Celia Llanes Rodríguez

**MEMORIA DE PRÁCTICAS**

Dispositivos IoT (MUT)

# Tabla de contenido

[Tabla de contenido 1](#_Toc154146232)

[Práctica1 2](#_Toc154146233)

[Ejercicio 1 2](#_Toc154146234)

[Ejercicio 2 3](#_Toc154146235)

[Ejercicio 3: 5](#_Toc154146236)

[Ejercicio 4: 6](#_Toc154146237)

[Ejercicio 5 7](#_Toc154146238)

[Práctica 2 9](#_Toc154146239)

[Ejercicio 1 9](#_Toc154146240)

[Ejercicio 2 10](#_Toc154146241)

[Ejercicio 3 13](#_Toc154146242)

[Práctica 3 15](#_Toc154146243)

[Ejercicio 1 15](#_Toc154146244)

[Ejercicio 2 16](#_Toc154146245)

[Ejercicio 3 17](#_Toc154146246)

[Práctica 4 19](#_Toc154146247)

# Práctica1

## Ejercicio 1

**Localice y escriba la ruta de los drivers de cada uno de los chips que componen la placa que se utiliza en las prácticas. Incluya tanto los ficheros fuente como los ficheros de cabecera. Incluya esta información en la memoria final de prácticas.**

* **Microcontrolador**

/home/vmu/work/contiki-ng/arch/cpu/nrf52840

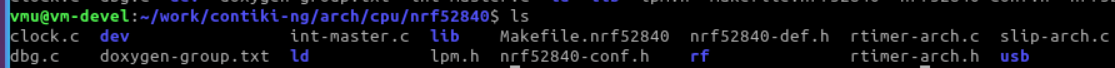
****

Figura 1. Ficheros del microcontrolador

* **Transceiver radio**

/home/vmu/work/contiki-ng/arch/dev/radio

****

Figura 2. Ficheros del transreceiver radio

* **Almacenamiento externo**

/home/vmu/work/contiki-ng/arch/dev/storage/ext-flash

****

Figura 3. Ficheros del almacenamiento externo

* **Almacenamiento interno**

/home/vmu/work/contiki-ng/arch/dev/storage/disk/mmc

****

Figura 4. Ficheros del almacenamiento interno

## Ejercicio 2

**Realice el Test inicial: Hello World del repositorio de instalación (**[**https://gitlab.com/laboratorio-ris/contiki-ng**](https://gitlab.com/laboratorio-ris/contiki-ng)**). Adjunte las capturas necesarias para mostrar la compilación, la carga del programa y la ejecución del mismo.**

* **Compilación:**

Texto

Descripción generada automáticamente

Figura 5. Compilación Hello World

* + **Carga del programa:**

**Texto

Descripción generada automáticamente**

Figura 6. Carga del programa Hello World

* **Ejecución del programa:**

**Texto

Descripción generada automáticamente**

Figura 7. Ejecución del programa Hello World

## Ejercicio 3:

**Realice una aplicación “Hello World” modificada de forma que ahora existan dos procesos:**

* **El original hello\_world\_process, que sólo se limitará a esperar eventos para imprimir un mensaje con un contador que se incrementará en cada evento recibido. Este contador llegará hasta 20, y posteriormente se reiniciará.**
* **Un nuevo proceso que se llamará periodic\_process. Se ejecutará desde el principio junto con el anterior. Generará un evento cada 5 segundos y será enviado hacia el proceso hello\_world\_process utilizando la función process\_poll.**

El código se encuentra comentado en: ***DIoT-practicas/practica1/ejercicio3***. Además, en la carpeta de la práctica podemos encontrar un vídeo que muestra el programa en funcionamiento corriendo en el dispositivo, de este modo podemos ver que se genera el evento cada 5 segundos y que el contador de eventos se reinicia al llegar a 20. A continuación, se presenta una captura de la ejecución de la aplicación.

