Aplicación de domótica en el contexto de IoT
A Domotic Application in the IoT context

Javier Emilio Luis Longo Imedio

Máster en Ingeniería Informática, Facultad de Informática,
Universidad Complutense de Madrid

Trabajo Fin Máster en Ingeniería Informática

Directores:

Alberto A. Del Barrio - Guillermo Botella Juan

Convocatoria de enero de 2019

Calificación: 6,5

Autorización de Difusión

Javier Emilio Luis Longo Imedio

8/Enero/2018

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Comunicación inalámbrica entre sistemas empotrados distribuidos heterogéneos”, realizado durante el curso académico 2017-2018 bajo la dirección de Alberto A. Del Barrio y Guillermo Botella Juan en el Departamento de Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Hoy en día es habitual encontrar dispositivos IoT en fábricas, centros de logística, en las ciudades e incluso en los hogares. Sin embargo, el despliegue de cientos de millones de dispositivos acarrea consigo un incremento en el consumo energético. Por ello, ha surgido un gran interés en cómo disminuir este consumo ya sea mediante protocolos ligeros o conectando al IoT dispositivos de ultra bajo consumo energético. Para ello, por un lado, se han desarrollado protocolos ligeros cómo CoAP (Constrained Application Protocol) siguiendo el modelo cliente-servidor de HTTP o MQTT (Message Queue Telemetry Transport) basado en el modelo *publish-subscribe*, que es una alternativa al modelo cliente-servidor. Por otro lado, se han creado chips de red que contienen la pila TCP/IP lo cual habilita a microcontroladores de muy bajo consumo energético con limitaciones hardware importantes conectarse al IoT. En este trabajo se estudia el cómo conectar dispositivos de bajo consumo energético al IoT a través del desarrollo de una aplicación de domótica para medir la temperatura mediante sensores. El sistema utiliza chips de red para conectar dispositivos de ultra bajo consumo energético al IoT y los protocolos HTTP, UDP y MQTT para el intercambio de datos. Se han realizado pruebas para conocer el tiempo de transmisión para datagramas UDP entre dispositivos, el índice de pérdida de datos y la calidad del enlace inalámbrico. Los resultados obtenidos muestran que el sistema desarrollado tiene la capacidad de procesar 18 muestras de temperatura por segundo obtenidas de tres sensores distintos con un índice de recepción de datos del 99% y con un tiempo de transmisión medio de 88 ms por datagrama.

Palabras clave

IoT, MQTT, MSP430, Tiva C Series Launchpad, CC3100

Abstract

Nowadays is quite usual to find IoT devices in factories, logistics floors, in cities and even at home. However, the deployment of millions of devices leads to an increase in energy consumption. Therefore, there has been a great interest in how to reduce this consumption either by means of lightweight protocols or by connecting directly to the IoT ultra-low energy consumption devices. For this, on the one hand, lightweight protocols have been developed such as CoAP (Constrained Application Protocol) following the client-server model of HTTP or MQTT (Message Queue Telemetry Transport) based on the publish-subscribe model which is an alternative to the client-server model. On the other hand, network chips have been created that contain the TCP/IP stack which enables ultra-low energy consumption microcontrollers with important hardware limitations to connect to the IoT. This thesis studies how to connect low energy consumption devices to the IoT through the development of a home automation application that measures temperature using sensors. The system uses network chips to connect ultra-low power consumption devices to the IoT and the HTTP, UDP and MQTT protocols for data exchange. Experiments were conducted to know the transmission time for UDP datagrams between devices, the data loss rate and the quality of the wireless link. The results obtained show that the developed system has the capacity to process 18 temperature samples per second obtained from three different sensors with a data reception rate of 99% and an average transmission time of 88 ms per datagram.

Keywords

IoT, MQTT, MSP430, Tiva C Series Launchpad, CC3100

Agradecimientos

Quiero agradecer a mis directores Alberto A. Del Barrio y Guillermo Botella Juan por su ayuda durante la realización de este proyecto.

Dedicatoria

A mis padres,

Emilio Longo y Adelina Imedio

por su apoyo incondicional.

Las verdades de todas las enseñanzas del género humano son similares
y se condensan en una sola:
encontrar el camino a lo que sientes cuando quieres tiernamente,
o cuando creas,
o cuando construyes tu casa,
o cuando das a luz a tus hijos,
o cuando miras a las estrellas por la noche.

-Wilhelm Reich-

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Código

Listado de acrónimos y abreviaturas

ACK	acknowledgement
ACTSS	active sample sequencer
ADC	analog to digital converter
ADCEMUX	ADC event multiplexer select
ADCISC	ADC interrupt status and clear
ADCPSSI	ADC processor sample sequence initiate
ADCPSSI	ADC Processor Sample Sequence Initiate
ADCSSFIFO	ADC Sample Sequence Result FIFO
ADCSSMUX	ADC sample sequence input multiplexer select
ALU	arithmetic/logic unit
AMSEL	analog mode select
AP	access point
ARM	Acorn RISC Machines
ARP	address resolution protocol
B	byte
BSON	binary object notation
BSSID	basic service set ID
CAN	controller area network
CoAP	constrained application protocol
CPU	central processing unit
CR	carriage return
CRC	cyclic redundancy check
dbm	decibelios milivatios
DCO	digitally controlled oscilator
DEN	digital enable
DH	Diffie–Hellman
DHCP	dynamic host configuration protocol
DNS	domain name system
DRA	direct register access
DSSS	direct sequence spead spectrum
EEPROM	electrically erasable programmable ROM
FCS	frame check sequence
FHSS	frequency hopping spread spectrum
FIFO	first in first out
FLL	frequency locked loop
GPIO	general purpose input output
GPIOAFSEL	GPIO alternate function select
HTTP	hypertext transfer protocol
I^2C	inter integrated circuit
ICDI	in-circuit debug interface
ID	identification
IDE	integrated development environment
IIS	internet information services
IOT	internet of things
JSON	Javascript object notation
JTAG	join test action group
LF	line feed
LFIOSC	low frequency internal oscillator
M2M	machine to machine

MAC	media access control
MCU	microcontroller
MIME	multipurpose internet mail extension
MOM	message oriented middleware
MOSC	main oscillator
MQTT	message queue telemetry transport
NoSQL	not only SQL
OFDM	orthogonal frequency divisional multiplexing
PC	personal computer
PIOSC	precision internal oscillator
PLC	programmable logic controllers
PLL	phase lock loop
PRR	packet reception ratio
PWM	pulse width modulator
PWM	pulse width modulator
QoS	quality of service
RAM	random access memory
RCGCADC	analog-to-digital converter run mode clock gating control
RCGCGPIO	general-purpose input/output run mode clock gating control
REFO	internal trimmed low frequency reference oscillator
RFC	request for comments
RFID	radio frequency identification
RISC	reduced instruction set computer
ROM	read only memory
RSSI	received signal strength indicator
RTCOSC	hibernate RTC oscillator
RTOS	real time operating system
SoC	system on chip
SPI	serial peripheral interface bus
SQL	structured query language
SSI	synchronous serial interface
SSID	service set id
STA	station
TBTT	target beacon transmit time
TCP	transmission control protocol
TI	Texas Instruments
TLS	transport layer security
UART	universal asynchronous receivers/transmitter
UDP	unrealiable datagram protocol
URI	uniform resource identifier
URL	universal resource locator
WAP	Wi-Fi protected access
WEP	wired equivalent privacy
Wi-Fi	wireless fidelity

1 Introducción

Hoy en día es cada vez más habitual que microcontroladores se conecten a la red para publicar y consumir información. Por ello, la red ha pasado de ser un ecosistema donde los humanos publican y consumen información a ser una red más amplia que incluye a cientos de millones de pequeñas máquinas que generan y consumen información. Este nuevo ecosistema se conoce como IoT (Internet of Things). IoT es un término acuñado por Kevin Ashton en 1997 para referirse a las etiquetas electrónicas RFID (Radio Frequency Identification) utilizadas en la gestión de cadenas de suministros. Estas etiquetas fueron patentadas por Mario W. Cardullo en 1973 y permiten identificar objetos sin que estén en el campo de visión del lector y sin intervención humana. Además, permiten al lector enviar datos sobre el objeto como la identidad del mismo y su posición a través de Internet en tiempo real.

La idea de comunicación autónoma entre máquinas es anterior a IoT y se conoce como M2M (Machine to Machine). Este concepto implica que un dispositivo autónomo sea capaz de comunicarse con otro dispositivo autónomo sin necesidad de intervención humana. Por tanto, M2M es el precursor de IoT y son términos estrechamente relacionados. Sin embargo, una de las principales diferencias entre M2M e IoT es que un dispositivo M2M pueden comunicarse mediante un canal no IP como un puerto serie sin tener que acceder a Internet mientras que un dispositivo IoT siempre accederá a Internet [1].

Para el año 2021 se calcula que habrá en torno a 33 mil millones de dispositivos IoT y para el año 2035 se augura que el número superará los cien mil millones de dispositivos [1]. Los principales sectores de las sociedades modernas pueden beneficiarse de IoT. Por ejemplo, se puede ganar en eficiencia con dispositivos que monitoricen de forma exhaustiva el proceso de fabricación en las fábricas. Además, IoT puede aportar mayor seguridad detectando posibles escapes de gases o cualquier otro fenómeno físico que ponga en peligro el buen funcionamiento de la maquinaría y la seguridad.

Por otro lado, son cada vez más los consumidores que están utilizando IoT especialmente en el apartado de domótica y casas inteligentes ya que estas aplicaciones permiten no sólo ahorrar energía, sino también aportan confort y bienestar. Los establecimientos comerciales también están utilizando dispositivos IoT para reducir gastos mediante un seguimiento de inventario detectando pérdidas y proporcionando datos que pueden ser procesados para optimizar la cadena de suministro.

Actualmente existen alrededor de 500 millones [1] del número de dispositivos *wearable* utilizados para la monitorización de signos vitales. Por ello, el sector de la salud va a ser uno de los grandes beneficiados de IOT. Un dispositivo puede monitorizar a un paciente que se encuentre en casa o detectar algún padecimiento tan pronto de produzcan los primeros síntomas lo cual puede permitir iniciar de inmediato un tratamiento con los medicamentos apropiados. Dentro de los hospitales, IOT puede utilizarse para controlar el equipo médico y otro material hospitalario.

Las ciudades van a ser las zonas donde cada vez habrá más dispositivos IOT. Por ejemplo, en la ciudad de Barcelona [1] ya tienen implementado un sistema para optimizar la recogida de basura basado en la capacidad de los contenedores y el tiempo transcurrido desde la última vez que fueron vaciados. Además, las ciudades pueden utilizar dispositivos IOT para monitorizar los niveles de contaminación, mejorar el tráfico mediante

semáforos inteligentes y ahorrar energía en el alumbrado público activándolo según sea necesario. Fenómenos como una fuerte nevada pueden ser atendidos de manera eficiente con dispositivos IOT que indiquen cuales han sido las zonas más afectadas. IOT puede utilizarse para aumentar la seguridad de las calles mediante cámaras y alertas en caso de actos delictivos. También se pueden evitar accidentes monitorizando el estado las principales infraestructuras de la ciudad para que en caso de ser necesario puedan ser reparadas o reemplazadas. Los dispositivos IOT pueden además utilizarse para la detección de amenazas terroristas abortando posibles ataques. Sin embargo, un despliegue masivo de dispositivos IoT acarrea un incremento en el consumo energético muy importante por lo es recomendable utilizar dispositivos que aporten la mayor eficiencia energética posible. Existen microcontroladores con modos de muy bajo consumo energético pero la principal desventaja es que no soportan la pila TCP/IP debido a que tienen grandes limitaciones en cuanto a recursos hardware.

1.1 Objetivos

En este trabajo, a través de una aplicación de domótica para medir la temperatura en tiempo real, se plantea una estrategia que combina hardware y software que permita el acceso a Internet de microcontroladores de muy bajo consumo energético. Se estudia el cómo incorporar los protocolos UDP, HTTP y MQTT [2] a los dispositivos y la utilización de un RTOS (Real Time Operating System) para lograr mayor eficiencia en el uso de recursos hardware. Por tanto, entre los objetivos principales se encuentran el estudiar en profundidad el protocolo MQTT e implementar un cliente MQTT y un servidor UDP utilizando FreeRTOS en un dispositivo de muy bajo consumo energético. También, incluir dispositivos con mayor capacidad hardware que utilicen los protocolos UDP [3] [4] y HTTP [5] para enviar los datos de temperatura obtenidos a través de sensores junto chips de red de bajo consumo energético que permitan a todos los dispositivos utilizar los protocolos TCP/IP [3] y los estándares 802.11. Como objetivos secundarios se encuentran el implementar un servidor web utilizando Node.js [2] que junto a una base de datos permita guardar la información y mostrarla mediante un gráfico en tiempo real a través de un navegador web. Además, realizar experimentos para obtener el tiempo de transmisión de los datagramas UDP, cuantificar la pérdida de datagramas y analizar el estado del enlace inalámbrico.

2 Estado del arte

El internet de las cosas implica conectar a la red microcontroladores que se encuentran empotrados en sistemas como coches, trenes o electrodomésticos. Estos sistemas pueden combinar microcontroladores de bajo rendimiento con procesadores de alto rendimiento como los SoC (System on Chip). No obstante, las principales cualidades que se busca en un microcontrolador para un sistema empotrado son bajo coste y poco consumo energético.

