelve el nombre de la placa asociada a este equipo (que deberá ser *nrf52840dk*). En el caso de la aplicación CoAP implementada con libCoAP, se preguntará por el listado de los recursos que hay disponibles en el servidor por medio del objeto "./well-known/core".

5.1. Escenario entre dos motas IoT con GCoAP

Como las motas están en la misma PAN, en cuanto consultamos las direcciones IPv6 link-local en el paso 6 de la sección 4.6.1, estamos en disposición de realizar las pruebas de conectividad a través de mensajes `ping` de una mota a otra. Recordemos que los paquetes se transmiten en la capa de enlace vía radio de acuerdo al estándar IEEE 802.15.4. Por tanto, desde las terminales del cliente y del servidor respectivamente, procedemos al envío de los `ping` hacia el otro extremo de la comunicación:

```
> ping fe80::d8c0:942f:47:4827
2025-03-24 12:10:17,032 # ping fe80::d8c0:942f:47:4827
2025-03-24 12:10:17,048 # 12 bytes from fe80::d8c0:942f:47:4827%6: icmp_seq=0 ttl=64 rssi=-168 dBm time=8.552 ms
2025-03-24 12:10:18,047 # 12 bytes from fe80::d8c0:942f:47:4827%6: icmp_seq=1 ttl=64 rssi=72 dBm time=7.913 ms
2025-03-24 12:10:19,045 # 12 bytes from fe80::d8c0:942f:47:4827%6: icmp_seq=2 ttl=64 rssi=72 dBm time=5.669 ms
2025-03-24 12:10:19,046 #
2025-03-24 12:10:19,049 # --- fe80::d8c0:942f:47:4827 PING statistics ---
2025-03-24 12:10:19,054 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-24 12:10:19,058 # round-trip min/avg/max = 5.669/7.378/8.552 ms
```

Figura 5.1: Salida del `ping` desde la mota cliente a la mota servidora

```
> ping fe80::64b3:7881:5633:4d32
2025-03-24 12:11:54,410 # ping fe80::64b3:7881:5633:4d32
2025-03-24 12:11:54,426 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=0 ttl=64 rssi=56 dBm time=8.232 ms
2025-03-24 12:11:55,424 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=1 ttl=64 rssi=56 dBm time=7.273 ms
2025-03-24 12:11:56,426 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=2 ttl=64 rssi=56 dBm time=8.871 ms
2025-03-24 12:11:56,426 #
2025-03-24 12:11:56,430 # --- fe80::64b3:7881:5633:4d32 PING statistics ---
2025-03-24 12:11:56,436 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-24 12:11:56,440 # round-trip min/avg/max = 7.273/8.125/8.871 ms
```

Figura 5.2: Salida del `ping` desde la mota servidora a la mota cliente

Ambos dispositivos responden al mensaje `ping` equivalente a un *ICMPv6 Echo Request*, por lo que de esta manera comprobamos que las motas pueden comunicarse entre sí a través del protocolo IPv6, lo que está en consonancia con el diagrama de intercambio de mensajes descrito en la figura 4.6.

Finalmente, verificamos el funcionamiento de GCoAP. Partimos de que, en la terminal del servidor, la orden `coap info` nos indica que este está escuchando en el puerto UDP 5683. Entonces, construimos como cliente la siguiente solicitud GET por "/riot/board", de modo que el paquete CoAP tenga como dirección de red destino la dirección IPv6 link-local de la mota servidora:

```
> coap get -c [fe80::d8c0:942f:47:4827]:5683 /riot/board
2025-03-24 12:13:40,618 # coap get -c [fe80::d8c0:942f:47:4827]:5683 /riot/board
2025-03-24 12:13:40,622 # gcoap_cli: sending msg ID 51174, 17 bytes
> 2025-03-24 12:13:40,637 # gcoap: response Success, code 2.05, 10 bytes
2025-03-24 12:13:40,638 # nrf52840dk
coap info
2025-03-24 12:13:53,205 # coap info
2025-03-24 12:13:53,208 # CoAP server is listening on port 5683
2025-03-24 12:13:53,210 #  CLI requests sent: 1
2025-03-24 12:13:53,212 # CoAP open requests: 0
2025-03-24 12:13:53,214 # Configured Proxy: None
```

Figura 5.3: Solicitud CoAP desde la mota cliente por el recurso "/riot/board"

Observamos desde el lado del cliente que se devuelve la respuesta esperada satisfactoriamente. Esta es *nrf52840dk*, el nombre de la placa que utiliza el servidor. Además, mediante el comando `coap info`, contemplamos en la figura 5.3 que la petición se ha enviado con éxito y queda registrada en la aplicación. Otra forma de verificar este aspecto es gracias a la línea de código de depuración añadida en el fichero *server.c*, pues, si consultamos la terminal del servidor, visualizamos el mensaje añadido:

```
> ping fe80::64b3:7881:5633:4d32
2025-03-24 12:11:54,410 # ping fe80::64b3:7881:5633:4d32
2025-03-24 12:11:54,426 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=0 ttl=64 rssi=56 dBm time=8.232 ms
2025-03-24 12:11:55,424 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=1 ttl=64 rssi=56 dBm time=7.273 ms
2025-03-24 12:11:56,426 # 12 bytes from fe80::64b3:7881:5633:4d32%6: icmp_seq=2 ttl=64 rssi=56 dBm time=8.871 ms
2025-03-24 12:11:56,426 #
2025-03-24 12:11:56,430 # --- fe80::64b3:7881:5633:4d32 PING statistics ---
2025-03-24 12:11:56,436 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-24 12:11:56,440 # round-trip min/avg/max = 7.273/8.125/8.871 ms
2025-03-24 12:13:40,628 # Solicitud recibida en /riot/board
```

Figura 5.4: Solicitud CoAP recibida con éxito por la mota servidora

Así, a partir del escenario montado, estas pruebas constatan que la comunicación mediante los protocolos IPv6 y CoAP entre motas IoT que comparten una red 6LoWPAN en RIOT funciona según los planteamientos teóricos comentados en la sección 4.3.