Texto

Descripción generada automáticamente

Figura 8. Ejecución práctica 1 - ejercicio 3

## Ejercicio 4:

**Realice la aplicación “Blink” que conmute los dos les de la placa cada 2 y 3 segundos respectivamente. Para ello se hará uso de 3 procesos/protohilos:**

* **parpadeo\_1\_process: Se encarga de conmutar el estado del Led 1 cada 2 segundos. Este proceso permanecerá esperando un evento del proceso timer\_process antes de realizar la primera conmutación.**
* **parpadeo\_2\_process: Se encarga de conmutar el estado del Led 2 cada 3 segundos. Este proceso permanecerá esperando un evento del proceso timer\_process antes de realizar la primera conmutación.**
* **timer\_process: Proceso que se ejecutará al inicio de la aplicación y tras esperar 5 segundos enviará un evento process\_poll a Parpadeo\_1 \_process y Parpadeo\_2\_process.**

El código se encuentra comentado en: ***DIoT-practicas/practica1/ejercicio4***. Además, en la carpeta de la práctica podemos encontrar un vídeo que muestra el programa en funcionamiento corriendo en el dispositivo, de este modo podemos ver que, tras 5 segundos en espera de la recepción del evento de proceso *“timer\_process”,* los procesos que hacen parpadear los leds se activan. A continuación, el proceso *“parpadeo\_led\_1\_proccess”* conmuta el Led1 cada 2 segundos y el proceso *“parpadeo\_led\_2\_proccess”* conmuta el Led2 cada 3 segundos.

Cabe destacar que para hacer que el led parpadee, se ha usado la función *“****void leds\_single\_toggle(leds\_num\_t led)”*** de la librería ***dev/leds.h***

En la siguiente captura podemos ver los logs de la aplicación en ejecución:

Texto

Descripción generada automáticamente

Figura 9. Ejecución práctica 1 - ejercicio 4

## Ejercicio 5

**Realice la aplicación “Temperature” que lea la temperatura de la placa cada 3 segundos y la imprima por pantalla. Utilice un proceso para la lectura del sensor y otro para la temporización de los 3 segundos.**

El código se encuentra comentado en: ***DIoT-practicas/practica1/ejercicio5***. Además, en la carpeta de la práctica podemos encontrar un vídeo que muestra el programa en funcionamiento corriendo en el dongle.

La lectura de la temperatura se realiza extrayendo el parámetro *"value(0)"* de la estructura *"temperatura\_sensor"*, la cual está definida en la librería *"arch/platform/nrf52840/common/temperature-sensor.h".* Además, esta estructura pertenece al tipo *"sensors\_sensor"* definido en librería *"os/lib/sensors.h".*

Cabe destacar que el sensor de temperatura de nuestro dispositivo tiene una resolución de 0.25. Además, la función encargada de leer la temperatura devuelve un valor cuyo tipo es *“int”*. De este modo, para imprimir la temperatura real en formato decimal se ha utilizado la siguiente línea de código:

***printf("[temperature\_sensor\_process] Temperatura leída: %d.%d ºC\n", temp/4, temp%4\*25);***

La obtención de la parte entera de la temperatura se logra multiplicando el valor de temperatura por la resolución (0.25), lo cual es equivalente a dividir entre 4. Por otro lado, para obtener la parte decimal, se calcula el resto de la división entre 4 y se multiplica por la resolución, seguido de una multiplicación por 100.

En la siguiente captura podemos ver los logs de la aplicación en ejecución. Vemos que la temperatura va aumentando y disminuyendo porque se ha ido aumentando la temperatura del exterior a modo de prueba.

Texto

Descripción generada automáticamente con confianza media

Figura 10. Ejecución práctica 1 - ejercicio 5

# Práctica 2

## Ejercicio 1

**En base a los ejercicios desarrollados en la Práctica 1, desarrollar una aplicación que conste de 3 tareas que realicen la siguiente función (el alumno pude modificar el nombre de las tareas):**

* **Tarea 1: Espera el evento de la Tarea 3 para comenzar un parpadeo del led 1 (leds\_toggle(LEDS\_GREEN)) cada 2 segundos.**
* **Tarea 2: Espera el evento de la Tarea 3 para comenzar un parpadeo del led 1 (leds\_toggle (LEDS\_YELLOW)) cada 4 segundos.**
* **Tarea 3: Dispone de un timer de 3 segundos que, tras expirar lanza un evento a las tareas “Tarea 1” y “Tarea 2”**