Por lo general los dispositivos IOT utilizan protocolos de comunicación que van sobre TCP/IP. Existen varios protocolos de comunicación que pueden utilizarse para IOT donde algunos de los más utilizados son HTTP, CoAP [7] y MQTT. Por otro lado, en el apartado de hardware de un sistema IOT existen multitud de fabricantes de microcontroladores como Atmel, Microchip, NXP y TI (Texas Instruments). También están las placas de prototipado como Arduino in SoC como Raspberry Pi [4]. Las plataformas de Arduino [5] se construyen alrededor de un microcontrolador Atmel e incluyen pines de entrada y salida GPIO (General Purpose Input Output), entradas ADC (Analog to Digital Converter) que permiten conectarle sensores, motores y luces además de otros periféricos como UARTs (Universal Asynchronous Receivers/Transmitter), SPI (Serial Peripheral Interface Bus), buses CAN (Controller Area Network) o I^2C (Inter Integrated Circuit). También existen otras plataformas como la Tiva C Series Launchpad o MSP430 de TI que disponen de entornos de desarrollo con herramientas de depuración sofisticadas como Code Composer Studio basado en el IDE Eclipse. Usualmente a este conjunto de plataformas se le añaden sensores [10] que permiten medir algún fenómeno físico como puede ser la temperatura. Existen todo tipo de sensores cuya elección debe tener en cuenta no sólo el precio sino también la precisión y la documentación disponible para realizar la implementación del código que permita acoplar el sensor al sistema.

Para conectar a Internet los dispositivos IOT se utilizan chips de red como el ESP8266 [6] o el CC3100 de TI. Estos chips de red incluyen la pila TCP/IP por lo que pueden utilizarse en dispositivos con poca memoria RAM y además permiten conectarse a una red Wi-Fi. Las principales diferencias entre distintos chips de red son por un lado el precio y por otro la documentación disponible, la API asociada, las herramientas avanzada como pueden ser funciones para conocer el estado del enlace inalámbrico y los modos disponibles de ahorro de energía entre otros.

A nivel software, los microcontroladores se pueden programar con un RTOS (Real Time Operating System) como FreeRTOS o sin utilizar un RTOS en cuyo caso se habla de una implementación bare metal. Existen multitud de lenguajes para programar microcontroladores como C [12] seguido de sus variantes como nesC, Keil C, Dynamic C y B#.

Por otro lado, los dispositivos IOT generan gran cantidad de volumen de datos por lo que para almacenarlos se necesita una base de datos que sea escalable como puede ser MongoDB [7]. Para almacenar los datos, por lo general el dispositivo de conecta a un servidor web [12] realizando peticiones POST HTTP [12]. Existen multitud de servidores web entre los que se encuentran Apache, IIS, Nginx, Oracle y también la plataforma de desarrollo Node.js [7]. Esta última permite implementar un servidor web no bloqueante utilizando

funciones callback lo cual facilita la gestión de gran volumen de peticiones sin comprometer el rendimiento. Por tanto, una aplicación IOT abarca tanto el hardware de bajo nivel y protocolos de comunicación como también software de alto nivel que permita almacenar y gestionar los datos generados.

2.1 MQTT

Existen principalmente dos grupos de protocolos de comunicación para IOT. Por un lado, están los protocolos de petición y respuesta donde uno de los más utilizados es HTTP y por otro lado están los protocolos de publicación y subscripción donde el más utilizado es MQTT que además está diseñado para el ámbito IOT.

MQTT es un protocolo pensado para minimizar el consumo de ancho de banda y energía [1]. Al igual que HTTP, funciona sobre TCP. Sin embargo, mientras que en HTTP existe un servidor que atiende las peticiones de los clientes en MQTT existen los publishers que publican mensajes y los subscribers que los consumen a través de un bróker. Un cliente MQTT puede ser publisher y subscriber a la vez. Los mensajes se publican bajo un tema que debe conocer con antelación para que el subscriber pueda recibir los mensajes del publisher. Tanto el publisher como el subsciber se conectan a un bróker (Fig. 2-1) que gestiona los mensajes y los temas.

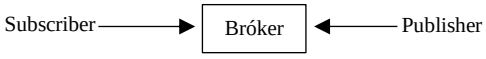

Figura 2-1 Modelo del protocolo MQTT. Fuente: elaboración propia.

MQTT no especifica un formato sobre los datos contenidos en los mensajes, aunque por lo general se utiliza JSON (Javascript Object Notation). Si un cliente se desconecta repentinamente de un bróker, existe el "last will" que es un mensaje normal MQTT que permite al cliente recuperar el último mensaje que recibió bajo ese tema junto a otros parámetros de configuración como el QoS (Quality of Service) de los temas al que estaba subscrito el antes de que se perdiese la conexión.

En MQTT existen tres tipos de QoS, QoS-0 que funciona como UDP ya que el publisher envía el mensaje, pero no recibe una confirmación de que el subscriber lo ha recibo. El QoS-1 permite al publisher saber que el mensaje se ha recibido al menos una vez por el subscriber. El funcionamiento de OoS-1 consiste en que el subscriber al recibir el mensaje, envía otro mensaje *PUBACK* (Fig. 2-2) de tipo *ACK* al servidor indicando que el mensaje se ha recibido correctamente.

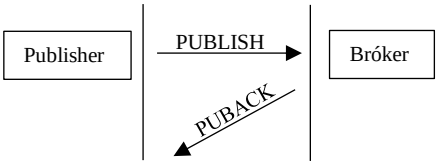

Figura 2-2 Secuencia de intercambio de mensajes durante una publicación de mensaje con QoS-1 entre el publisher y el bróker MQTT. Fuente: elaboración propia.

El QoS-3 (Fig. 2-3) avisa tanto al publisher como al subscriber que el mensaje se ha recibido correctamente. El subscriber del mensaje primeramente envía un mensaje de tipo *PUBREC* indicando que la publicación ha sido recibida. El publisher recibe este mensaje y a continuación responde como un mensaje *PUBREL* que permite al subscriber descartar cualquier retrasmisión del mensaje. La recepción del mensaje *PUBREL* se confirma por parte del subscriber con un mensaje *PUBCOMP*. Hasta que no se envía el mensaje *PUBCOMP* al publisher, el subscriber mantiene el mensaje original guardado por seguridad.

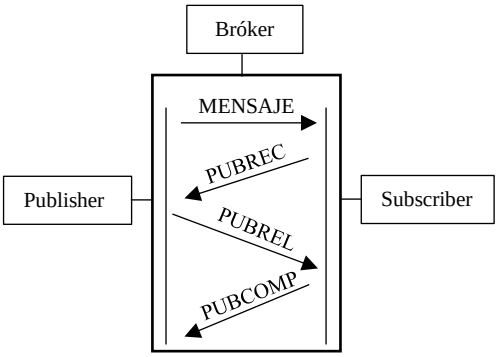

Figura 2-3 Intercambio de mensajes entre el emisor y receptor para QoS-3. Fuente: elaboración propia.

Los paquetes MQTT tienen una cabecera que consta de un mínimo de 2 bytes que son obligatorios y extensiones a esta cabecera que son opcionales seguidas del payload que también es opcional. La comunicación entre un cliente y un bróker siempre la inicia el cliente con un mensaje *CONNECT* (Fig. 2-4).

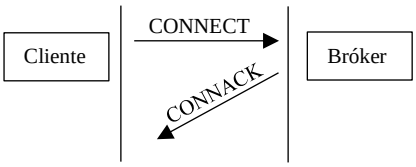

Figura 2-4 Secuencia de conexión entre un cliente y un bróker MQTT. Fuente: elaboración propia.

El mensaje debe contener un identificador único que permita al bróker identificar al cliente y este campo es obligatorio. El resto de la información contenida en el mensaje *CONNECT* es opcional. Esta información opcional incluye la posibilidad de indicarle a bróker que guarde el estado de la sesión o que se descarte la última sesión e inicie una nueva. También se puede indicar un nombre de usuario y contraseña. Además, se puede configurar el "last will message" en caso de desconexión inesperada. También existe la opción de utilizar un keepalive para que el bróker sepa que la conexión sigue siendo válida ya que muchas veces los dispositivos IOT permanecen sin enviar mensajes durante largos periodos de tiempo.

Una vez el bróker recibe el mensaje *CONNECT*, responde con un mensaje *CONNACK* y un código. Si la conexión ha tenido éxito el bróker devuelve un 0. Si la conexión ha sido rechazada porque la versión MQTT del cliente no es válida el bróker devuelve un 1. Si el identificador del cliente no es aceptado el bróker devuelve un 2. Si el bróker no está disponible se devuelve un 3. En caso de que el nombre de usuario o la contraseña no sean correctos se devuelve un 4. Si el cliente no está autorizado a conectarse al bróker se devuelve un 5.

Si el cliente ha indicado en su mensaje *CONNECT* que desea un keepalive entonces tanto el bróker como el cliente intercambian mensajes ping mediante el mensaje *PINGRESP* y el mensaje *PINGREQ* respectivamente (Fig. 2-5). Tanto el envío de un *PINGREQ* antes de que expire el intervalo indicado en el campo keepalive del mensaje *CONNECT* como un mensaje *PUBLISH* por parte del cliente mantendrá la comunicación abierta.

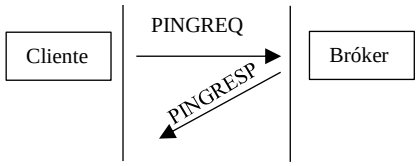

Figura 2-5 Secuencia de intercambio de mensajes durante un keepalive entre el bróker MQTT y el cliente. Fuente: elaboración propia.

Una vez abierta la conexión, el cliente puede publicar mensajes mediante un mensaje *PUBLISH*. El mensaje *PUBLISH* contiene cuatro campos que son obligatorios. El primero es un identificador del mensaje que en caso de utilizar QoS-0 se inicializa a 0. El siguiente campo es el nombre del tema bajo el cual se va a publicar el mensaje. Después le sigue el campo donde se indica si se utiliza QoS-0, QoS-1 o QoS-2. El siguiente campo indica el nombre del bróker para propósitos de autenticación. A continuación, está el payload en cualquier formato ya que MQTT no especifica el tipo de formato. Sin embargo, tanto el publisher como el subscriber tienen que estar de acuerdo con el tipo de formato utilizado, aunque es muy habitual que se utilice JSON. Si un cliente desea subscribirse a un tema envía un mensaje de petición de tipo *SUBSCRIBE* que el bróker confirma con un mensaje *SUBACK* (Fig. 2-6).

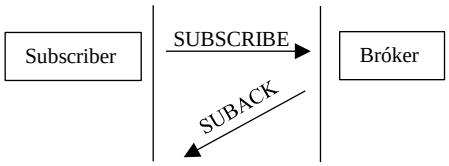

Figura 2-6 Secuencia de intercambio de mensajes durante una subscripción entre el publisher y el bróker MQTT. Fuente: elaboración propia.

El mensaje *SUBSCRIBE* tiene como primer campo el identificador del mensaje. A continuación, se encuentra el tema al cual se quiere subscribir seguidamente del campo que contiene el QoS de los mensajes que se publiquen bajo ese tema. En caso de querer subscribirse a varios temas el campo dos y tres se repiten para cada uno de los temas adicionales. El subscriber puede darse de baja de un tema enviando una petición *UNSUBSCRIBE* que el bróker confirmará con *UNSUBACK*.

2.2 Clasificación de microcontroladores

En el RFC 7228 *Terminology for Constrained-Node Networks* se concretan las limitaciones hardware de los microcontroladores y se clasifican en tres clases. La clase 0 tienen menos 10 KB de RAM [11] y aproximadamente 100 KB de Flash. Este tipo de dispositivos no soportan la pila TCP/IP por lo que para enviar datos a la red necesitan o bien enviarlos a un nodo base más potente o utilizar un chip de red que contenga la

pila TCP/IP. Los microcontroladores de clase 1 tienen aproximadamente 10 KB de RAM y 100 KB de Flash con unas restricciones de memoria que les impenden soportar en su totalidad los protocolos de red habituales por lo que hacen uso de protocolos de aplicación que ocupan poca memoria como CoAP que es una versión minimalista de HTTP y utiliza sockets UDP en lugar de TCP [11]. Los microcontroladores de clase 2 con 50 KB de RAM y 250 KB de Flash soportan protocolos de aplicación habituales como HTTP y TLS [11].