5.2. Escenario entre una mota IoT y el PC con GCoAP

Dado que el escenario entre una mota IoT como servidor CoAP y el PC como cliente se caracteriza porque los mensajes son enrutados por un punto intermedio, el Border Router, un paso previo que constata que la comunicación extremo a extremo va a funcionar según lo esperado es el de probar la conectividad IPv6 entre el PC y la mota de forma individual con el Border Router. Sin ella, está claro que no se alcanzaría el resultado deseado. De acuerdo a lo montado en la sección 4.6.2, una vez completado el paso 5 (es decir, tras crear la interfaz virtual del PC y asignarle tanto a esta como a la interfaz cableada del Border Router una dirección IPv6 global dentro del mismo segmento de red), ya podemos verificar directamente que tenemos conectividad de red entre el Dongle que ejerce de Border Router y el PC mediante el envío de mensajes `ping`. Mostramos las salidas obtenidas en la terminal del Border Router y la del PC, respectivamente:

```
> ping 2001:db8:1::1
2025-03-23 17:01:47,793 # ping 2001:db8:1::1
2025-03-23 17:01:47,797 # 12 bytes from 2001:db8:1::1: icmp_seq=0 ttl=64 time=2.015 ms
2025-03-23 17:01:48,796 # 12 bytes from 2001:db8:1::1: icmp_seq=1 ttl=64 time=1.095 ms
2025-03-23 17:01:49,797 # 12 bytes from 2001:db8:1::1: icmp_seq=2 ttl=64 time=1.066 ms
2025-03-23 17:01:49,797 #
2025-03-23 17:01:49,798 # --- 2001:db8:1::1 PING statistics ---
2025-03-23 17:01:49,800 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-23 17:01:49,801 # round-trip min/avg/max = 1.066/1.392/2.015 ms
```

Figura 5.5: Salida del `ping` desde el Border Router al PC

```
iab@iab:~$ ping 2001:db8:1::2 -c3
PING 2001:db8:1::2(2001:db8:1::2) 56 data bytes
64 bytes from 2001:db8:1::2: icmp_seq=1 ttl=64 time=2.34 ms
64 bytes from 2001:db8:1::2: icmp_seq=2 ttl=64 time=1.55 ms
64 bytes from 2001:db8:1::2: icmp_seq=3 ttl=64 time=1.51 ms

--- 2001:db8:1::2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 1.511/1.800/2.341/0.382 ms
```

Figura 5.6: Salida del `ping` desde el PC al Border Router

Además, la implementación llevada a cabo para asignar las direcciones IPv6 globales a las motas IoT (es decir, el nRF52840 DK y el nRF52840 Dongle) y configurar la red RPL entre ellas nos permite que, tras el paso 8 del montaje, el envío de mensajes `ping` desde la mota al Border Router y viceversa se realice satisfactoriamente:

```
ping 2001:db8:2:0:d8c0:942f:47:4827
2025-03-23 17:11:42,104 # ping 2001:db8:2:0:d8c0:942f:47:4827
2025-03-23 17:11:42,117 # 12 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=0 ttl=64 rssi=56 dBm time=9.369 ms
2025-03-23 17:11:43,115 # 12 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=1 ttl=64 rssi=8 dBm time=7.766 ms
2025-03-23 17:11:44,116 # 12 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=2 ttl=64 rssi=8 dBm time=8.086 ms
2025-03-23 17:11:44,116 #
2025-03-23 17:11:44,118 # --- 2001:db8:2:0:d8c0:942f:47:4827 PING statistics ---
2025-03-23 17:11:44,119 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-23 17:11:44,119 # round-trip min/avg/max = 7.766/8.407/9.369 ms
```

Figura 5.7: Salida del `ping` desde el Border Router a la mota

```
ping 2001:db8:2:0:7b67:1860:420c:f43e
2025-03-23 17:12:59,320 # ping 2001:db8:2:0:7b67:1860:420c:f43e
2025-03-23 17:12:59,338 # 12 bytes from 2001:db8:2:0:7b67:1860:420c:f43e: icmp_seq=0 ttl=64 rssi=-152 dBm time=10.010 ms
2025-03-23 17:13:00,339 # 12 bytes from 2001:db8:2:0:7b67:1860:420c:f43e: icmp_seq=1 ttl=64 rssi=-136 dBm time=10.331 ms
2025-03-23 17:13:01,340 # 12 bytes from 2001:db8:2:0:7b67:1860:420c:f43e: icmp_seq=2 ttl=64 rssi=-24 dBm time=10.009 ms
2025-03-23 17:13:01,340 #
2025-03-23 17:13:01,345 # --- 2001:db8:2:0:7b67:1860:420c:f43e PING statistics ---
2025-03-23 17:13:01,350 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-23 17:13:01,354 # round-trip min/avg/max = 10.009/10.116/10.331 ms
```

Figura 5.8: Salida del `ping` desde la mota al Border Router

De esta manera, tenemos conectividad a nivel de red entre los dos extremos de la comunicación con el Border Router, y tras establecer las rutas estáticas de los pasos 9 y 10, comprobamos que la mota IoT y el PC responden a la transmisión de mensajes `ping` del otro:

```
ping 2001:db8:1::1
2025-03-23 17:16:18,236 # ping 2001:db8:1::1
2025-03-23 17:16:18,255 # 12 bytes from 2001:db8:1::1: icmp_seq=0 ttl=63 rssi=-136 dBm time=11.384 ms
2025-03-23 17:16:19,252 # 12 bytes from 2001:db8:1::1: icmp_seq=1 ttl=63 rssi=-152 dBm time=9.055 ms
2025-03-23 17:16:20,253 # 12 bytes from 2001:db8:1::1: icmp_seq=2 ttl=63 rssi=-152 dBm time=10.087 ms
2025-03-23 17:16:20,254 #
2025-03-23 17:16:20,257 # --- 2001:db8:1::1 PING statistics ---
2025-03-23 17:16:20,262 # 3 packets transmitted, 3 packets received, 0% packet loss
2025-03-23 17:16:20,266 # round-trip min/avg/max = 9.055/10.175/11.384 ms
```

Figura 5.9: Salida del `ping` desde la mota al PC

```
lab@iab:~$ ping 2001:db8:2:0:d8c0:942f:47:4827 -c3
PING 2001:db8:2:0:d8c0:942f:47:4827(2001:db8:2:0:d8c0:942f:47:4827) 56 data bytes
64 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=1 ttl=63 time=17.5 ms
64 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=2 ttl=63 time=18.1 ms
64 bytes from 2001:db8:2:0:d8c0:942f:47:4827: icmp_seq=3 ttl=63 time=18.3 ms

--- 2001:db8:2:0:d8c0:942f:47:4827 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 17.501/17.972/18.271/0.337 ms
```

Figura 5.10: Salida del `ping` desde el PC a la mota

Así, hemos logrado implementar sobre RIOT el escenario de red explicado en la sección 4.4. Finalmente, probamos la comunicación a nivel de aplicación con CoAP. Recordemos que, en la terminal del servidor, nos aseguramos al final del montaje que el nRF52840 DK que actúa como tal está en escucha.

Entonces, desde la terminal del PC, este construye un mensaje CoAP correspondiente a una solicitud GET por el recurso "/riot/board", cuya dirección destino es la dirección IPv6 global asignada a la mota, que pertenece al prefijo de la red 6LoWPAN. En la salida del comando con el que se realiza tal solicitud, contemplamos que el servidor CoAP devuelve el nombre de la placa, por lo que la petición ha seguido el flujo asociado al diagrama de la figura 4.10 y ha llegado hasta el destinatario final.

```
iab@iab:~$ coap-client-gnutls coap://[2001:db8:2:0:d8c0:942f:47:4827]/riot/board
nrf52840dk
```

Figura 5.11: Solicitud CoAP desde el PC a la mota por el recurso "/riot/board"

Por otra parte, en la terminal del servidor, gracias a la modificación que se hizo en el fichero *server.c* de la aplicación "gcoap", también podemos ver que el servidor CoAP recibió la solicitud correctamente, de forma análoga a como ocurría en la figura 5.4. Esta prueba del envío de un mensaje CoAP al servidor ubicado en la mota completa los objetivos propuestos para el escenario diseñado en la sección 4.4.