El código se encuentra comentado en: **DIoT-practicas/practica2/ejercicio1**. Además, en la carpeta de la práctica podemos encontrar un vídeo que muestra el programa en funcionamiento corriendo en el dispositivo, de este modo podemos ver que, tras 5 segundos en espera de la recepción del evento del proceso *“timer\_process”,* los procesos que hacen parpadear los leds se activan. A continuación, el proceso *“parpadeo\_led\_1\_proccess”* conmuta el Led1 cada 2 segundos y el proceso *“parpadeo\_led\_2\_proccess”* conmuta el Led2 cada 4 segundos.

Este ejercicio se ha ejecutado en el simulador MSPSIM, que tiene la capacidad de simular la placa Z1 sin necesidad de disponer del hardware.

A continuación, se adjunta una captura de la simulación.

Interfaz de usuario gráfica, Texto

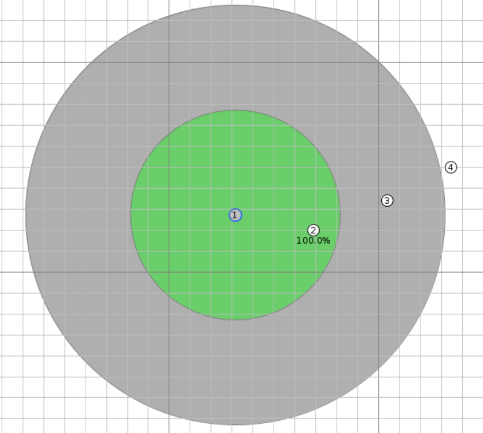
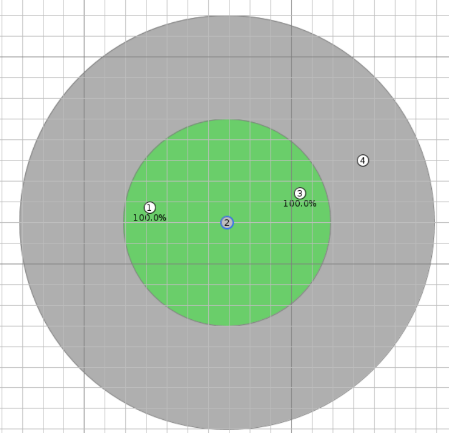
Descripción generada automáticamente

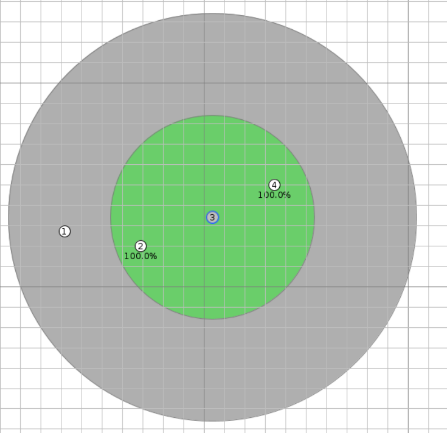
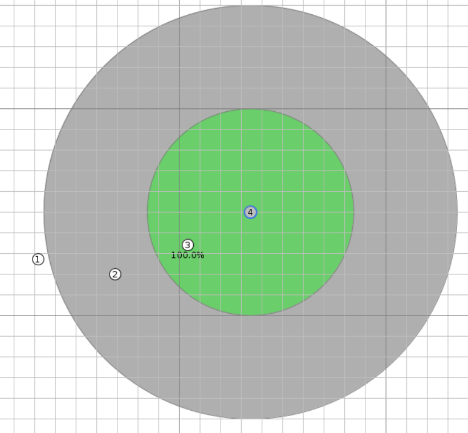
Figura 11. Ejecución práctica 2 - ejercicio 1

## Ejercicio 2

**Con el ejemplo de red RPL y nodo frontera simulado en esta práctica, sitúe los 4 nodos de forma que: el nodo (1) sólo tiene cobertura para el (2); el nodo (2) sólo cubre el (1) y el (3); el nodo (3) únicamente es visto por el (2) y el (4). Por último, el nodo (4) solamente tendrá conectividad con el (3).**

En primer lugar, se ha configurado el escenario tal y como se solicita, como podemos ver en las siguientes capturas.