2.3 Tiva C Series Launchpad TM4C123GH6PM

Para el desarrollo del sistema de domótica se va a utilizar la placa Tiva C Series Launchpad de TI como dispositivo para la captura y emisión de los datos del sensor de temperatura. Estos datos se enviarán hacia un servidor web utilizando HTTP y también a una plataforma MSP mediante datagramas UDP. La placa Tiva C Series Launchpad de 32- bit [13] funciona como máximo a 80 MHz y tiene un procesador ARM (Acorn RISC Machines) Cortex M4F [12] [13]. Se considera de bajo consumo energético con 370 µA/MHz, dispone de 32KB de memoria RAM para datos temporales, 256 KB de memoria Flash para instrucciones del programa y 2KB de EEPROM (Electrically Erasable Programmable ROM) para tablas y constantes. Para generar la señal cuadrada del reloj el microcontrolador dispone de cuatro osciladores. El primero es un PIOSC (Precision Internal Oscillator) u oscilador interno de precisión de 16 MHz $\pm$ 3% [12]. Este oscilador es menos preciso que el cristal de cuarzo sin embargo requiere menos potencia y no necesita un cristal externo. El segundo oscilador MOSC (Main Oscillator) es el principal y puede conectarse a un generador de señal externo o a un cristal de cuarzo y permite controlar la velocidad de ejecución del dispositivo mediante software utilizando PLL (Phase Lock Loop) [12]. El tercer oscilador LFIOSC (Low Frequency Internal Oscillator) de baja frecuencia está pensado para utilizarse en modo de bajo consumo energético [12]. El cuarto oscilador es RTCOSC (Hibernate RTC Oscillator) de tiempo real con un cristal de cuarzo de 32.768 MHz está pensado para utilizarse con el módulo de hibernación. Además, la plataforma tiene ocho UARTs, cuatro módulos SSI (Synchronous Serial Interface), cuatro módulos I^2C y dos módulos de bus CAN. Incluye dos módulos PWM (Pulse Width Modulator) con dieciséis salidas para modular el ancho de pulso y controlar por ejemplo un servo motor o convertir un valor digital a analógico. También se incluye dos módulos ADC con un límite de un millón de muestras por segundo, hasta 12 timers de 16-bit o de 32-bit y seis bloques de GPIO con 43 líneas de entrada y salida [13]. La plataforma se puede programar utilizando un IDE llamado Energia que es similar a Arduino, mediante el lenguaje C o utilizando el lenguaje ensamblador [14]. Además, la plataforma Tiva C Series Launchpad permite depurar a medida que se va ejecutando el código gracias a un emulador integrado. Este emulador se conoce como el ICDI (In-Circuit Debug Interface) que habilita la depuración utilizando el estándar IEEE 1149.1 conocido como el protocolo JTAG (Join Test Action Group). Por tanto, la placa de evaluación Tiva C series Launchpad tiene dos microcontroladores TM4C123GH6PM donde uno de ellos es el controlador/adptador de la interfaz ICDI. Para acceder a esta interfaz se utiliza el driver conocido como Stellaris ICDI driver. Este conjunto de interfaces y protocolos son los que permiten depurar el código a medida que se va ejecutando.

2.4 MSP430F5529LP

La plataforma MSP430 [20] de TI se va a utilizar como dispositivo receptor de los datagramas enviados por la plataforma Tiva. En caso de que los datos recibidos indiquen que la temperatura ha superado un umbral, esta plataforma enviará los datos a un bróker MQTT convirtiéndose en un publisher MQTT. La plataforma lleva un

microcontrolador de tipo RISC de 16-bit que está pensado para dispositivos que funcionen con pilas durante largos periodos de tiempo ya que tiene un consumo energético de 150 µA/MHz. Puede alcanzar los 25 MHz, tiene una memoria RAM de 8 KB, una memoria Flash de 128 KB e interfaces serie SPI, I^2C y UART y un ADC de 12-bit. Esta plataforma dispone de un reloj rápido MCLK que proporciona la señal cuadrada a la CPU y los periféricos. Este MCLK tiene un oscilador controlado digitalmente o DCO (Digitally Controlled Oscilator) capaz de iniciarse en 35 microsegundos lo cual es más rápido que un oscilador de cristal. Para mantener la frecuencia estable del DCO se utiliza el módulo FLL (Frequency Locked Loop) [20].

Los relojes lentos consumen menos energía por lo que desactivar el reloj rápido y activar el reloj lento permite ahorrar energía. En este caso, el microcontrolador dispone tres relojes lentos para mantener funcionando los timers y otros periféricos cuando el dispositivo se encuentra en modo de ahorro energético. El primero es el reloj REFO (Internal Trimmed Low Frequency Reference Oscillator) que no utiliza un cristal por lo cual es la opción para sistemas que no quieren o no desean incluir un cristal y así ahorrar dinero. El reloj REFO se encuentra en el chip y opera a una frecuencia de 32 kHz [20]. El segundo es el reloj LFXT1 que tiene un oscilador de cristal, pero consume menos energía que el reloj REFO. El tercero es el reloj VLO que es el que menos energía consume, opera entre 12 kHz y 20 kHz y se recomienda para aplicaciones que no requieran mucha precisión.

2.5 Sensor de temperatura LM34

Se ha seleccionado el sensor de temperatura LM34 [21] de TI cuyo voltaje tiene una proporcionalidad lineal con grados Fahrenheit. El sensor no necesita ser calibrado y tiene una precisión de $\pm 1/2$ ºF con un rango de funcionamiento de -50 ºF a 300 ºF lo que equivale a -45 ºC a 148 ºC.

2.6 Wi-Fi

Usualmente cuando se habla de Wi-Fi se utiliza como otro nombre para la familia de estándares que emanan del estándar 802.11 del año 1997 para redes inalámbricas [1]. Este estándar fue el resultado de un protocolo creado por la empresa NCR en 1991 [1] para utilizar en una red de cajas registradoras. Fue en el año 1999 fue cuando se creó la Wi-Fi Alliance para certificar productos inalámbricos. El estándar 802.11 proporciona una velocidad de transmisión de hasta 2 Mbps en el rango de frecuencia de 2.4 GHz. Utiliza las técnicas de reducción de interferencias conocidas como formatos de unidifusión inalámbrica de tipo FHSS (Frequency Hopping Spread Spectrum) o espectro ensanchado por salto de frecuencia y DSSS (Direct Sequence Spread Spectrum) o espectro ensanchado por secuencia directa. El estándar 802.11a alcanza unas velocidades de hasta 54 Mbps en el rango de 5 GHz. Utiliza el formato de unidifusión inalámbrica OFDM (Orthogonal Frequency Divisional Multiplexing) o multiplexado por división de frecuencias ortogonal. El estándar 802.11b tiene tasas de transmisión entre 5 Mbps y 11 Mbps en la frecuencia de 2.4 GHz y el formato de unidifusión inalámbrica HR/DSSS o multiplexado por secuencia directa de alta tasa. El estándar 802.11g tiene una tasa de transmisión de hasta 54 Mbps en el rango de 2.4 GHz y utiliza formato de unidifusión inalámbrica OFDM. El estándar 802.11n tiene una tasa de transmisión de hasta 450 Mbps en la frecuencia de 2.4 GHz y utiliza formato de unidifusión inalámbrica OFDM.

El handshake de una red Wi-Fi se inicia por el dispositivo con el chip de red conocido como la STA (Station). Este dispositivo se encuentra en estado de escucha en un canal concreto. Primeramente, la STA realiza lo que se conoce como la "scanning phase" o fase de búsqueda y consiste en buscar un AP (Access Point) al que conectarse. Se puede realizar de forma pasiva o de forma activa. En la forma pasiva la STA recibe un mensaje del AP conocido como beacon que utiliza la STA para unirse a la red (Fig. 2-7).

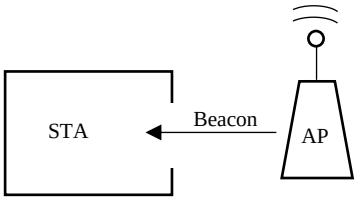

Figura 2-7 Fase de búsqueda en modo pasivo. Fuente: elaboración propia.

Durante la búsqueda activa es la STA la que envía un mensaje conocido como probe request. Durante la búsqueda activa el STA utiliza más energía, pero este modo permite conectarse a la red más rápido. Si el AP recibe el probe request (Fig. 2-8) contesta con un probe response que es similar a un beacon.

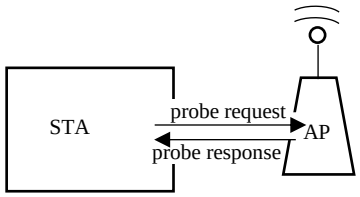

Figura 2-8 Fase de búsqueda en modo activo. Fuente: elaboración propia.

El beacon o probe response de AP contiene la información que necesita el STA para unirse a la red. El mensaje contiene un primer campo con el SSID (Service Set Id) o el nombre de la red de una longitud de 1 a 32 caracteres, aunque se puede esconder inicializando este campo a 0. El siguiente campo es el BSSID (Basic Service Set ID) que incluye un identificador único al estilo MAC (Media Access Control) de 48-bit. El siguiente campo contiene el ancho del canal que puede ser de *x* MHz. A continuación, está el campo con los canales válidos para el país donde se encuentran el AP y la STA. Le sigue el campo con TBTT (Target Beacon Transmit Time) que indica cada cuanto tiempo se envía un beacon por parte del AP. Le sigue el campo configuraciones para gestionar el consumo mediante intervalos de apagado y encendido. El último campo contiene los servicios de seguridad como WEP (Wired Equivalent Privacy), WPA (Wi-Fi Protected Access) y WPA2. Si la STA encuentra un AP entra en la fase de autenticación. Una vez superada la fase de autenticación se pasa a la fase de asociación. Durante esta fase la STA envía al AP una petición de asociación o association request. El AP contestará con un association response (Fig. 2-9) que permite a la STA unirse a la red o la excluye.

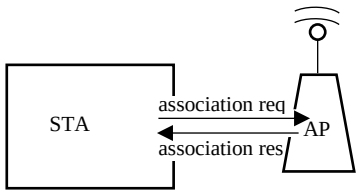

Figura 2-9 Fase de asociación. Fuente: elaboración propia.

Si el AP incluye a la STA le enviará un identificador de asociación y añadirá la STA a la lista de clientes conectados.

2.7 CC3100

Para ambas plataformas se ha seleccionado el chip de red CC3100 de TI que permite conectar microcontroladores de 8-bit, 16-bit y 32-bit a Internet mediante una red inalámbrica Wi-Fi. El chip contiene una antena y un microcontrolador ARM con la pila TCP/IP, la pila TLS/SSL y otros protocolos de Internet [22] [23]. Soporta los estándares 802.11b y 802.11g de 2.4 GHz y también 802.11n de 5 GHz. Se pueden utilizar hasta ocho sockets BSD TCP o UDP de manera simultánea y dos sockets TLS y SSL. También se puede inducir a un modo hibernación con un consumo de 4µA de corriente. Se conecta al microcontrolador host a través de SPI o UART. Lleva un microcontrolador ARM con un cristal de 32 kHz y un cristal de 40 MHz con un oscilador interno. Además, tiene una memoria Micron M25P80 de 8 Mbits. Para la técnica de modulación DSSS que permite reducir los niveles de interferencia, el poder de transmisión de la antena son 18 dBm a 1 DSSS y de recepción de -95 dBm. Para transmisión por OFDM es de 14.5 dBm a 54 OFDM y la sensibilidad de recepción es de –74.0 dBm a 54 OFDM.

El chip de red CC3100 incluye una API conocida como SimpleLink Host Driver para el desarrollo de aplicaciones y está dividida en 6 módulos [22] (Tab. 2-1).

Módulo	Número de funciones
Device	8
Wlan	19
Socket	19
Netapp	10
Netcfg	2
FileSystem	7

Tabla 2 -1 Módulos que conforman la API SimpleLink. Fuente: elaboración propia.

El módulo Device aporta 8 funciones para controlar el hardware del dispositivo tales como iniciarlo, paralo y obtener el estado del mismo. El módulo WLAN engloba todo lo relacionado con la comunicación inalámbrica de radio Wi-Fi mediante 19 funciones. Permite gestionar los profiles utilizados para conectarse al AP mediante el SSID, el tipo de seguridad y la clave. Este módulo también permite gestionar los Rxfilters. Estos filtros permiten seleccionar los frames que serán transmitidos al host y cuáles serán descartados. Además, permite recopilar estadísticas sobre los frames recibidos a nivel de capa de acceso a red como se verá más adelante en el apartado de experimentos. El módulo Socket permite la programación de los sockets RAW, UDP y TCP mediante 19 funciones. El módulo Netapp proporciona funciones relacionadas con el DHCP (Dynamic Host

Configuration Protocol) para obtener la IP de la red, DNS (Domain Name System) para la gestión de nombres y Ping. El módulo NetCfg tiene dos funciones sl_NetCfgSet y sl_NetCfgGet que permiten configurar y obtener respectivamente las direcciones IP y en la capa de enlace la MAC de 48-bit. Por último, el módulo File System permite gestionar ficheros que pueden utilizar tanto el usuario como el chip de red. Dispone de 7 funciones que permiten por ejemplo crear un fichero y escribir datos o leer desde un fichero almacenados en el propio dispositivo.

2.8 FreeRTOS

Para implementar el cliente MQTT se ha utilizado el kernel de tiempo real FreeRTOS. En un sistema donde el procesador sólo lleva un core, un sólo hilo o tarea puede ejecutarse en un momento dado. Un RTOS como FreeRTOS decide qué tarea se ejecuta utilizando la prioridad que tiene asignada cada tarea. El kernel se encarga de gestionar el tiempo asignado a cada tarea lo cual proporciona mayor eficiencia que si se realiza un polling hasta que tenga lugar un evento. FreeRTOS permite sincronizar tareas ejecutándolas sólo cuando tienen trabajo que realizar lo cual resulta en mayor eficiencia y permite un ahorro energético.

2.9 MongoDB

Para guardar los datos de la aplicación de domótica se va a utilizar la base de datos NoSQL (Not Only SQL) MongoDB. Esta base de datos está diseñada para almacenar grandes volúmenes de datos como los generados por los dispositivos IoT. Utiliza el formato BSON (Binary Object Notation) que es una versión ligera de JSON. Se considera una base de datos de alto rendimiento y muy escalable. Los datos se almacenan en MongoDB como documentos y un conjunto de documentos similares se agrupan en colecciones. El tamaño máximo de un documento son 16 MB.