Una vez completado lo anterior, recordemos que en el paso 2 del montaje de este escenario empezamos a capturar todo el tráfico generado en adelante desde la interfaz *enxa6e91023eea6*. Así, si consultamos la traza obtenida, filtrando por los mensajes ICMPv6 y CoAP, observamos en primer lugar los `ping` que enviamos del Border Router al PC:

```
70 133.927143216  2001:db8:1::2   ff02::1:ff00:1   ICMPv6   86 Neighbor Solicitation for 2001:db8:1::1 from a6:e9:10:23:f2:a6
71 133.927211556  2001:db8:1::1   2001:db8:1::2    ICMPv6   86 Neighbor Advertisement 2001:db8:1::1 (sol, ovr) is at a6:e9:10:23:ee:a6
72 133.928032242  2001:db8:1::2   2001:db8:1::1    ICMPv6   66 Echo (ping) request id=0x3dca, seq=0, hop limit=64 (reply in 73)
73 133.928061941  2001:db8:1::1   2001:db8:1::2    ICMPv6   66 Echo (ping) reply id=0x3dca, seq=0, hop limit=64 (request in 72)
74 134.929107715  2001:db8:1::2   2001:db8:1::1    ICMPv6   66 Echo (ping) request id=0x3dca, seq=1, hop limit=64 (reply in 75)
75 134.929145667  2001:db8:1::1   2001:db8:1::2    ICMPv6   66 Echo (ping) reply id=0x3dca, seq=1, hop limit=64 (request in 74)
```

Figura 5.12: Flujo de los mensajes `ping` enviados por el Border Router al PC

Notemos que los mensajes 70 y 71 forman parte del protocolo Neighbor Discovery de IPv6. El Border Router pregunta por multicast con un *Neighbor Solicitation* quién tiene la dirección IPv6 2001:db8:1::1, y la interfaz del PC con esa dirección responde en el mensaje *Neighbor Advertisement*, proporcionando su dirección MAC (subrayada en la figura 5.12). De esta manera, el Border Router conoce la dirección MAC de la interfaz virtual del PC y puede comunicarse con ella mediante mensajes *ICMPv6 Echo Request* a nivel de enlace vía Ethernet, como se visualiza en esta imagen:

```
   72 133.928032242  2001:db8:1::2   2001:db8:1::1   ICMPv6   66 Echo (ping) request

Frame 72: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface enxa6e91023eea6, id 0
Ethernet II, Src: a6:e9:10:23:f2:a6 (a6:e9:10:23:f2:a6), Dst: a6:e9:10:23:ee:a6 (a6:e9:10:23:ee:a6)
Internet Protocol Version 6, Src: 2001:db8:1::2, Dst: 2001:db8:1::1
Internet Control Message Protocol v6
```

Figura 5.13: Transmisión vía Ethernet de un mensaje *ICMPv6 Echo Request* del Border Router al PC

De la figura anterior, contemplamos también que el contenido del paquete ICMPv6 sigue el formato estudiado en el diseño del escenario. Después, la traza registra una secuencia de mensajes análoga cuando los `ping` los enviamos desde la interfaz virtual del PC a la interfaz cableada del Border Router:

```
87 210.637894497  2001:db8:1::2  2001:db8:1::1  ICMPv6  86 Neighbor Advertisement 2001:db8:1::2 (sol, ovr) is at a6:e9:10:23:f2:a6
88 210.637936303  2001:db8:1::1  2001:db8:1::2  ICMPv6  118 Echo (ping) request id=0x0002, seq=1, hop limit=64 (reply in 89)
89 210.638880202  2001:db8:1::2  2001:db8:1::1  ICMPv6  118 Echo (ping) reply id=0x0002, seq=1, hop limit=64 (request in 88)
90 211.638265923  2001:db8:1::1  2001:db8:1::2  ICMPv6  118 Echo (ping) request id=0x0002, seq=2, hop limit=64 (reply in 91)
91 211.639354830  2001:db8:1::2  2001:db8:1::1  ICMPv6  118 Echo (ping) reply id=0x0002, seq=2, hop limit=64 (request in 90)
92 212.639829385  2001:db8:1::1  2001:db8:1::2  ICMPv6  118 Echo (ping) request id=0x0002, seq=3, hop limit=64 (reply in 93)
93 212.640820115  2001:db8:1::2  2001:db8:1::1  ICMPv6  118 Echo (ping) reply id=0x0002, seq=3, hop limit=64 (request in 92)
94 215.635849608  2001:db8:1::2  2001:db8:1::1  ICMPv6  86 Neighbor Solicitation for 2001:db8:1::1 from a6:e9:10:23:f2:a6
95 215.635895059  2001:db8:1::1  2001:db8:1::2  ICMPv6  78 Neighbor Advertisement 2001:db8:1::1 (sol)
```

Figura 5.14: Flujo de los mensajes `ping` enviados por el PC al Border Router

El único cambio significativo lo vemos en los mensajes 94 y 95, donde ahora el *Neighbor Solicitation* no se envía por multicast, sino por unicast a la dirección de la interfaz virtual del PC. Esto vuelve a ocurrir posteriormente en la traza de forma periódica y se debe a que el Border Router está comprobando que el PC no tiene una dirección duplicada, o bien que sigue activo. A continuación, visualizamos los mensajes enviados por la mota IoT al PC correspondientes a la figura 5.9.

```
109 693.689133030  2001:db8:2:0:d8c0:942f:47:4827  2001:db8:1::1          ICMPv6  66 Echo (ping) request id=0xe0ba, seq=0, hop limit=63 (reply in 110)
110 693.689178350  2001:db8:1::1                   2001:db8:2:0:d8c0:94…  ICMPv6  66 Echo (ping) reply id=0xe0ba, seq=0, hop limit=64 (request in 109)
111 694.688511726  2001:db8:2:0:d8c0:942f:47:4827  2001:db8:1::1          ICMPv6  66 Echo (ping) request id=0xe0ba, seq=1, hop limit=63 (reply in 112)
112 694.688553260  2001:db8:1::1                   2001:db8:2:0:d8c0:94…  ICMPv6  66 Echo (ping) reply id=0xe0ba, seq=1, hop limit=64 (request in 111)
113 695.687559126  2001:db8:2:0:d8c0:942f:47:4827  2001:db8:1::1          ICMPv6  66 Echo (ping) request id=0xe0ba, seq=2, hop limit=63 (reply in 114)
114 695.687599088  2001:db8:1::1                   2001:db8:2:0:d8c0:94…  ICMPv6  66 Echo (ping) reply id=0xe0ba, seq=2, hop limit=64 (request in 113)
115 698.687326572  2001:db8:1::2                   2001:db8:1::1          ICMPv6  86 Neighbor Solicitation for 2001:db8:1::1 from a6:e9:10:23:f2:a6
116 698.687374925  2001:db8:1::1                   2001:db8:1::2          ICMPv6  78 Neighbor Advertisement 2001:db8:1::1 (sol)
```