* **Ejecutar la simulación durante cierto tiempo (hasta que todos los nodos tengan conexión con el border-router).**
* **Abrir el socket serie y el navegador con la página principal. ¿Qué se muestra ahora en la web del nodo frontera?**

Tras ejecutar la simulación el tiempo suficiente para que todos los nodos tengan conexión con el border-router, en la web del nodo frontera se muestra la siguiente información.

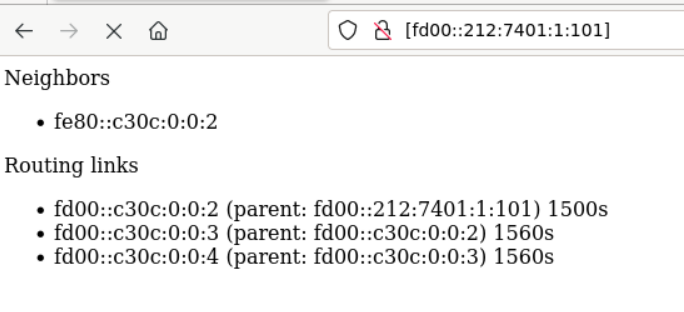


Figura 12. Web nodo frontera

Como era de esperar, se observa que ahora el único nodo al que puede acceder el nodo frontero directamente (vecino) es el nodo 1. Para acceder al resto de nodos (2, 3 y 4) se han configurado tres enlaces de enrutamiento.

* **Interpretación de los archivos de los clientes y el servidor. (adjuntar los .c, .h, makefile y simulación Cooja).**

***Border router:***

**Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

El border router se ejecuta como un proceso denominado *contiki\_ng\_br* que se inicia automáticamente al arrancarse. Lo que realiza es comprobar si se ha habilitado *BORDER\_ROUTER\_CONF\_WEBSERVER*. En ese caso, inicia el proceso *webserver\_nogui\_process* que funciona como un servidor web.

***Nodos terminales:***

*Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente*

El **nodo cliente** se ejecuta como un proceso denominado *udp\_client\_process* que se inicia automáticamente al arrancarse. En primer lugar, inicializa la conexión UDP con el servidor a través de la función *"simple\_udp\_register()"* a la que se le indica el puerto del cliente (*UDP\_CLIENT\_PORT*), el puerto del servidor (*UDP\_SERVER\_PORT*) y la función de callback que se ejecutará cuando se reciba un paquete UDP del servidor. Cabe destacar que en la función de callback se procesa el paquete recibido del servidor.

A continuación, inicializa un temporizador periódico que expira cada 60 segundos. Cada vez que el temporizador expira, se comprueba si el servidor es alcanzable y si es así, se envía un mensaje al mismo mediante la función *"simple\_udp\_sendto()".*

El **nodo servidor** se ejecuta como un proceso denominado *udp\_server\_process* que se inicia automáticamente al arrancarse. Lo que realiza es inicializar la conexión UDP con el cliente mediante la función *“simple\_udp\_register()”* en la que se indica el puerto del servidor (*UDP\_SERVER\_PORT*), el puerto del cliente (*UDP\_CLIENT\_PORT*) y la función de callback que se ejecutará cuando se reciba un paquete UDP. Cabe destacar que en la función de callback se procesa el paquete recibido del cliente y se envía una respuesta al mismo.

* **¿Qué información envían los nodos terminales hacia el router?**

Todos los nodos terminales envían hacia el router un mensaje con el formato "hello %d", donde %d es el valor de la variable *count* que representa el contador de los paquetes “hello” enviados por el nodo. Para ello, utilizan la conexión UDP establecida.