2.10 Node.js

Node.js fue creado en 2009, está programado en C++ y utiliza Chrome V8 que es un intérprete de JavaScript que compila directamente a código máquina. Node.js funciona mediante módulos que son librerías que permite añadir funcionalidad. Los módulos se agrupan en paquetes. Node Package Manager permite acceder a Node Package Registry desde la línea de comandos. Node Package Registry es un repositorio que permite publicar módulos, buscar módulos y bajar módulos. El modelo tradicional de los servidores web se base en threads o modelo lineal donde cada petición es un thread que hace uso de la CPU. Las peticiones bloqueantes acceden a los dispositivos de I/O y la ejecución del thread se para. Algunos ejemplos de peticiones bloqueantes son la lectura de un fichero o realizar una consulta a una base de datos. Sin embrago, Node.js se basa en el modelo orientado a eventos donde pueden atenderse aproximadamente 6.000 peticiones por segundo por core. Esto es posible gracias a una cola FIFO donde las peticiones bloqueantes como el acceso a una base de datos se realizan mediante funciones callback. Una función callback en Node.js es equivalente a una función asíncrona. Esto permite que Node.js sea escalable por su capacidad de procesar gran volumen de peticiones sin esperar a que la función devuelva el resultado.

Entre las principales ventajas de Node.js se encuentra la facilidad que proporciona programar tanto el cliente como el servidor utilizando JavaScript. Además, el modelo asíncrono permite crear aplicaciones

escalables ya que el número de peticiones no afecta tanto el rendimiento como ocurre en el modelo tradicional basado en threads. Proporciona un nivel de abstracción elevado lo cual facilita el desarrollo de aplicaciones como, por ejemplo, las que utilizan el protocolo HTTP. Además, Node.js tiene un driver para MongoDB y al utilizar JavaScript es idóneo para trabajar con datos en formato JSON.

3 Desarrollo

El sistema es una PoC (Prueba de Concepto) de las tecnologías anteriormente citadas. Dicha prueba consiste en una aplicación de domótica inalámbrica de tiempo real (Fig. 3-1) que permita obtener la temperatura de 3 habitaciones de una casa utilizando tanto microcontroladores de bajo consumo energético y protocolos ligeros que permitan minimizar el uso de recursos hardware.

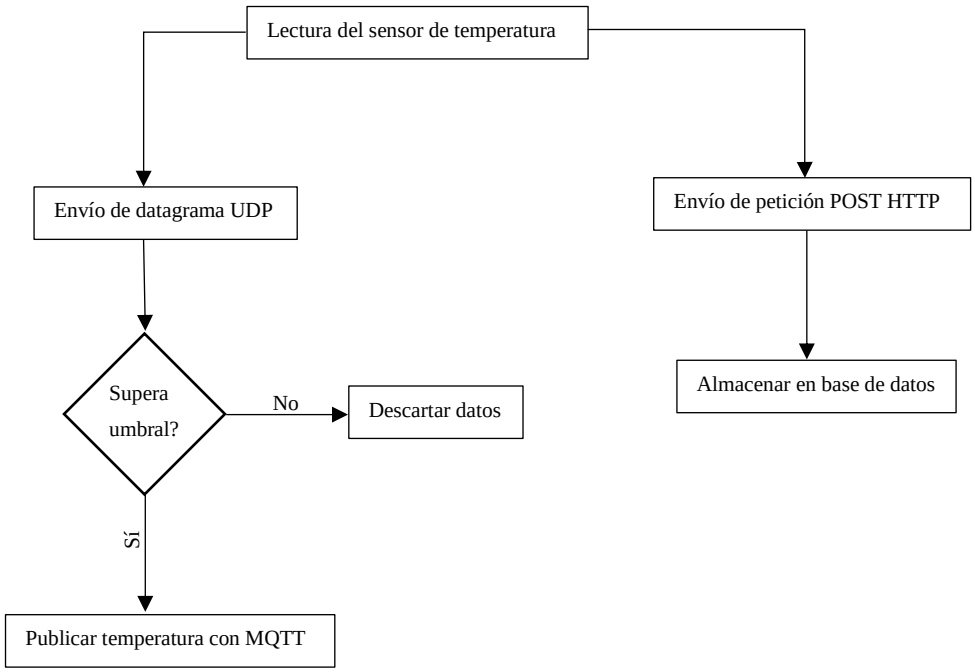

Figura 3-1 Funcionamiento de la aplicación de domótica. Fuente: elaboración propia.

El desarrollo de la aplicación está dividido en varios apartados. El primero consiste en crear un cliente UDP utilizando la plataforma Tiva C Series Launchpad que permita enviar datos de temperatura mediante datagramas UDP a una plataforma MSP430. En este apartado también se implementará una función que permita obtener la temperatura para cada una de las habitaciones utilizando un sensor. También se va a implementar en la plataforma Tiva C Series Launchpad un cliente HTTP que permita enviar la temperatura obtenida del sensor junto al identificador de la habitación en formato JSON a un servidor web para que sean almacenados en una base de datos.

El siguiente apartado consiste en crear un servidor UDP utilizando la plataforma MSP que permita recibir los mensajes enviados desde la plataforma Tiva C Series Launchpad mediante datagramas UDP. Una vez se reciban los datos de temperatura, se procesarán y si superan un umbral predefinido será publicado un mensaje de aviso utilizando un bróker MQTT para que pueda recibir cualquier cliente que se subscriba al tema bajo el que se publicarán los mensajes. El último apartado muestra el desarrollo del servidor HTTP que recibirá los datos de la plataforma Tiva C Series Launchpad para su posterior almacenamiento.

3.1 Arquitectura del sistema

El hardware necesario para implementar la aplicación de domótica (Fig. 3-2) son tres sensores de temperatura LM34 de TI, tres placas de evaluación Tiva C Launchpad de TI, una placa de evaluación MSP430 de TI, cuatro chips de red CC3100 de TI, un AP y un portátil.

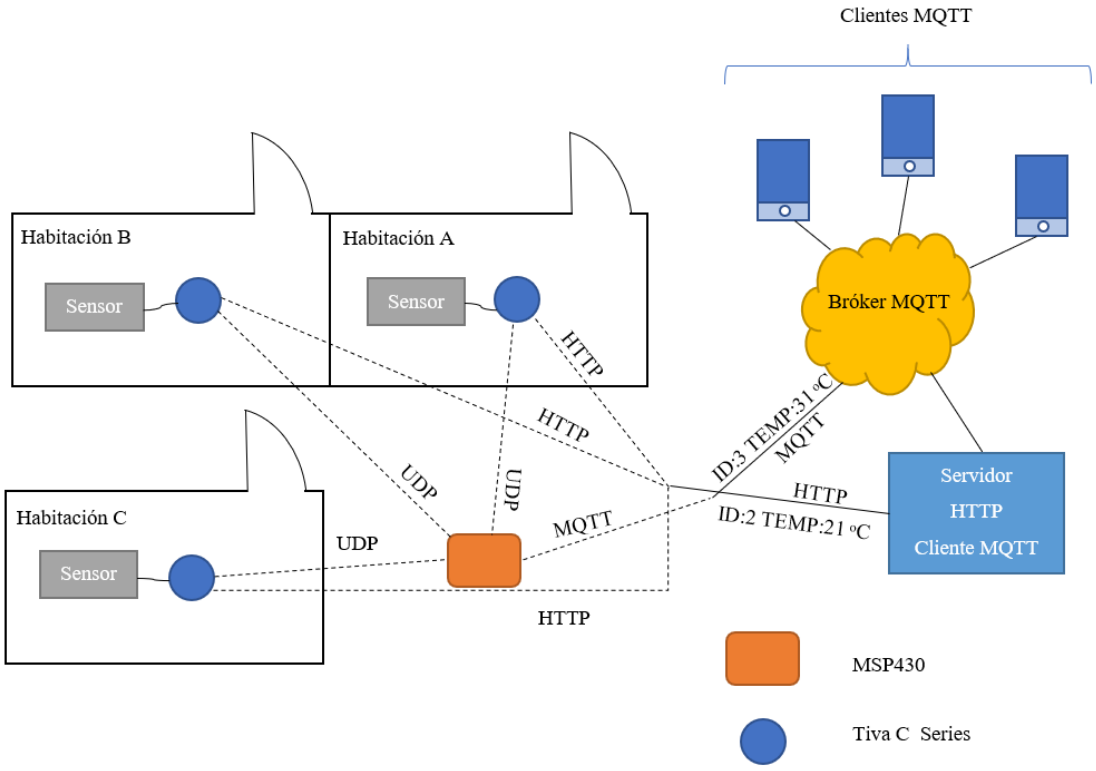

Figura 3-2 Disposición general de la arquitectura de la aplicación de domótica. Fuente: elaboración propia.

Las plataformas Tiva C Series Launchpad tienen cada una un sensor de temperatura LM34. Estas plataformas realizan tres tareas (Fig. 3-3).

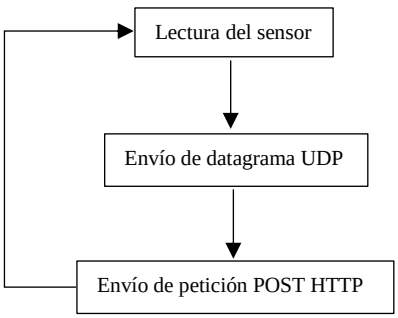

Figura 3-3 Tareas realizadas por la Plataforma Tiva C series Launchpad. Fuente: elaboración propia.

La primera tarea (Fig. 3-4) consiste en realizar una lectura de temperatura del sensor. La segunda tarea consiste enviar los datos de temperatura junto con un el identificador de la habitación mediante UDP a través del AP a la plataforma MSP430.

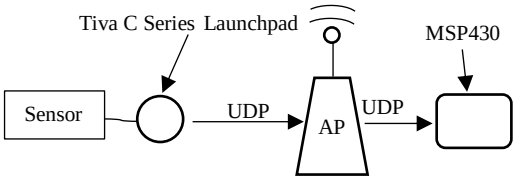

Figura 3-4 Envío de datagramas UDP desde la plataforma Tiva C Series Launchpad a la plataforma MSP430. Fuente: elaboración propia.

La tercera tarea (Fig 3-5) consiste en enviar los datos de temperatura junto con un el identificador de la habitación mediante HTTP al AP en formato JSON. Esta tarea no puede realizarse en la plataforma MSP430 ya que el dispositivo consta sólo de 8 KB que son insuficientes para soportar MQTT y HTTP. El AP hace llegar al servidor web implementado en Node.js los datos. Cuando el servidor web recibe la lectura del sensor de temperatura y el identificador de la habitación, procede a almacenar esta información en una base de datos MongoDB.

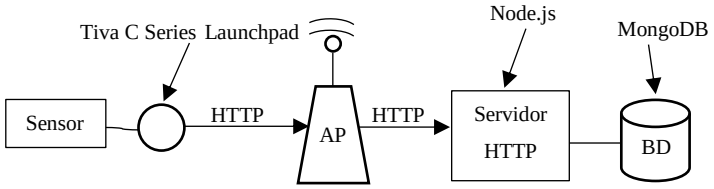

Figura 3-5 Envío de datos desde la plataforma Tiva C Series Launchpad al servidor mediante HTTP. Fuente: elaboración propia.

La plataforma MSP430 realiza dos tareas (Fig 3-6).

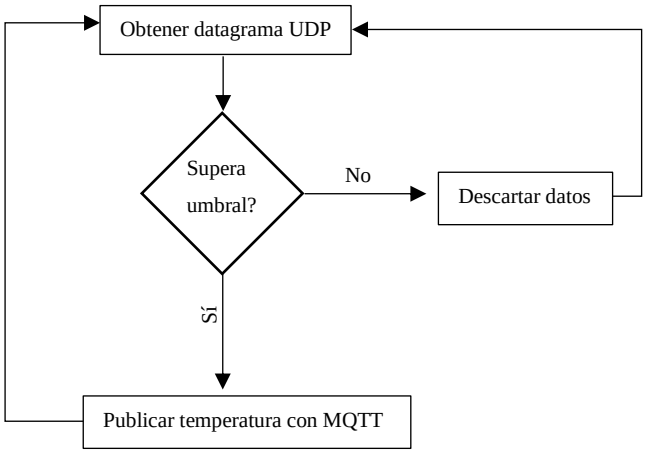

Figura 3-6 Tareas realizadas por la Plataforma MSP430. Fuente: elaboración propia.

La primera consiste en una comprobación sobre la temperatura que recibe de las plataformas Tiva C Series Launchpad. Si la temperatura supera un umbral realiza una segunda tarea que consiste en el envío de un mensaje MQTT (Fig.3-7) para ser publicado por el bróker MQTT. Para ello, la placa MSP accede al Internet como publisher MQTT enviando mensajes con los datos sobre la temperatura de las habitaciones avisando que se ha superado un umbral. En caso, por ejemplo, de encontrarse el sensor en una cámara frigorífica puede enviar de inmediato a los responsables un aviso de que se ha superado el umbral de seguridad.

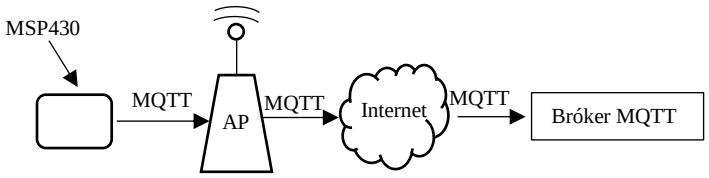

Figura 3-7 Envío de mensaje MQTT desde la plataforma MSP430 al bróker MQTT. Fuente: elaboración propia.

Por tanto, la plataforma MSP430 actúa como servidor UDP y cliente MQTT y las plataformas Tiva C Series Launchpad por su parte actúan como clientes HTTP y clientes UPD.