Figura 5.15: Flujo de los mensajes `ping` enviados por la mota IoT al PC

Si todo funciona de acuerdo a lo explicado, estos mensajes *ICMPv6 Echo Request* deben tener como direcciones IPv6 origen y destino las de la interfaz inalámbrica de la mota y la cableada del PC respectivamente, pero a nivel de enlace la dirección MAC origen debe ser la del Border Router y la destino la del PC (subrayadas por separado en las figuras 5.12 y 5.14). En efecto, esto es lo que sucede:

```
109 693.689133030  2001:db8:2:0:d8c0:942f:47:4827  2001:db8:1::1  ICMPv6  66 Echo (ping) request

Frame 109: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface enxa6e91023eea6, id 0
Ethernet II, Src: a6:e9:10:23:f2:a6 (a6:e9:10:23:f2:a6), Dst: a6:e9:10:23:ee:a6 (a6:e9:10:23:ee:a6)
  > Destination: a6:e9:10:23:ee:a6 (a6:e9:10:23:ee:a6)
  > Source: a6:e9:10:23:f2:a6 (a6:e9:10:23:f2:a6)
    Type: IPv6 (0x86dd)
    [Stream index: 7]
Internet Protocol Version 6, Src: 2001:db8:2:0:d8c0:942f:47:4827, Dst: 2001:db8:1::1
Internet Control Message Protocol v6
```

Figura 5.16: Cabeceras de un mensaje *ICMPv6 Echo Request* de la mota IoT al PC

Tras los `ping` que van del PC a la mota, la traza concluye con los mensajes CoAP generados, que siguen el mismo patrón en las direcciones a nivel de red y de enlace. En la respuesta del servidor CoAP, encontramos dentro del payload el valor del recurso solicitado (*nrf52840dk*).

```
149 855.939290521  2001:db8:1::1                   2001:db8:2:0:d8c0:94…  CoAP  77 CON, MID:26984, GET, /riot/board
150 855.950140668  2001:db8:2:0:d8c0:942f:47:4827  2001:db8:1::1          CoAP  78 ACK, MID:26984, 2.05 Content (text/plain)

Frame 150: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface enxa6e91023eea6, id 0
Ethernet II, Src: a6:e9:10:23:f2:a6 (a6:e9:10:23:f2:a6), Dst: a6:e9:10:23:ee:a6 (a6:e9:10:23:ee:a6)
Internet Protocol Version 6, Src: 2001:db8:2:0:d8c0:942f:47:4827, Dst: 2001:db8:1::1
User Datagram Protocol, Src Port: 5683, Dst Port: 37541
Constrained Application Protocol, Acknowledgement, 2.05 Content, MID:26984
    01.. .... = Version: 1
    ..10 .... = Type: Acknowledgement (2)
    .... 0000 = Token Length: 0
    Code: 2.05 Content (69)
    Message ID: 26984
  > Opt Name: #1: Content-Format: text/plain; charset=utf-8
    End of options marker: 255
  > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 10
Line-based text data: text/plain (1 lines)
    nrf52840dk
```

Figura 5.17: Flujo de mensajes CoAP entre la mota IoT y el PC

Así, concluimos que la estructura de los paquetes intercambiados en este escenario implementado en RIOT entre una mota IoT y el PC funciona tal y como cabía esperar. Entraremos más en detalle acerca del contenido de los mensajes ICMPv6 y CoAP en el último escenario, pues para este caso el estudio asociado sería análogo, con la salvedad de las diferencias que se han recalcado en lo que concierne a la transmisión en la capa de enlace.

5.3. Escenario entre dos motas IoT con libCoAP

Por último, replicamos el escenario entre dos motas IoT en el que utilizamos las aplicaciones de libCoAP en RIOT indicadas al final de la sección 4.5.2.

Nuevamente, la comunicación es inalámbrica sobre una red 6LoWPAN compartida y los paquetes se transmiten vía radio de manera directa entre las motas IoT, así que, tras consultar las direcciones IPv6 link-local al final del paso 10 del montaje, podemos probar la conectividad a nivel de red con el envío de mensajes `ping`.

Las salidas obtenidas son equivalentes a las de las figuras 5.1 y 5.2, cambiando el rol de cada componente pues las direcciones IPv6 consideradas para el cliente y el servidor con libCoAP son las que tomamos para el servidor y el cliente en GCoAP, respectivamente. En todo caso, la consecución de estas pruebas confirma el correcto establecimiento de la conectividad IPv6 y que los dispositivos IoT se han descubierto entre sí a través de la red 6LoWPAN a la que los dos pertenecen.

Queda entonces construir un mensaje CoAP desde el cliente al servidor, con la misma estructura que las peticiones anteriores, salvo por el recurso que se demanda, que esta vez es "./well-known/core". Observemos que, en el escenario entre la mota IoT y el PC, nos apoyamos en las direcciones IPv6 globales que configuramos durante el montaje, pero en esta situación (y en el primero de los escenarios montados en RIOT) basta con hacer uso de las direcciones IPv6 de tipo link-local.

Dado que el servidor CoAP está en escucha (como comprobamos a propósito de la figura 4.40), desde la terminal del cliente podemos enviar su solicitud hacia la dirección IPv6 del servidor.

```
> coapc coap://[fe80::64b3:7881:5633:4d32]/.well-known/core
2025-04-14 19:51:33,625 # coapc coap://[fe80::64b3:7881:5633:4d32]/.well-known/core
2025-04-14 19:51:33,639 # </time>;if="clock";rt="ticks";ct=0;obs
```

Figura 5.18: Solicitud CoAP por los recursos disponibles en el servidor libCoAP

La respuesta del servidor nos informa de que, entre los recursos disponibles, encontramos únicamente el recurso `time`. Este recurso es observable y representa el tiempo en ticks en texto plano. De esta manera, recibiendo la respuesta esperada de acuerdo a la definición de la aplicación "libcoap_server", hemos conseguido reflejar el escenario entre dos motas IoT con libCoAP, donde, tal y como hemos mencionado, los fundamentos teóricos son idénticos a los que se explicaron en el diseño del escenario general entre dos dispositivos IoT visto en la sección 4.3.

A continuación, mostramos la panorámica de la traza correspondiente a todas las pruebas desarrolladas en este escenario, las cuales capturamos desde *Wireshark* gracias a la incorporación del nRF52840 Dongle como Sniffer llevada a cabo en el montaje.