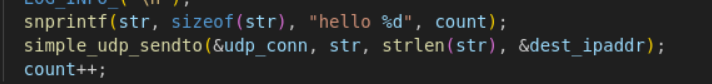


Figura 13. Información enviada por cliente UDP práctica 2 - ejercicio 2

## Ejercicio 3

* **Modifique el código de los nodos terminales (el original está en ~/contiki-ng/examples/rpl-udp/udp-client.c), para que ahora envíen información sobre la temperatura interna y externa del microcontrolador de forma alterna.**
* **Ambas temperaturas serán fijas con valores de 38 y 35 grados respectivamente.**
* **Se enviarán al nodo servidor de forma alterna. Las veces pares, la temperatura interna, y las impares, la externa.**

En primer lugar, se ha modificado el código del ejemplo *examples/rpl-udp/udp-client.c* para que el nodo cliente ahora envíe la información de la temperatura tal y como se indica en el enunciado de la práctica. La parte del código modificada corresponde al bucle infinito del proceso principal y la encontramos en la siguiente captura.

Texto

Descripción generada automáticamente

Figura 14. Código cliente UDP para enviar temperatura en Celcius

* **Para el nodo “sumidero”, se puede utilizar el código fuente disponible en el mismo ejemplo ~/contiki-ng/examples/rpl-udp/udp-server.c (al añadir más funcionalidades no va a ser posible utilizar el servidor web visto anteriormente).**
* **Con ayuda de los ejemplos de en ~/contiki-ng/examples/rpl-udp/, modifique el código del nodo “sumidero” (udp-server) para que cuando reciba una medida de temperatura de un nodo terminal, la convierta a grados Fahrenheit y le responda con dicho valor al nodo. Para evitar cifras decimales, la fórmula de conversión será la siguiente: ºF=2\*ºC+32, donde ºF son grados Fahrenheit y ºC son grados Celsius.**

A continuación, se ha modificado el código del ejemplo *examples/rpl-udp/udp-server.c* para que el nodo servidor ahora responda al nodo cliente con la temperatura recibida del mismo convertida a grados Fahrenheit tal y como se indica en el enunciado de la práctica. La parte del código modificada corresponde la función de callback que se ejecuta siempre que se recibe un mensaje del cliente; y la encontramos en la siguiente captura.

Texto

Descripción generada automáticamente

Figura 15. Código servidor UDP para responder con temperatura en Fahrenheit

* **El nodo sumidero (servidor) será de tipo Sky Mote**
* **Los tres nodos cliente serán de tipo Z1.**
* **El intercambio de información deberá producirse (incluyendo los mismos mensajes de depuración) tal y como se muestra en la siguiente captura.**

Tras montar el escenario en COOJA y realizar una simulación, se obtiene el siguiente resultado. Podemos observar que cada uno de los nodos clientes (2, 3, 4) envía la temperatura en grados Celsius al nodo servidor (5); y que el nodo servidor les responde con el valor de la temperatura en grados Fahrenheit.