3.2 Programación de la plataforma MSP430

La plataforma MSP430 debe ser capaz de recibir los datagramas UDP que se envíen desde las plataformas Tiva C Series Launchpad. Para ello, es necesario implementar[1] un servidor UDP que reciba los datagramas. Además, la plataforma MSP430 debe poder conectarse como publisher a un bróker MQTT. El envío de un mensaje al bróker MQTT está condicionado a que la temperatura supere un umbral. Esta dependencia entre los datos que recibe el servidor UDP y la llamada a una función que publique un mensaje MQTT propicia el uso del RTOS FreeRTOS. Mediante FreeRTOS se puede utilizar un flag y una cola para coordinar dos tareas (Fig. 3-8). La primera tarea se implementa en la función *BsdUdpServer* que inicializa el dispositivo CC3100 al estado por defecto, realiza la conexión al AP para a continuación realizar la configuración y puesta en marcha del servidor UDP. La segunda tarea la lleva a cabo la función *MqttClient* que consiste en conectarse a un bróker MQTT para publicar datos acerca de las temperaturas de las distintas habitaciones si se supera un umbral.

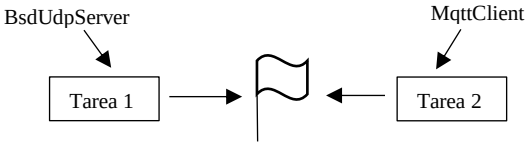

Figura 3-8 Sincronización de las tareas mediante un flag. Fuente: elaboración propia.

[1] Código disponible en https://drive.google.com/drive/folders/1lqTowkURhjRuFvISLLBFYT3_KaffEg-Y?usp=sharing bajo licencia GNU.

La tarea 2 tiene mayor prioridad que la tarea 1 pero sólo se ejecuta una vez se haya completado la inicialización del dispositivo CC3100 y la conexión al AP que se realiza desde la tarea 1. Para coordinar ambas tareas se utiliza un flag de tipo lista enumerada (Cod. 3-1).

```
/*Lista enumerada*/
enum ready{no,yes};
enum ready device_ready;
```

Código 3-1 Flag en forma de lista enumerada. Fuente: elaboración propia.

El scheduler de FreeRTOS conmuta entre las tareas que se encuentra en estado *Running* y *Not Running*. El estado *Not Running* incluye los estados *Blocked*, *Ready* y *Suspended*. Se crea una variable *device_ready* de tipo enum ready tiene un valor *no* la primera vez que se ejecuta la tarea 2 ya que aún no se ha realizado la inicialización y conexión del dispositivo CC3100 al AP.

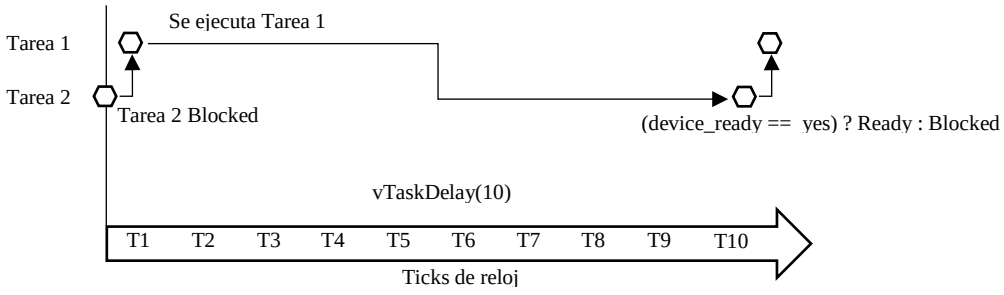

Figura 3-9 Flujo de ejecución de las tareas. Fuente: elaboración propia.

La tarea 2 pasa por tanto al estado *Blocked* durante de 10 ticks de reloj (Fig. 3-9) mediante la función de FreeRTOS *vTaskDelay* (Cod. 3-2) y el scheduler pasa a ejecutar la tarea 1.

```
/*Esperar por el dispositivo*/
do{
   vTaskDelay(10);
}while(device_ready == no);
```

Código 3-2 Estado de bloqueo de la tarea 2. Fuente: elaboración propia.

Cuando han pasado los 10 ticks de reloj, el scheduler mueve ambas tareas al estado *Ready* y se ejecutan nuevamente en orden de prioridad. Si la tarea 1 no ha finalizado la inicialización y conexión del dispositivo CC3100 se repite el ciclo. Una vez concluida la inicialización y conexión del dispositivo al AP se le asigna el valor de *yes* a la variable *device_ready* por lo que la tarea 2 sale del estado de *Blocked* y pasa al estado *Ready* y luego al estado *Running*. La tarea 2 ya no vuelve a entrar en estado *Blocked*. Este sistema es más eficiente que si se realiza un polling sobre la variable *device_ready* ya que mientras las tareas se encuentran en estado *Blocked* no utiliza tiempo del procesador.

3.2.1 Servidor UDP

Los clientes UDP implementados en las plataformas Tiva C Series Launchpad deben conocer con antelación la dirección IP del servidor UDP para poder enviar los datagramas por lo que la IP del servidor UDP debe ser estática. Sin embargo, cuando se inicializa el dispositivo CC3100 SimpleLink para conectarse a un AP el mismo adquiere por defecto una IP dinámica mediante DHCP. Si se desea que adquiera siempre la misma IP estática es necesario utilizar la función *sl_NetCfgSet* dentro de la función *configureSimpleLinkToDefaultState*. Esta función permite deshabilitar DHCP y asignar una IP estática.

La función recibe 4 argumentos donde el primero es el tipo de configuración que se desea que en este caso es modo IP estático y para ello se utiliza la macro *SL_IPV4_STA_P2P_CL_STATIC_ENABLE*. El segundo argumento es la macro *IPCONFIG_MODE_ENABLE_IPV4* que habilita IPv4. El tercer y cuarto son la longitud y la dirección de la estructura que contiene la información de la IP respectivamente. Esta estructura es del tipo *SlNetCfgIpV4Args_t*. La estructura contiene la IP estática asignada, la máscara de red, el gateway y el servidor DNS (Cod. 3-3).

```
/*Configuracion IP estática*/
SlNetCfgIpV4Args_t ipV4;
ipV4.ipV4         = (_u32)SL_IPV4_VAL(192,168,1,201);
ipV4.ipV4Mask     = (_u32)SL_IPV4_VAL(255,255,255,0);
ipV4.ipV4Gateway  = (_u32)SL_IPV4_VAL(192,168,1,1);
ipV4.ipV4DnsServer = (_u32)  SL_IPV4_VAL(80,58,61,250);
```

Código 3-3 Estructura de tipo SlNetCfgIpV4Args_t para almacenar los datos de la IP estática. Fuente: elaboración propia.

Una vez se ha configurado la IP estática se pasa a implementar el servidor UDP (Fig. 3-10) en la función *BsdUdpServer*.

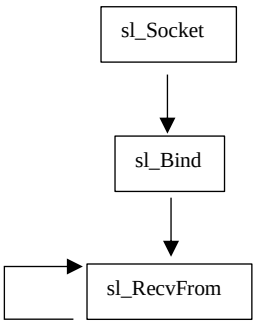

Figura 3-10 Flujo del servidor UDP. Fuente: elaboración propia.

Primeramente, se utiliza la función *sl_Socket* para obtener un socket handle. El socket handle se vincula al puerto por donde van a llegar los datagramas UDP mediante la función *sl_Bind* que tiene 3 parámetros (Fig. 3-11). El primero es el socket handle y el segundo es un puntero a una estructura de 3 miembros que contienen el protocolo que se va a utilizar que es IPv4, el número del puerto y la dirección IP. Para recibir los datagramas UDP que vayan destinados al puerto indicado desde cualquier interfaz se utiliza un *0* como argumento que

corresponde a una dirección genérica de tal forma que se vincula el socket al puerto indicado y a todas las interfaces de host. El tercer argumento es el tamaño de esta estructura.

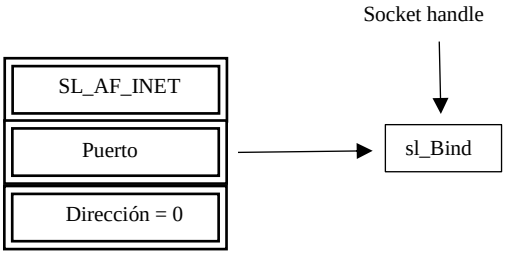

Figura 3-11 Argumentos tomados por la función sl_Bind. Fuente: elaboración propia.

La función sl_*RecvFrom* se utiliza para leer los datos recibidos a través del socket UDP. Esta función toma seis argumentos donde el primero de ellos es el socket handle. El segundo argumento es un puntero al buffer donde se va a almacenar los datos. Los clientes UDP envían dos bytes donde el primero es un entero sin signo con el identificador de la habitación y el segundo es un entero con signo con la temperatura. Esta información está contenida en el payload de un cada datagrama por lo que el buffer del servidor se reduce a dos bytes (Fig. 3-12).

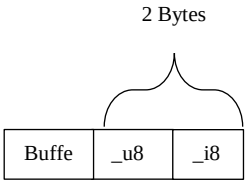

Figura 3-12 Buffer mínimo para almacenar los datos recibidos por UDP. Fuente: elaboración propia.

Por tanto, el buffer consta de dos bytes suficiente para almacenar tanto el identificador de la habitación como la temperatura. El tercer argumento es el tamaño del buffer y el cuarto no se utiliza. El quinto y sexto tienen que ver con la dirección del remitente y su tamaño respectivamente. Una vez implementada esta función se reciben los datos a través de buffer mediante un bucle infinito. Los datos recibidos se almacenan en una estructura de tipo *Room_t* (Cod. 3-4) por lo que se accede al primer y segundo elemento del buffer para extraer el identificador y la temperatura respectivamente.

```
/*Obtener datos del buffer*/
room_data.room_id = uBuf.BsdBuf[0];
room_data.temp = uBuf.BsdBuf[1];
```

Código 3-4 Los datos recibidos se almacenan en una estructura de tipo Room_t. Fuente: elaboración propia.

3.2.2 Publisher MQTT

Una vez recibida la temperatura, se compara con un umbral mínimo y máximo y en caso de sobrepasar estos umbrales se le notifica a la función *MqqtClient* que realiza la tarea de enviar un mensaje al bróker MQTT. La

comunicación entre la tarea *BsdUdpServer* y la tarea *MqqtClientse* realiza mediante una cola. Una cola se implementa usualmente como un buffer FIFO y se utilizan para la comunicación entre tareas. La API SimpleLink incluye la función *osi_MsgQCreate* (Fig. 3-13) para crear una cola y está basada en la función *xQueueCreate* proporcionada por FreeRTOS. La función recibe cuatro argumentos. El primer argumento es un puntero a una estructura de tipo *OsiMsgQ_t.*

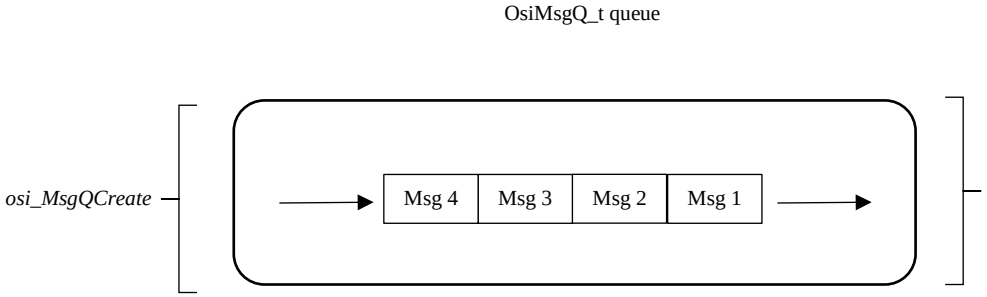

Figura 3-13 Función osi_MsgQCreate crea una cola que se guarda en &queue. Fuente: elaboración propia.

Esta estructura se utiliza tanto para la escritura de datos en la cola como para su lectura. El segundo argumento es un string con el nombre de la cola. El tercer argumento es el tamaño de los mensajes de la cola. Estos mensajes están codificados en una lista enumerada (Cod. 3-5).

```
/* Mensajes de la cola*/
typedef enum events
{TEMPERATURE_STABLE,TEMPERATURE_ALERT,BROKER_DISCONNECTION}osi_messages;
```

Código 3-5 Lista enumerada con los mensajes de la cola. Fuente: elaboración propia.

Si la temperatura supera un umbral se escribe *TEMPERATURE_ALERT* en la cola por medio de la función *osi_MsgQWrite* (Fig. 3-14). Esta función toma tres parámetros que son la dirección de la estructura donde se encuentra almacenada la cola, el mensaje y el tiempo en milisegundos que se espera hasta que la cola tenga espacio disponible para escribir el mensaje.