1	0.000000	fe80::d8c0:942f:47:4…	ff02::2	ICMPv6	71 Router Solicitation from da:c0:94:2f:00:47:48:27
2	17.025756	fe80::64b3:7881:5633…	ff02::2	ICMPv6	71 Router Solicitation from 66:b3:78:81:56:33:4d:32
3	37.634819	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) request id=0x2064, seq=0, hop limit=64 (reply in 5)
4	37.636522			IEEE 802.15.4	31 Ack
5	37.639244	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) reply id=0x2064, seq=0, hop limit=64 (request in 3)
6	37.640946			IEEE 802.15.4	31 Ack
7	38.633803	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) request id=0x2064, seq=1, hop limit=64 (reply in 9)
8	38.635505			IEEE 802.15.4	31 Ack
9	38.638414	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) reply id=0x2064, seq=1, hop limit=64 (request in 7)
10	38.640117			IEEE 802.15.4	31 Ack
11	39.636122	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) request id=0x2064, seq=2, hop limit=64 (reply in 13)
12	39.637825			IEEE 802.15.4	31 Ack
13	39.639133	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) reply id=0x2064, seq=2, hop limit=64 (request in 11)
14	39.640835			IEEE 802.15.4	31 Ack
15	56.366263	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) request id=0x722e, seq=0, hop limit=64 (reply in 17)
16	56.367966			IEEE 802.15.4	31 Ack
17	56.369273	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) reply id=0x722e, seq=0, hop limit=64 (request in 15)
18	56.370976			IEEE 802.15.4	31 Ack
19	57.368255	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) request id=0x722e, seq=1, hop limit=64 (reply in 21)
20	57.369958			IEEE 802.15.4	31 Ack
21	57.372543	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) reply id=0x722e, seq=1, hop limit=64 (request in 19)
22	57.374246			IEEE 802.15.4	31 Ack
23	58.367175	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	ICMPv6	64 Echo (ping) request id=0x722e, seq=2, hop limit=64 (reply in 25)
24	58.368878			IEEE 802.15.4	31 Ack
25	58.371916	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	ICMPv6	64 Echo (ping) reply id=0x722e, seq=2, hop limit=64 (request in 23)
26	58.373618			IEEE 802.15.4	31 Ack
27	60.001530	fe80::d8c0:942f:47:4…	ff02::2	ICMPv6	71 Router Solicitation from da:c0:94:2f:00:47:48:27
28	77.028662	fe80::64b3:7881:5633…	ff02::2	ICMPv6	71 Router Solicitation from 66:b3:78:81:56:33:4d:32
29	91.793707	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	CoAP	86 CON, MID:38341, GET, /.well-known/core
30	91.796114			IEEE 802.15.4	31 Ack
31	91.799148	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	CoAP	103 ACK, MID:38341, 2.05 Content
32	91.802099			IEEE 802.15.4	31 Ack
33	97.190400	fe80::d8c0:942f:47:4…	fe80::64b3:7881:5633…	CoAP	80 CON, MID:28377, GET, /time?ticks
34	97.192615			IEEE 802.15.4	31 Ack
35	97.196112	fe80::64b3:7881:5633…	fe80::d8c0:942f:47:4…	CoAP	70 ACK, MID:28377, 2.05 Content (text/plain)
36	97.198007			IEEE 802.15.4	31 Ack

Figura 5.19: Flujo de mensajes del escenario entre dos motas IoT con libCoAP

En primer lugar, observamos dos mensajes *Router Solicitation*, con números 1 y 2 en la traza, enviados cada uno por una de las motas a la dirección multicast ff02::2. En ICMPv6, un mensaje de este tipo es transmitido por un nodo periódicamente (por ello vuelven a aparecer en los mensajes 27 y 28 de la figura 5.19) para solicitar información de enrutamiento al router, tal y como el prefijo de red, con el que dicho nodo puede configurar su dirección IPv6.

A continuación, se producen los envíos de mensajes *ICMPv6 Echo Request* e *ICMPv6 Echo Reply* entre las motas. Los mensajes con números 3, 7 y 11 son los `ping` enviados por la mota servidora, con las respuestas de la mota cliente en los mensajes 5, 9 y 13, y en sentido inverso tenemos los *ping* con números 15, 19 y 23, y sus correspondientes respuestas en los mensajes 17, 21 y 25. Cada vez que se transmite uno de estos mensajes ICMPv6, aparece de manera intercalada un *IEEE 802.15.4 Ack*, que confirma la correcta recepción de la trama en la capa de enlace. Si analizamos el contenido de un mensaje *ICMPv6 Echo Request*, tenemos:

Figura 5.20: Contenido de un *ICMPv6 Echo Request* en el escenario entre dos motas IoT con libCoAP

Wireshark nos indica en el frame que se ha capturado el paquete en el puerto "/dev/ttyACM0", donde escucha el Sniffer. Después, encontramos la información de las cabeceras. La del nivel de enlace sigue el estándar IEEE 802.15.4 y nos indica las direcciones MAC origen y destino, así como el identificador de la PAN (0x0023), entre otros datos. Seguidamente, tenemos la cabecera 6LoWPAN, que encapsula la cabecera IPv6. Ambas poseen las mismas direcciones IPv6 origen y destino (recordemos que ya mencionamos este aspecto a lo largo del capítulo 4).

Dentro de la cabecera IPv6, el campo *Next Header* especifica el tipo de la siguiente cabecera, que corresponde a la del mensaje ICMPv6. En este último, se comparte la información por la que se verifica la conectividad entre los nodos. Por otro lado, el contenido de un *ICMPv6 Echo Reply* es similar, con las debidas permutas en direcciones.

El flujo de la figura 5.19 concluye con el envío de dos solicitudes CoAP. Nótese que, respecto a las pruebas seguidas en esta misma sección, hemos añadido una petición adicional por el recurso "/time?ticks", que devuelve el valor actual de los ticks del sistema (donde un tick es una medida de tiempo en unidades del reloj interno del sistema operativo). Por la similitud entre el contenido de los mensajes CoAP, nos fijamos exclusivamente en la información que nos ofrece la última respuesta CoAP, con número 35 en la traza.

Figura 5.21: Contenido de una respuesta CoAP en el escenario entre dos motas IoT

De nuevo, a nivel de enlace, se usa IEEE 802.15.4, y ahora 6LoWPAN encapsula las cabeceras IPv6, UDP y CoAP. De hecho, la cabecera 6LoWPAN indica, aparte de las direcciones IP, los puertos UDP origen y destino. Observemos que, como el servidor CoAP escucha en el puerto 5683, responde a las peticiones desde él. En cuanto al paquete IPv6, esta vez *Next Header* apunta al datagrama UDP, que transporta la respuesta CoAP.

En el contenido del mensaje CoAP, vemos que es una respuesta con asentimiento (*Type: Acknowledgement (2)*) y encontramos el identificador *Message ID*, que la relaciona con la petición. Esto coincide con el comportamiento de la aplicación CoAP descrita en la sección 4.3. De hecho, si observamos la petición asociada a esta respuesta:

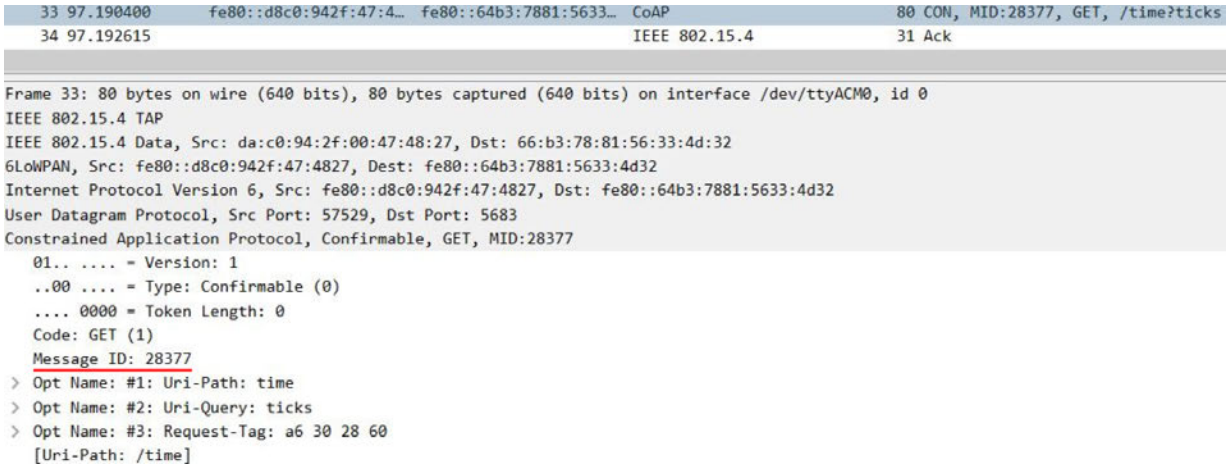


Figura 5.22: Contenido de la petición CoAP asociada a la respuesta de la figura 5.21

Verificamos que el *Message ID* es efectivamente el mismo. Por último, en la figura 5.21, se especifica el formato del payload devuelto y su contenido, que es el número de ticks solicitado, igual a 1582.

La razón principal detrás de este análisis acerca del tráfico capturado es comprobar que el formato de cabeceras de los paquetes ICMPv6 y CoAP es exactamente aquel que se ha descrito en las figuras 4.6 y 4.7, respectivamente. Así, queda reflejado que el diseño teórico del escenario no solo se ha trasladado al ámbito del sistema operativo RIOT con éxito gracias al correcto envío de los `ping` y los mensajes CoAP, sino que también el contenido de estos paquetes es acorde a lo que se espera según los protocolos y estándares de los que estamos haciendo uso en nuestro modelo particular de la capa OSI.

6. Conclusiones y vías futuras

El desarrollo de este trabajo ha supuesto el estudio teórico, despliegue y puesta en marcha de pruebas de diversos escenarios de red sujetos a los protocolos IEEE 802.15.4, 6LoWPAN, IPv6 y CoAP para lograr el establecimiento de comunicaciones a nivel de red y aplicación entre dispositivos IoT, y entre dispositivos de este tipo con un PC externo, utilizando como elemento central el sistema operativo RIOT. Para ello, se ha llevado a cabo un análisis exhaustivo y profundo, partiendo de una visión de RIOT desde cero hasta adquirir el aprendizaje acerca de cómo realizar los montajes y sus correspondientes evaluaciones tal y como se han descrito en los dos capítulos anteriores.

Para la consecución de los objetivos propuestos, a lo largo de este trabajo se ha procedido a introducir las características básicas de RIOT y otros sistemas operativos dedicados a IoT, a profundizar en la definición de la pila de red GNRC y a indagar en el funcionamiento de las aplicaciones en RIOT, aprendiendo a compilarlas y a interactuar con sus líneas de comandos por medio del uso de *pyterm*.

Además, se han explicado detalladamente las tecnologías utilizadas para diseñar cada uno de los escenarios, así como las herramientas software necesarias para compilar aplicaciones de RIOT sobre dos tipos de placas específicas de Nordic Semiconductor, todo ello siguiendo un enfoque orientado a empezar a trabajar con RIOT sin la suposición de ningún conocimiento previo acerca del entorno asociado a este sistema operativo.

Este último aspecto resulta de gran interés, pues hace que el presente TFG inicie a investigadores noveles en el ámbito de los sistemas operativos dedicados a dispositivos IoT y, en particular, da una guía para que cualquiera que quiera interactuar por primera vez con RIOT, aparte de familiarizarse con el uso de dispositivos hardware físicos como los de Nordic, pueda beneficiarse de la experiencia detrás de la realización de este trabajo y desarrollar sus propios escenarios de red para montar comunicaciones realistas en redes de bajo consumo.

Igualmente, recordemos que este trabajo concentra y sintetiza una gran cantidad de información dispersa sobre cómo desplegar escenarios de red en el sistema operativo RIOT, haciendo uso de un modelo OSI completo de acuerdo a la pila de red GNRC. Para ello, se han resuelto los problemas derivados de integrar todas las fuentes de información, se ha identificado la forma adecuada para establecer un Border Router con base en el material disponible y se han descubierto las modificaciones a realizar para que las aplicaciones software elegidas pudieran configurarse y funcionar de manera conjunta.

Asimismo, gracias al despliegue y a las pruebas realizadas sobre los escenarios, se ha podido estudiar de primera mano cómo funciona la capa OSI considerada para redes inalámbricas. En concreto, se ha analizado cada una de las capas que componen los paquetes ICMPv6 y CoAP, entrando en detalle sobre la forma en la que se transmiten los mensajes a nivel de enlace por IEEE 802.15.4 y Ethernet, al igual que se ha estudiado el estado de las direcciones MAC e IPv6 cuando 6LoWPAN encapsula un paquete, ya tenga la red 6LoWPAN funcionalidad de enrutamiento gracias al soporte de RPL o no.

Por otra parte, mediante la configuración de un Border Router para redes 6LoWPAN que asigna automáticamente direcciones IPv6 globales a otros nodos de dicha red y ejerce de gateway a redes IPv6 públicas, este trabajo profundiza en una de las áreas de estudio prioritarias en la comunidad de RIOT [59], y también verifica al mismo tiempo que la pila de red GNRC está correctamente implementada.

Por último, es preciso recalcar algunas de las posibles vías que permitirán la evolución de este trabajo, cuyas aportaciones constituyen un punto de partida para futuras investigaciones relativas a RIOT:

- El proceso seguido para el montaje de cada escenario de red puede tomarse como referencia para el diseño de escenarios más ambiciosos, orientados a gran escala y que involucren, a modo de ejemplo, la participación de un número mucho mayor de nodos en la red, o que adopten políticas restrictivas en términos de calidad de servicio.

- Notemos que, en el segundo escenario desarrollado con la comunicación extremo a extremo entre una mota IoT y un PC, este último equipo actúa de gateway entre el dispositivo IoT y redes IPv6 que podrían conectar con Internet o redes Wi-Fi. Así, este escenario sirve de base a otros modelos que tengan como objetivo extenderlo para que la mota IoT sea capaz de recibir comandos desde interfaces web, enviar datos a servicios en la nube o se integre en redes totalmente basadas en IP.

- A partir de los escenarios de red funcionales que se han desplegado, pueden llevarse a cabo estudios acerca de la integración en RIOT de otros protocolos que utilicen IPv6 y CoAP como el protocolo de seguridad Ephemeral Diffie-Hellman Over COSE (EDHOC) [60], o análisis que se centren en evaluar en profundidad la latencia, los fallos en la transmisión de paquetes o el nivel de consumo energético que se registran en comunicaciones con IPv6 y CoAP dentro del sistema operativo RIOT, incluso restringiéndose dicho estudio simplemente al rendimiento de la solución propuesta en este trabajo. Es por ello que uno de los aspectos en los que este TFG ayuda en gran medida a proyectos de RIOT que estén por venir es porque indica las herramientas para configurar dispositivos IoT que ejerzan de Sniffer y, de esta manera, se obtengan capturas con el contenido de los mensajes transmitidos de cara a su análisis.