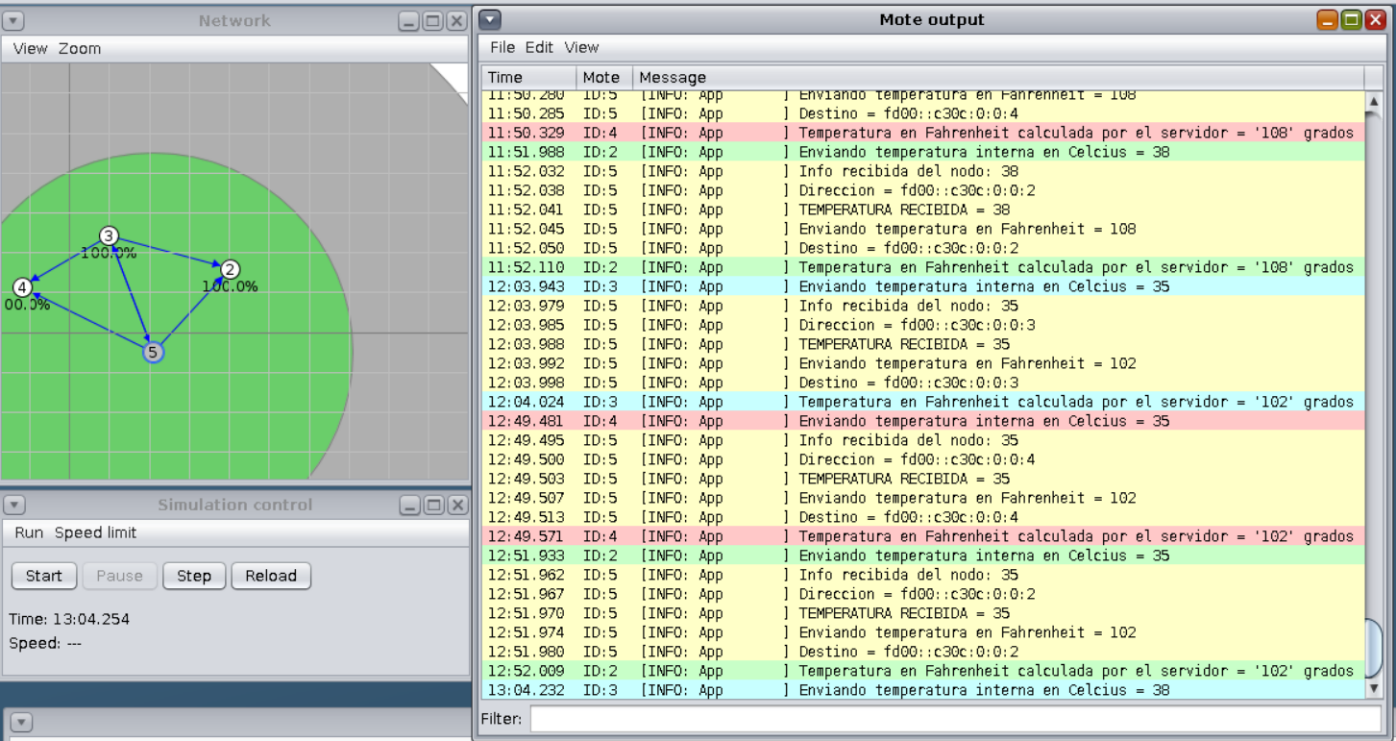
****

Figura 16. Resultado simulación COOJA práctica 2 - ejercicio 3

# Práctica 3

## Ejercicio 1

La configuración por defecto del PANID y del canal utilizado se encuentra en ***os/contiki-default-conf.h.***

**Texto

Descripción generada automáticamente**

Figura 17 Configuración por defecto del PANID y del canal

Para la realización del ejercicio, he ejecutado el servidor en mi dongle y el cliente en el dongle de un compañero. En las siguientes capturas se observa el intercambio de mensajes entre ambos nodos.

**Texto

Descripción generada automáticamente**

Figura 18 Servidor en ejecución en el dongle

**Texto

Descripción generada automáticamente**



Figura 19 Cliente en ejecución en el dondle

## Ejercicio 2

Se ha modificado la configuración por defecto en un fichero aparte de configuración para usar un PANID y un canal distinto al de mis compañeros. Este fichero se ha incluido en el código del cliente y del servidor para que utilicen la configuración indicada.

Texto

Descripción generada automáticamente

Figura 20. Inclusión de la configuración en cliente y servidor

Texto

Descripción generada automáticamente

Figura 21. Configuración PANID y canal

A continuación, se ha desplegado en COOJA una red con un servidor y 3 clientes. En la siguiente captura se puede observar la comunicación entre los 3 clientes con su servidor.

**Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media**

Figura 22. Simulación COOJA

## Ejercicio 3

Para desplegar la red en COOJA, se han configurado las redes diseñadas en los ejercicios 1 y 2 para que cada una de ellas tengan un PAN ID y un canal diferente. En concreto, se ha empleado el PAN ID “0xabb” y el canal 25 para la red del ejercicio 1; y el PAN ID “0xabac” y el canal 26 para la red del ejercicio 2. A continuación, se presentan capturas de la configuración mencionada.