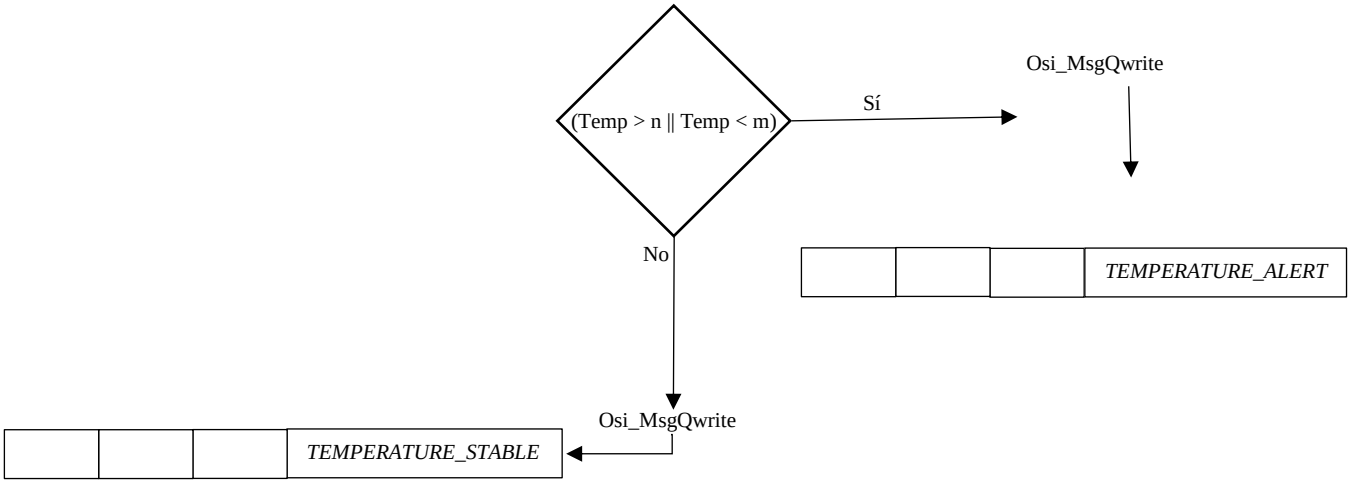

Figura 3-14 Flujo para la escritura en la cola. Fuente: elaboración propia.

El cuarto parámetro de la función *osi_MsgQCreate* es el tamaño de la cola que especifica cual es el número máximo de mensajes que puede almacenar en la misma. La cola que utilizan tanto la tarea 1 para escritura como la tarea 2 para lectura tiene una profundidad de diez mensajes. Los datos con la temperatura se almacenan como un string (Cod. 3-6) para que puedan estar disponibles cuando la función tenga que enviarlos al bróker MQTT.

```
sprintf(room_dat,"Room id: %d Temperature: %d",room_data.room_id, room_data.temp);
```

Código 3-6 Se copian los datos a un string que luego se envía al bróker MQTT. Fuente: elaboración propia.

La función *MqttClient* realiza una la lectura sobre el mensaje de la cola y si es *TEMPERATURE_ALERT* envía un mensaje a un bróker MQTT con el identificador de la habitación y la temperatura (Fig. 3-15). Para crear esta función se utiliza la implementación MQTT de la API SimpleLink.

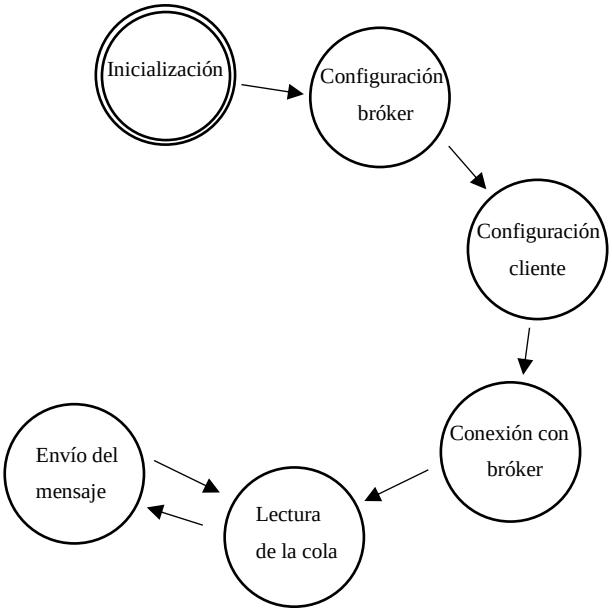

Figura 3-15 Máquina de estados del cliente MQTT. Fuente: elaboración propia.

El flujo del cliente MQTT (Fig 3-16) comienza con la inicialización de la implementación MQTT mediante la función *sl_ExtLib_MqttClientInit* para luego realizar las llamadas a la API.

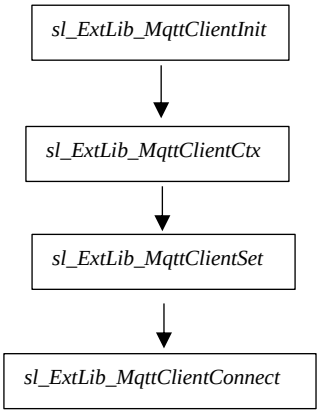

Figura 3-16 Flujo de la conexión del cliente MQTT. Fuente: elaboración propia.

Esta función recibe como argumento una estructura *SlMqttClientLibCfg_t* que está predefinida con parámetros de inicialización. Una vez se ha inicializado la implementación MQTT se puede utilizar la función *sl_ExtLib_MqttClientCtxCreate* para crear la información necesaria que permita al cliente iniciar sesión con un bróker. El primer parámetro de esta función es la información sobre el servidor. El segundo parámetro son las rutinas callback de la aplicación. El tercer parámetro es un puntero al array que contiene la información de configuración de la conexión. La información sobre la conexión se almacena en una estructura de trece miembros de tipo *connection_config* (Tab. 3-1) y contiene la información que se pasa como argumentos a la función *sl_ExtLib_MqttClientCtxCreate*.

Nombre del miembro
Información sobre el bróker
Handle del cliente
Identificador de cliente
Nombre de usuario
Clave de autenticación
Clean session
Keepalive
Rutinas callback
Número de temas
Nombres de los temas
QoS de los temas
Will message
Conexión del cliente

Tabla 3-1 Miembros de la estructura connection_config. Fuente: elaboración propia.

El primer miembro de esta estructura contiene la información sobre el bróker y es a su vez es una estructura de tipo *SlMqttClientCtxCfg_*t (Fig. 3-17).

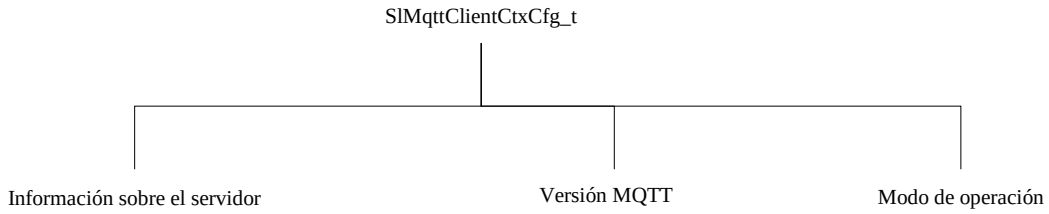

Figura 3-17 Información de configuración del bróker. Fuente: elaboración propia.

La versión MQTT es un boolean que indica si la librería MQTT opera con la versión 3.1 o 3.1.1. El modo de operación es un boolean que indica si al realizar las operaciones de publicar mensajes, subscribirse o anular la subscripción a tópicos se espera la confirmación de forma bloqueante o se utilizan callbacks. La información sobre el bróker es a su vez una estructura de tipo *SlMqttServer_t* (Fig. 3-18).

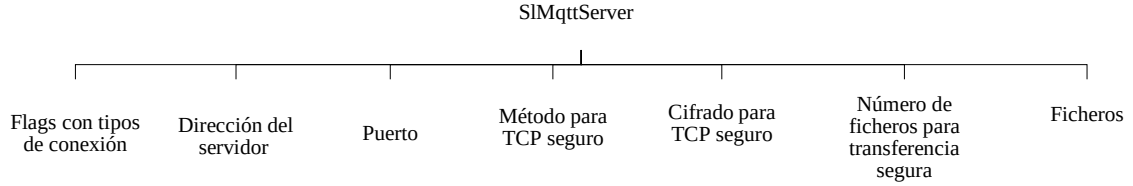

Figura 3-18 Contenido de la estructura SlMqttServer. Fuente: elaboración propia.

Los flags, con el tipo de conexión, se pueden utilizar para indicar si la conexión es IPv6 ya que por defecto es IPv4. También se puede indicar si la dirección del bróker es una URL y no una IP. La tercera opción indica el puerto que para MQTT es habitual utilizar el 1883. La cuarta opción permite indicar que la conexión ha de ser segura utilizando TLS (Transport Layer Security). El método seguro indica el tipo de protocolo de encriptado. El cifrado indica el tipo algoritmo. El número de ficheros necesarios para crear una conexión segura puede variar entre uno y cuatro. El primero es la clave privada, el segundo es el certificado digital, el tercero es la entidad emisora y el cuarto es la clave DH (Diffie–Hellman). Los últimos cuatro miembros se inicializan a null si no se utiliza la conexión segura.

El segundo miembro de la estructura *connection_config* es un puntero a void con el handle del cliente. Este miembro se inicializa a null hasta que reciba el valor que devuelve la función *sl_ExtLib_MqttClientCtxCreate*. El tercer miembro es el identificador de cliente. Cada cliente tiene un identificador único. Este identificador se define en el fichero *mqtt_client.h*. El identificador puede ser de 1 a 23 bytes y es un campo obligatorio. El cuarto miembro es el nombre de usuario utilizado por el bróker para autenticación. El quinto miembro es una clave de autenticación de 0 a 65 KB con un prefijo de dos bytes. El sexto miembro es un boolean e indica si el bróker debe resumir la comunicación con el cliente o debe descartar la sesión previa y empezar una nueva. El séptimo miembro es un keepalive en segundos que el cliente debe enviar de manera periódica en forma de mensaje antes de que timer expire. Un 0 como argumento deshabilita el keepalive.

El octavo miembro es una estructura de tipo *SlMqttClientCbs_t* (Fig. 3-19). Esta estructura tiene tres miembros que son punteros a tres rutinas callback.

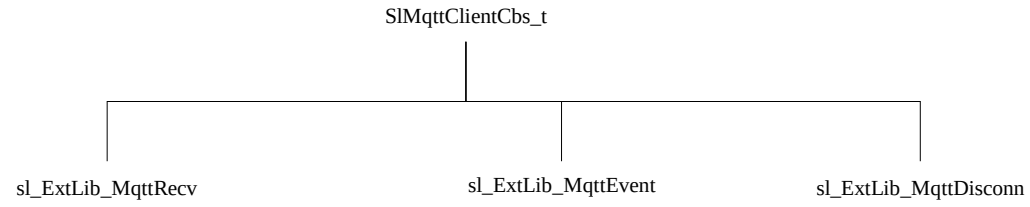

Figura 3-19 Miembros de la estructura SlMqttClientCbs_t. Fuente: elaboración propia.

La rutina *sl_ExtLib_MqttRecv* permite recibir un mensaje enviado por el bróker. Esta rutina la implementa el cliente en caso de que esté subscrito a un tema del bróker. La rutina tiene ocho parámetros donde el primero es el handle del cliente. El segundo parámetro es el nombre del tem al cual el cliente está subscrito. El tercer parámetro es la longitud del nombre del tema. El cuarto parámetro es el payload de mensaje del bróker. El quinto parámetro es la longitud del payload. El sexto parámetro es un boolean que indica si este mensaje está duplicado. El séptimo parámetro indica el QoS del mensaje. El octavo parámetro indica que el mensaje ha sido publicado.

La rutina *sl_ExtLib_MqttEvent* se utiliza para recibir los ACKs del bróker. Esta rutina tiene 4 parámetros. El primero es el handle del cliente. El segundo es un identificador del evento. El tercer argumento es el buffer con el evento y el cuarto el tamaño del buffer.

La tercera rutina es sl_ExtLib_MqttDisconn y notifica al cliente de ha terminado la conexión con el bróker. Esta rutina tiene un sólo parámetro que es el handle del cliente.

El noveno miembro de la estructura *connection_config* es el número de temas a los cuales está subscrito el cliente. El décimo miembro son los nombres de los temas. El undécimo miembro es el QoS de los mensajes que se publican en los temas. El duodécimo miembro es una estructura de tipo *SlMqttWill_t* que contienen la información del will message. El "will message" es un mensaje MQTT normal que utiliza el bróker en caso de desconexión repentina por parte del cliente. Los datos del mensaje son el tema, el último mensaje, el QoS y un flag que indica si el servidor mantiene el mensaje después de ser publicado. El miembro decimotercero de la estructura *connection_config* es un bool que almacena el estado del cliente ya sea conectado o desconectado. Los miembros de la conexión de configuración utilizan macros definidas en el fichero *sl_mqtt_client.h*.

```
/*Cofiguración de conexión*/
connect_config usr_connect_config[] =
{   {   {  {
SL_MQTT_NETCONN_URL,
SERVER_ADDRESS,
PORT_NUMBER,0, 0, 0, NULL}, SERVER_MODE, true,}, NULL, CLIENT_ID,  NULL, NULL, true,
KEEP_ALIVE_TIMER,{0,sl_MqttEvt, sl_MqttDisconnect},SUB_TOPIC_COUNT,
{},{},{WILL_TOPIC, WILL_MSG, WILL_QOS, WILL_RETAIN},false }};
```

Código 3-7 Configuración de conexión para el cliente del servidor UDP Publisher MQTT. Fuente: elaboración propia.