- Para ambos escenarios, se puede perseguir el objetivo de añadir seguridad a las comunicaciones en RIOT y estudiar cómo podría llegar a implementarse OSCORE correctamente, o qué soluciones ofrece la pila GNRC mediante el soporte de GCoAP para proporcionar seguridad a nivel de objeto para cada uno de los mensajes que se intercambian en la comunicación.

Bibliografía

[1] *TinyOS*. URL: https://github.com/tinyos.

[2] Oikonomou G. et al. *The Contiki-NG open source operating system for next generation IoT devices*. SoftwareX, Volume 18, 2022. URL: https://doi.org/10.1016/j.softx.2022.101089.

[3] RIOT - The friendly Operating System for the Internet of Things. *RIOT Documentation*. URL: https://doc.riot-os.org/.

[4] Zephyr. *Zephyr Project*. URL: https://www.zephyrproject.org/.

[5] Hinden B. y Deering S. E. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. 1998. URL: https://datatracker.ietf.org/doc/rfc2460.

[6] Montenegro G., Schumacher C. y Kushalnagar N. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*. RFC 4919. 2007. URL: https://datatracker.ietf.org/doc/rfc4919.

[7] Alexander R. et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. 2012. URL: https://datatracker.ietf.org/doc/rfc6550.

[8] Shelby Z., Hartke K. y Bormann C. *The Constrained Application Protocol (CoAP)*. RFC 7252. 2014. URL: https://datatracker.ietf.org/doc/rfc7252.

[9] López G. *Thread Network with Zephyr and nrf52840dk and dongle*. 2024. URL: https://gitlab.com/gabilm/thread-network-with-zephyr-and-nrf52840dk-and-dongle.

[10] Thread Group. *Thread Network Fundamentals*. 2020. URL: https://www.threadgroup.org/Portals/0/documents/support/Thread%20Network%20Fundamentals_v3.pdf.

[11] RIOT - The friendly Operating System for the Internet of Things. *Generic (GNRC) network stack*. URL: https://doc.riot-os.org/group__net__gnrc.html#details.

[12] Nordic Semiconductor. *nRF52840 DK*. URL: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK.

[13] Nordic Semiconductor. *nRF52840 Dongle*. URL: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle.

[14] Montenegro G. et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. 2007. URL: https://datatracker.ietf.org/doc/rfc4944.

[15] ZigBee Alliance. *ZigBee Specification*. 2015. URL: https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf.

[16] Kempf J. et al. *SEcure Neighbor Discovery (SEND)*. RFC 3971. 2005. URL: https://datatracker.ietf.org/doc/rfc3971.

[17] J. Mohácsi et al. *IPv6 Router Advertisement Guard*. RFC 6105. 2011. URL: https://datatracker.ietf.org/doc/rfc6105.

[18] Gupta M. y Conta A. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443. 2006. URL: https://datatracker.ietf.org/doc/rfc4443.

[19] Althalji A., Khawatmi S. y Kassab B. A. *Mobility-Aware Routing for Low Power and Lossy Networks*. Syrian Journal for Science e Innovation, Volume 1, 2023. URL: https://journal.hcsr.gov.sy/archives/210.

[20] Nielsen H. et al. *Hypertext Transfer Protocol - HTTP/1.1*. RFC 2616. 1999. URL: https://datatracker.ietf.org/doc/rfc2616.

[21] Bassi A. *Goto IoT - Introducción a CoAP*. 2021. URL: https://www.gotoiot.com/pages/articles/coap_intro/index.html.

[22] Eddy W. *Transmission Control Protocol (TCP)*. RFC 9293. 2022. URL: https://datatracker.ietf.org/doc/rfc9293.

[23] Postel J. *User Datagram Protocol*. RFC 768. 1980. URL: https://datatracker.ietf.org/doc/rfc768.

[24] Millán A. *Estudio de implementaciones del Protocolo CoAP*. Escuela Técnica Superior de Ingeniería, Universidad de Sevilla, 2019. URL: https://iotmadlab.es/wp-content/uploads/2023/10/TFG-2514-MILLAN-YANES.pdf.

[25] Hahm O. *Operating Systems - RIOT: An Open OS for an Open IoT*. Frankfurt University of Applied Sciences, 2022. URL: https://www.christianbaun.de/BTS2122/Skript/opsys-ws21-riot.pdf.

[26] Baccelli E. et al. *RIOT: One OS to Rule Them All in the IoT*. Research Centre Saclay, 2012. URL: https://www.cs.virginia.edu/~bjc8c/class/cs6501-f18/papers/baccelli12riot.pdf.

[27] *Contiki*. Wikipedia, la enciclopedia libre, 2023. URL: https://es.wikipedia.org/w/index.php?title=Contiki&oldid=152449944.

[28] Zephyr. *Introduction - Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/introduction/index.html.

[29] OpenThread, released by Google. *OpenThread*. URL: https://openthread.io/?hl=es-419.

[30] Baccelli E. et al. *RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT*. IEEE Internet of Things Journal, Volume 5, 2018. URL: https://ieeexplore.ieee.org/document/8315125/.

[31] Kumar T. S. P. *RIOT: An Operating System for the IoT*. Open Source for You. 2016. URL: https://www.opensourceforu.com/2016/03/riot-an-operating-system-for-the-iot/.

[32] Hahm O. y Lenders M. *RIOT-OS/Tutorials*. URL: https://github.com/RIOT-OS/Tutorials.

[33] *RIOT/examples/basic/hello-world*. URL: https://github.com/RIOT-OS/RIOT/tree/master/examples/basic/hello-world.

[34] Hahm O. *Working with RIOT on the nrf52840dk in the lab*. 2024. URL: https://teaching.dahahm.de/teaching/ss24/iot/2024/05/03/working_with_RIOT_on_nrf52840dk_in_the_lab.html.

[35] *RIOT/examples/networking/gnrc/gnrc_networking*. URL: https://github.com/RIOT-OS/RIOT/tree/master/examples/networking/gnrc/gnrc_networking.

[36] RIOT - The friendly Operating System for the Internet of Things. *GCoAP*. URL: https://doc.riot-os.org/group__net__gcoap.html.

[37] Bergmann O. *C-Implementation of CoAP*. 2015. URL: https://libcoap.net/.

[38] *RIOT/examples/basic/blinky*. URL: https://github.com/RIOT-OS/RIOT/tree/master/examples/basic/blinky.

[39] Simpson W. A. et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. 2007. URL: https://datatracker.ietf.org/doc/rfc4861.

[40] Zandberg K. *Border router using USB CDC ECM (ethernet over USB)*. 2019. URL: https://github.com/RIOT-OS/RIOT/wiki/Border-router-using-USB-CDC-ECM-(ethernet-over-USB).