Texto

Descripción generada automáticamente

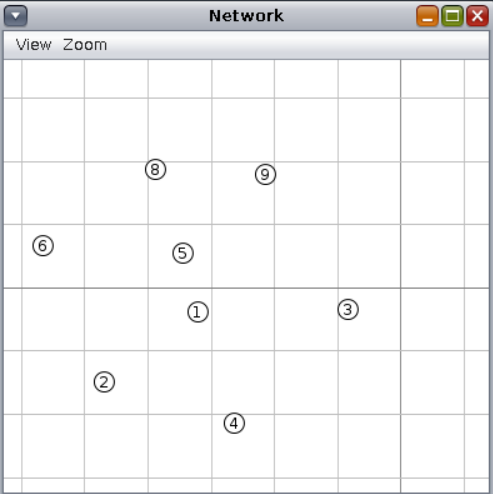
Figura 23. Configuración red ejercicio 1

Texto

Descripción generada automáticamente

Figura 24. Configuración red ejercicio 2

Tras compilar los correspondientes servidores (con TARGET sky) y clientes (con TARGET z1), se ha desplegado una red en COOJA con las redes de los ejercicios 1 y 2, cada una con 3 nodos clientes y 1 servidor.



Red ejercicio 1

Red ejercicio 2

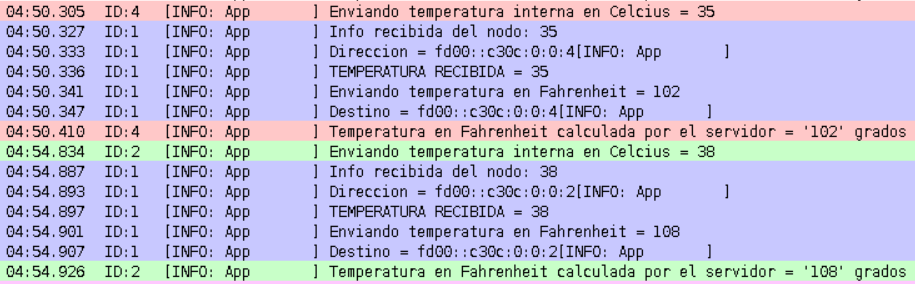
Servidor

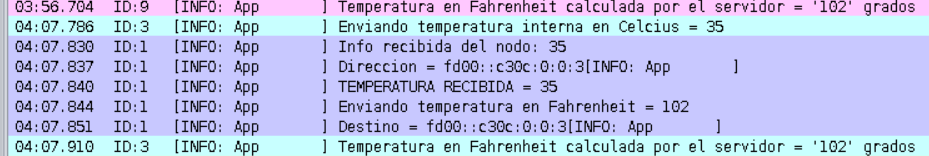
Servidor

Figura 25. Red COOJA ejercicio 3

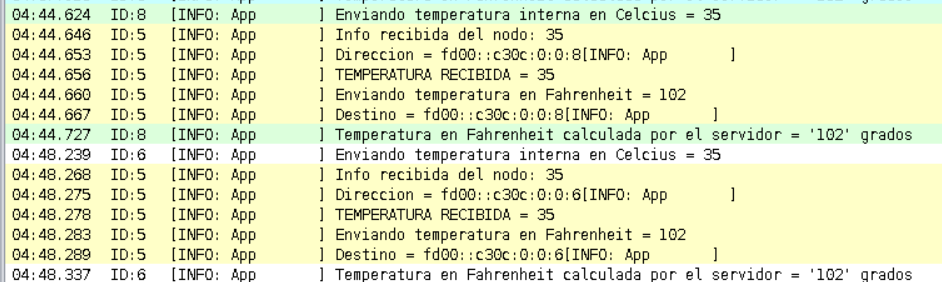
Se espera que los clientes de cada red solo se comuniquen con el servidor de esa red. Tras realizar la simulación, se comprueba que el comportamiento es el siguiente:

* Los nodos 2, 3 y 4 solo se comunican con el nodo 1 (comportamiento esperado):





* Los nodos 6, 8 y 9 solo se comunican con el nodo 5 (comportamiento esperado):



Texto

Descripción generada automáticamente

# Práctica 4