Se ha utilizado la configuración (Cod. 3-7) para la plataforma MSP430 que comienza con la parametrización de los valores del bróker mediante la macro *SL_MQTT_NETCONN_URL*. Primeramente, se indica que la dirección del servidor es una URL no una IP. La macro *SERVER_ADDRESS* contiene la dirección del bróker que en este caso es *m2m.eclipse.org* y *PORT_NUMBER* asigna el número de puerto 1883. Los cuatro últimos miembros de la configuración son null ya que no se utiliza la conexión segura.
A continuación, el handle del cliente se inicializa a null y se indica el identificador del cliente mediante la macro *CLIENT_ID*. No se utiliza nombre de usuario ni clave de autenticación por lo que estos miembros son null. El sexto miembro es true lo que indica que tanto el bróker como el cliente descartan el estado de la última sesión y comiencen una nueva. El siguiente miembro es la macro *KEEP_ALIVE_TIMER* con un valor de 25 segundos. A continuación, se encuentra el miembro que consta a su vez de tres miembros con los nombres de las funciones callback. El cliente actúa como publisher pero no se subscribe a ningún terma por lo que no recibe mensajes del bróker sobre temas. Por ello, la rutina callback *sl_ExtLib_MqttRecv* no se incluye. La macro *SUB_TOPIC_COUNT* indica el número de temas a los cuales el cliente está subscrito que en este caso es 0. El nombre los temas, QoS y will message están vacíos al no trabajar con temas del bróker, estos miembros no son necesarios. El último miembro se inicializa a false hasta que tome el valor de true cuando el cliente cambie de estado al conectar con al bróker.

Después de obtener el handle del cliente mediante la función *sl_ExtLib_MqttClientCtxCreate* se invoca la función *sl_ExtLib_MqttClientSet*. Esta función configura los parámetros de la implementación MQTT y debe llamarse antes de realizar cualquier otra transacción. Permite inicializar el identificador del cliente que es un parámetro obligatorio antes de iniciar una conexión con el bróker. La función toma cuatro argumentos. El primero es handle del cliente MQTT. El segundo argumento es la macro *SL_MQTT_PARAM_CLIENT_ID* que indica a la función que se va configurar el identificador del cliente. El tercer argumento es el identificador del cliente que es el tercer miembro de la estructura de tipo *connection_config* cuyo valor se obtiene de la macro *CLIENT_ID* y el cuarto argumento es el tamaño del identificador.

Una vez configurado el identificador del cliente se llama a la función *sl_ExtLib_MqttClientConnect* para realizar la conexión con el bróker. Esta función recibe tres argumentos. El primero es el handle del cliente que es el segundo miembro de la estructura *connection_config*. El segundo argumento es un boolean que indica si se inicia una sesión nueva o se recupera la última sesión mantenida entre el cliente y el bróker. Este boolean es el sexto miembro de la estructura *connection_config*. El tercer argumento es el keepalive en segundos que es el séptimo miembro de la estructura connection_config y está definido en la macro *KEEP_ALIVE_TIMER*. Una vez se conecta al bróker se puede comenzar a publicar mensajes si el cliente MQTT lee un mensaje *TEMPERATUR_ALERT* (Cod. 3-8) en la cola.

```
/*Lectura de la cola*/
for(;;)
{osi_MsgQRead( &g_PBQueue, &RecvQue, OSI_WAIT_FOREVER);
if(TEMPERATURE_ALERT ==
RecvQue){publishData((void*)local_con_conf[iCount].clt_ctx, pub_topic_1,
room_dat);}
```

Código 3-8 Lectura de la cola y publicación del mensaje en el servidor MQTT. Fuente: elaboración propia.

La lectura de la cola se realiza (Fig. 3-20) mediante la función *osi_MsgQRead* que toma tres argumentos. El primero es la dirección de la cola. El segundo argumento es la dirección donde se guarda el mensaje. El tercer argumento indica en milisegundos cuanto tiempo se mantiene en esta función a la espera de que haya un mensaje disponible en la cola. La macro *OSI_WAIT_FOREVER* indica que se espera el máximo tiempo posible por lo que el cliente se queda esperando en esta función hasta que llegue un mensaje. Cuando se recibe el mensaje *TEMPERATURE_ALERT* se envía al servidor para su publicación bajo el tema indicado por medio de la función *publishData*.

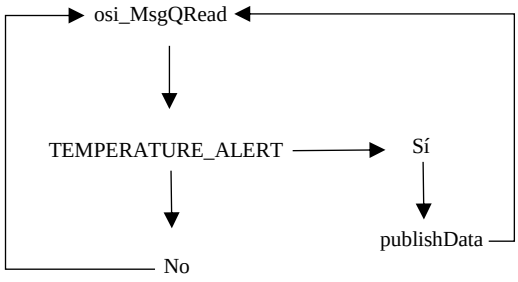

Figura 3-20 Flujo de la lectura sobre la cola. Fuente: elaboración propia.

La función *publishData* toma tres argumentos. El primero es el handle del cliente. El segundo es el tema bajo el que se public el mensaje y el tercero es el mensaje que se quiere publicar. Esta función realiza una llamada a la función *sl_ExtLib_MqttClientSend* (Cod. 3-17) que toma seis argumentos.

```
/*Realizar envío de datos al bróker MQTT*/
sl_ExtLib_MqttClientSend((void*)clt_ctx, publish_topic ,publish_data, pal_Strlen(publish_data), QOS0, RETAIN);
```

Código 3-9 Función sl_ExtLib_MqttClientSend con los argumentos utilizados por el servidor UDP publisher MQTT. Fuente: elaboración propia.

El primer argumento es el handle del cliente. El segundo argumento es el tema bajo el cual se va a publicar el mensaje. El tercero argumento son los datos a publicar. El cuarto argumento es la longitud de los datos. El quinto es el QoS que tiene el mensaje. El sexto utiliza la macro *RETAIN* e indica al bróker que mantenga el mensaje después de publicarlo.

Los mensajes publicados se pueden consultar desde un móvil utilizando una aplicación gratuita para Android como puede ser MyMQTT disponible en Google Play. Para realizar la configuración es necesario introducir la URL del bróker que es *m2m.eclipse.org* y el puerto es el 1883. El tema donde se publican los mensajes es */ucm/mii/tfm* (Fig. 3-20A). Con esta configuración se recibirán mensajes sólo si se supera un umbral de temperatura.

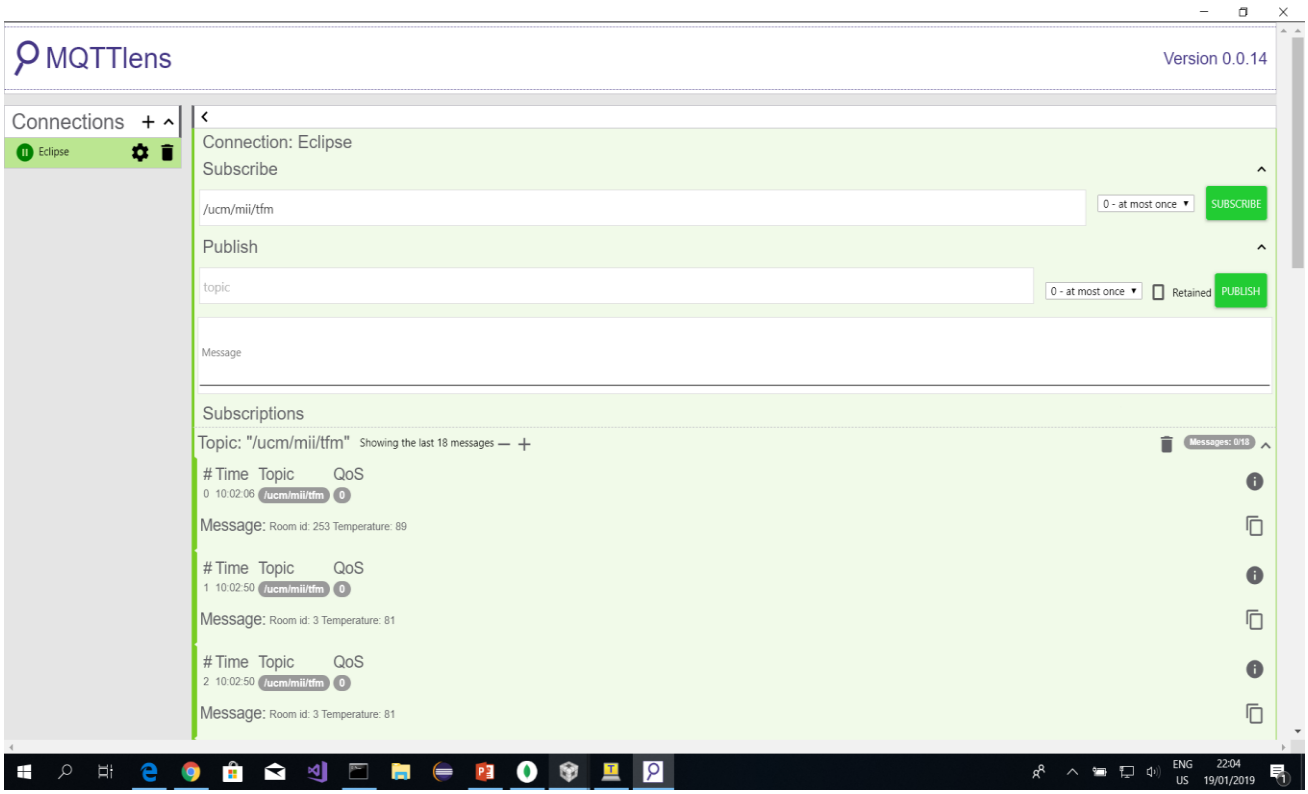

Figura 3-20A Recepción de mensajes MQTT.

Se ha utilizado el 99% de la memoria RAM y el 95 % de la memoria Flash 1 disponible en la plataforma MSP430 para realizar esta implementación.

3.3 Programación de las plataformas Tiva C Series Launchpad

Las plataformas Tiva C Series Launchpad se encargan de enviar los datos obtenidos del sensor de temperatura a un servidor UDP a través de un socket[2]. Además, deben enviar estos mismos datos codificados en fomato JSON a un servidor HTTP. Estas funcionalidades implementadas en la función *HttpUdpClient*, al no necesitar conmutar entre varias tareas, se realizan bare metal o sin sistema operativo mediante un bucle infinito (Fig. 3-21).

[2] Código disponible en https://drive.google.com/drive/folders/1lqTowkURhjRuFvISLLBFYT3_KaffEg-Y?usp=sharing bajo licencia GNU.

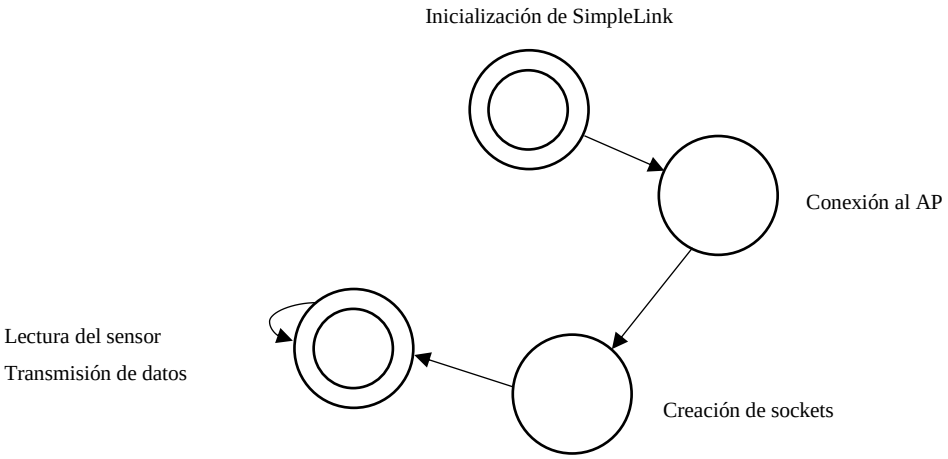

Figura 3-21 Máquina de estados del cliente UDP HTTP. Fuente: elaboración propia.

El primer paso de la secuencia es la inicialización del dispositivo CC3100 SimpleLink. La misma se realiza mediante la función *configureSimpleLinkToDefaultState* facilitada por la API SimpleLink. Seguidamente, se realiza la conexión al AP mediante la función *establishConnectionWithAP*. Esta función hace una llamada a la función sl_*WlanConnect* que toma cinco argumentos y pertenece al módulo Wlan (Fig. 3-22) de la API SimpleLink.

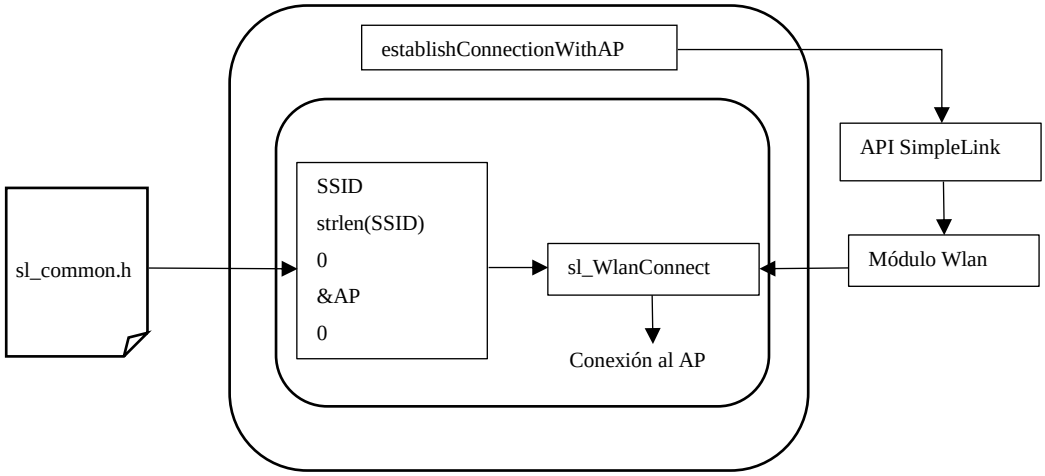

Figura 3-22 Función establishConnectionWithAP. Fuente: elaboración propia.