[41] XMOS. *USB CDC-ECM Class for Ethernet over USB*. 2016. URL: https://www.xmos.com/download/AN00131:-USB-CDC-ECM-Class-for-Ethernet-over-USB(2.0.2rc1).pdf/.

[42] *RIOT/examples/networking/gnrc/gnrc_border_router*. URL: https://github.com/RIOT-OS/RIOT/tree/master/examples/networking/gnrc/gnrc_border_router.

[43] Mockapetris P. *Domain names - Implementation and specification*. RFC 1035. 1987. URL: https://datatracker.ietf.org/doc/rfc1035.

[44] R. Droms. *Dynamic Host Configuration Protocol*. RFC 2131. 1997. URL: https://datatracker.ietf.org/doc/rfc2131.

[45] *RIOT/examples/networking/coap/gcoap*. URL: https://github.com/RIOT-OS/RIOT/tree/master/examples/networking/coap/gcoap.

[46] *mrdeep1/RIOT at libcoap_add*. URL: https://github.com/mrdeep1/RIOT/tree/libcoap_add.

[47] Selander G. et al. *Object Security for Constrained RESTful Environments (OSCORE)*. RFC 8613. 2019. URL: https://datatracker.ietf.org/doc/rfc8613.

[48] *coap-security/liboscore*. 2025. URL: https://github.com/coap-security/liboscore.

[49] Rosenow L. J. *Integration of OSCORE into RIOT OS*. Hamburg University of Applied Sciences, 2023. URL: https://inet.haw-hamburg.de/teaching/ws-2023-24/project-class/fw1__lasse_rosenow.pdf.

[50] RIOT - The friendly Operating System for the Internet of Things. *Getting started*. URL: https://doc.riot-os.org/getting-started.html.

[51] Nordic Semiconductor. *nRF Util*. 2025. URL: https://docs.nordicsemi.com/bundle/nrfutil/page/README.html.

[52] Segger. *SEGGER J-Link debug probes*. URL: https://www.segger.com/products/debug-probes/j-link/.

[53] Nordic Semiconductor. *nRF Connect for Desktop*. URL: https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop.

[54] CompTIA. *What Is Wireshark and How to Use It - Cybersecurity*. URL: https://www.comptia.org/content/articles/what-is-wireshark-and-how-to-use-it.

[55] Nordic Semiconductor. *nRF Sniffer for 802.15.4*. URL: https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-802154.

[56] PySquad. *Explore PySerial: Serial Communication Libraries*. 2024. URL: https://medium.com/@pysquad/explore-pyserial-serial-communication-libraries-e79b32a6dfe7.

[57] Cufí C. et al. *NordicSemiconductor/pc-nrfutil*. URL: https://github.com/NordicSemiconductor/pc-nrfutil.

[58] Segger. *SEGGER - The Embedded Experts - Downloads - J-Link / J-Trace*. URL: https://www.segger.com/downloads/jlink/.

[59] RIOT. *Roadmap*. URL: https://doc.riot-os.org/roadmap.html.

[60] Selander G., Mattsson J. P. y Palombini F. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. RFC 9528. 2024. URL: https://datatracker.ietf.org/doc/rfc9528.

Anexo

I. Descripción de las dependencias de RIOT

- `git`: este es un sistema software que permite clonar repositorios, gestionando el código asociado, sus versiones y ramas, entre otras funcionalidades.
- `gcc-arm-none-eabi`: se trata de un compilador para microcontroladores Advanced RISC Machine (ARM) sin sistema operativo.
- `make`: es una herramienta que automatiza la compilación de proyectos.
- `gcc-multilib`: es un compilador de programas de 32 bits en sistemas de 64 bits.
- `libstdc++-arm-none-eabi-newlib`: corresponde a las librerías estándar de C++ y C para ARM.
- `openocd`: es una herramienta orientada a la programación y depuración en microcontroladores.
- `gdb-multiarch`: es un tipo de GNU Debugger (GDB) que soporta múltiples arquitecturas, incluyendo ARM, para la identificación y solución de errores en los programas.
- `doxygen`: es un generador automático de documentación de código fuente.
- `wget`: es una utilidad para la descarga de archivos desde Internet a través de la terminal en diversos sistemas operativos.
- `unzip`: se trata de un programa que descomprime archivos de extensión ".zip".
- `python3-serial`: es el módulo de Python encargado del establecimiento de comunicaciones serie.
- `python3-psutil`: es el módulo de Python que gestiona el acceso a la información del sistema.

Listado de Acrónimos y Abreviaturas

- **API**: Application Programming Interface.
- **ARM**: Advanced RISC Machine.
- **CDC-ECM**: Communication Device Class-Ethernet Control Model.
- **CoAP**: Constrained Application Protocol.
- **CPU**: Central Processing Unit.
- **DAG**: Directed Acyclic Graph.
- **DAO**: Destination Advertisement Object.
- **DFU**: Device Firmware Update.
- **DHCP**: Dynamic Host Configuration Protocol.
- **DIO**: DODAG Information Object.
- **DK**: Development Kit.
- **DNS**: Domain Name System.
- **DODAG**: Destination Oriented DAG.
- **EDHOC**: Ephemeral Diffie-Hellman Over COSE.
- **EUI**: Extended Unique Identifier.
- **GDB**: GNU Debugger.
- **GNRC**: Generic.
- **HTTP**: Hypertext Transfer Protocol.
- **ICMPv6**: Internet Control Message Protocol version 6.
- **IDE**: Integrated Development Environment.
- **IEEE**: Institute of Electrical and Electronics Engineers.
- **IoT**: Internet of Things.
- **IPv4**: Internet Protocol version 4.
- **IPv6**: Internet Protocol version 6.

- **IPC**: Inter-Process Communication.
- **ISM**: Industrial, Scientific and Medical.
- **KISS**: Keep It Simple, Stupid.
- **LAN**: Local Area Network.
- **LED**: Light Emitting Diode.
- **MAC**: Media Access Control.
- **MCU**: Microcontroller Unit.
- **MMU**: Memory Management Unit.
- **NAT**: Network Address Translation.
- **OSCORE**: Object Security for Constrained RESTful Environments.
- **OSI**: Open Systems Interconnection.
- **PAN**: Personal Area Network.
- **PC**: Personal Computer.
- **RA-Guard**: Router Advertisement-Guard.
- **RAM**: Random Access Memory.
- **REST**: Representational State Transfer.
- **RGB**: Red, Green, Blue.
- **ROM**: Read Only Memory.
- **RPL**: Routing Protocol for Low Power and Lossy Networks.
- **SDK**: Software Development Kit.
- **SEND**: Safe Neighbor Discovery Protocol.
- **SoC**: System-on-a-Chip.
- **TCP**: Transmission Control Protocol.
- **TLS**: Transport Layer Security.
- **TKL**: Token Length.
- **UDP**: User Datagram Protocol.
- **URL**: Uniform Resource Locator.
- **USB**: Universal Serial Bus.
- **VDD**: Voltage Digital Drain.
- **WPAN**: Wireless Personal Area Network.
- **6LoWPAN**: IPv6 over Low-Power Wireless Personal Area Networks.