Los parámetros tres y cinco no se utilizan por lo que se pasa null como argumentos. Es necesario indicar el nombre del AP y la longitud del mismo. El cuarto argumentos es la dirección de una estructura, &AP, de tipo *SlSecParams_t* definida en el fichero *wlan.h* que contiene el SSID, la clave y tipo de seguridad del AP que se debe haber sido previamente definida en el fichero de cabecera *sl_common.h* proporcionado por la API SimpleLink.

La función se queda realizando un polling sobre dos bits hasta que estos bits indiquen que la conexión ha sido correcta y que se ha adquirido una IP. Al no estar utilizando un sistema operativo es necesario llamar a la función *SlNonOsMainLoopTask* cada vez que se hace polling. Esta función permite atrapar los nuevos valores de variables globales que se actualizan de forma asíncrona. Por ejemplo, una vez se produce el evento de conexión a un AP, la función callback *SimpleLinkWlanEventHandler* lo notifica. Entonces la función

SlNonOsMainLoopTask captura esta función callback y la ejecuta. Esto permite actualizar los datos necesarios que es este caso es la variable global que almacena el valor del bit *STATUS_BIT_CONNECTION* que permite salir del bucle principal. Además de verificar la conexión es necesario saber si se ha obtenido una IP por lo que se utiliza otra función callback llamada *SimpleLinkNetAppEventHandler* encargada de actualizar el valor de la variable que almacena el *STATUS_BIT_IP_ACQUIRED*.

La función principal, *HttpUdpClient*, se apoya en siete funciones auxiliares para realizar las tareas (Fig. 3-3) de inicialización del sensor, configuración del cliente HTTP, configuración del cliente UDP, conexión al servidor HTTP, lectura del sensor, envío de datos por UDP y envío de datos por HTTP.

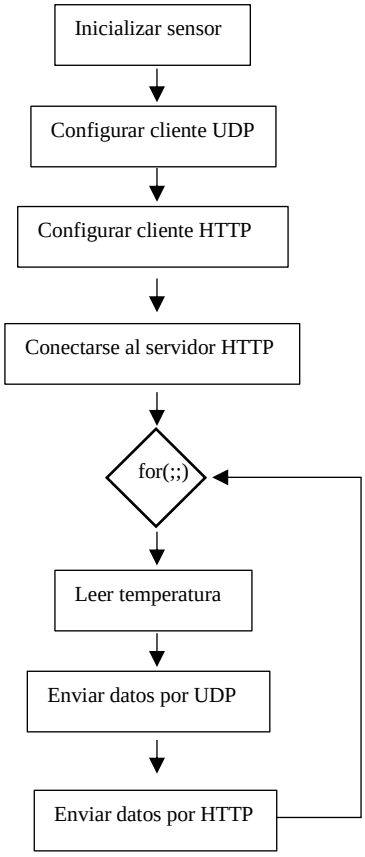

Figura 3-23 Secuencia de las funciones auxiliares contenidas en la función HttpUdpClient. Fuente: elaboración propia.

3.3.1 Cliente HTTP

La API de SimpleLink incluye una implementación de HTTP genérica que se puede portar a cualquier plataforma compatible con el dispositivo CC3100 SimpleLink. La API HTTP es una implementación de HTTP/1.1 que permite realizar peticiones POST, GET, PUT y DELETE aunque en este caso el cliente HTTP sólo va a realizar peticiones POST. Para ello son necesario tres ficheros (Tab. 3-2).

Fichero	Contenido
httpcli.c	Funciones para la implementación del cliente
httpsrc.c	Contiene los tipos MIME
ssock.c	Funciones para la gestión de los sockets

Tabla 3-2 Descripción de los ficheros necesarios para crear la librería HTTP. Fuente: elaboración propia.

Con estos ficheros se puede crear una librería estática (Fig. 3-24) que permita utilizar las funciones una vez compiladas y asociada la ruta al linker. Previamente a compilar la librería es necesario incluir la cabecera <*stdint.h*> en el fichero *httpcli.h* antes que cualquier otro *#include* para evitar conflicto de tipos. Es necesario además incluir *__SL__* como símbolo predefinido para evitar recibir un mensaje de error que indica una configuración de red no reconocida.

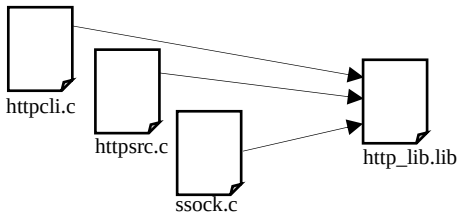

Figura 3-24 Ensamblado de la librería HTTP. Fuente: elaboración propia.

Para conectar con un servidor web se utiliza la función *ConnectToHTTPServer* que toma como argumento un handle que ha de ser declarado de tipo *HTTPCli_Struct*. Dentro de la función, se utiliza la dirección del servidor almacenada en la macro *HOST_NAME* que debe definirse previamente mediante un string de tipo "192.168.1.33". Para convertir este string a una dirección IP válida se utiliza la función *sl_NetAppDnsGetHostByName* que pertenece al módulo Netapp de la API SimpleLink, la cual accede al DNS y escribe la IP del servidor en un entero sin signo de 4 bytes. Esta función toma cuatro argumentos que son o bien un string con el nombre del servidor HTTP o directamente la IP también en formato string, el tamaño de esta cadena de caracteres, la dirección del entero sin signo de 32-bit y la macro *SL_AF_INET* definida en el fichero *socket.h* que indica que el protocolo usado por el socket es IPv4. La función accede al DNS (Fig. 3-25) y devuelve la IP como un entero de 32-bit, pero si es en formato IP devuelve directamente el número sin acceder a DNS.

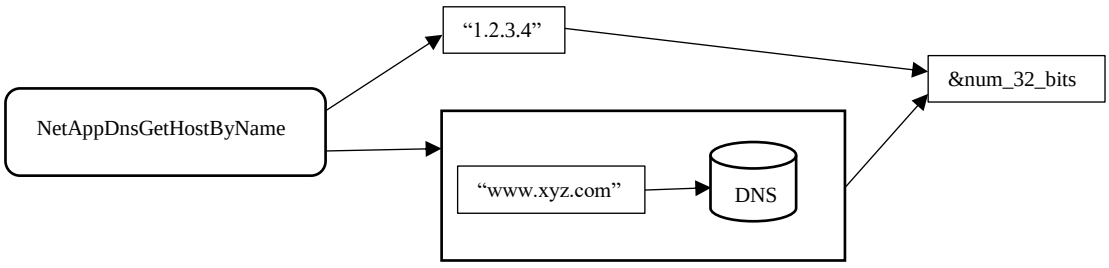

Figura 3-25 Secuencia de la función NetAppDnsGetHostByName. Fuente: elaboración propia.

Es necesario rellenar una estructura de tipo *SlSockAddrIn_t* definida en el fichero *socket.h* con la información del lado del servidor del socket. En esta estructura se indica el protocolo que es IPv4, el número de puerto y la dirección IP que esta codificada en un entero sin signo de 32 bit (Fig 3-26).

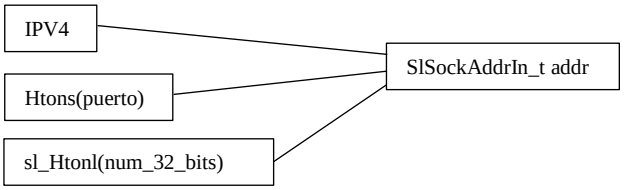

Figura 3-26 Valores asignados a los miembros de la estructura *SlSockAddrIn_t.* Fuente: elaboración propia.

Hay que convertir tanto el puerto como la dirección de little endian, que es por defecto el sistema utilizado por la plataforma Tiva C Series Launchpad, a big endian que es el utilizado en la red. Es necesario, además, crear una instancia del cliente HTTP realizando una llamada a la función *HTTPCli_construct* de la API HTTP (Fig. 3-27) con el handle pasado como argumento a *ConnectToHTTPServer* para finalmente llamar a la función *HTTPCli_connect* perteneciente también a la API HTTP que toma cuatro argumentos donde los primeros dos son el handle y la información del lado servidor del socket. El tercer y cuarto parámetro no se usan por lo que son null.

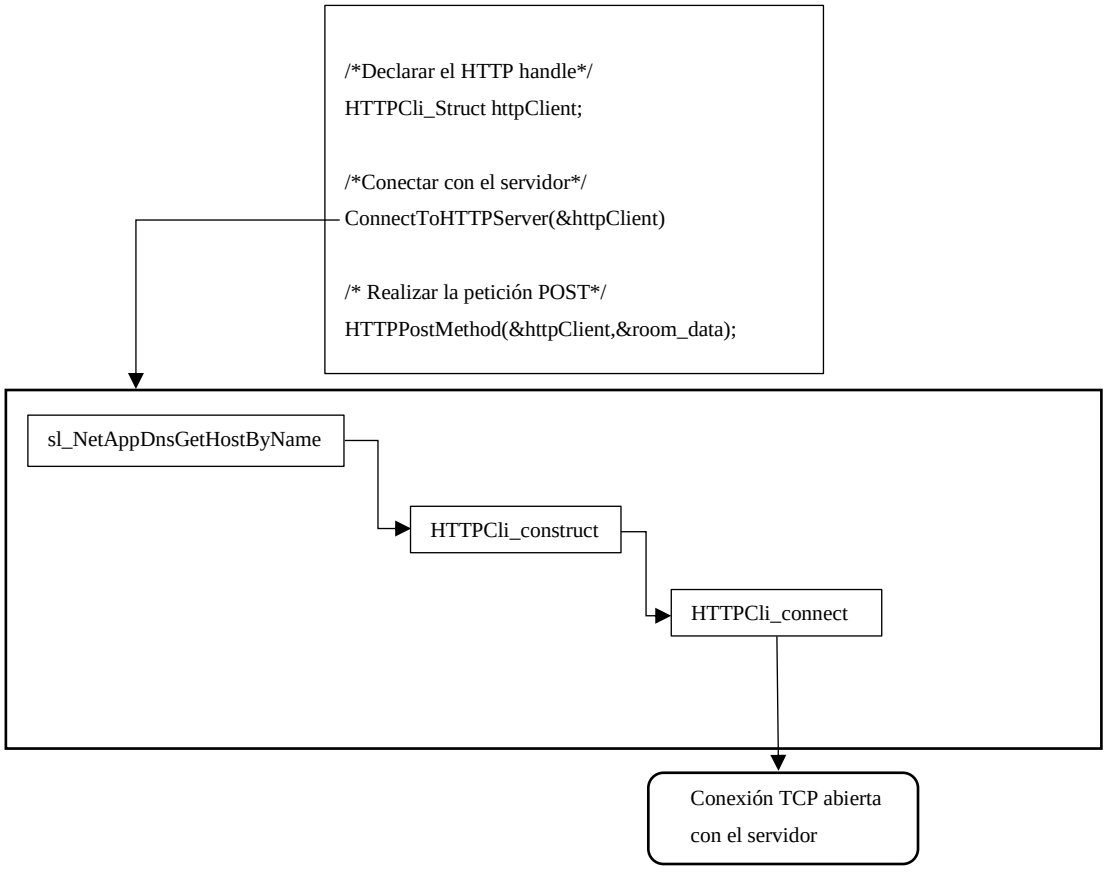

Figura 3-27 Secuencia de la configuración del cliente HTTP. Fuente: elaboración propia.

La función *HTTPPostMethod* realiza el envío de los datos al servidor y toma como primer argumento un handle al igual que *ConnectToHTTPServer*. El segundo argumento es la dirección de la estructura que almacena la temperatura de la habitación y su identificador. La temperatura del sensor puede ser negativa por lo que debe guardarse en un entero de 1 byte con signo. El identificador de la habitación es un entero de 1 byte sin signo. Para los datos de una habitación se utiliza la estructura *Room_t* (Cod. 3-10).

```
typedef struct room{
_i8 temp;
_u8 id;
}Room_t;
```

Código 3-10 Estructura para almacenar la temperatura y el identificador de la habitación. Fuente: elaboración propia.

La construcción del objeto JSON (Cod. 3-11) se realiza a partir de los miembros de la estructura *Room_t*.

```
static _i32 HTTPPostMethod(HTTPCli_Handle httpClient, Room_t * room_data)
{
        sprintf((char *)json_obj, "{\"id\":%d,\"temp\":%d}",room_data->room_id,room_data->temp)}
```

Código 3-11 Construcción de JSON a partir de la temperatura y el identificador de la habitación. Fuente: elaboración propia.

La codificación de las cabeceras HTTP se realiza utilizando un array de estructuras de tipo *HTTPCli_Field* (Fig. 3-28) definido en el fichero *httpcli.h*.

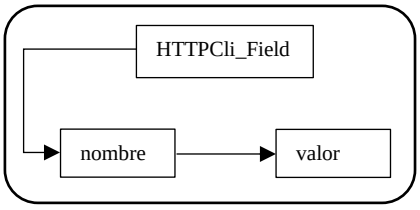

Figura 3.28 Estructura de la cabecera HTTP. Fuente: elaboración propia.

El número total de cabeceras (Fig. 3.29) es cuatro donde la primera es el nombre del servidor, esta cabecera es obligatoria. La segunda cabecera indica el tipo de datos que el cliente acepta del servidor. Aunque esta implementación del cliente no recibe datos del servidor se incluye esta cabecera para que acepte cualquier tipo de datos, */*, en caso de ser necesario en futuras ampliaciones. La tercera cabecera le indica a servidor el tipo de MIME (multipurpose internet mail extension) que se va a enviar que en este caso es *application/json* y es obligatorio incluir una cuarta cabecera null para indicar el fin de las cabeceras.

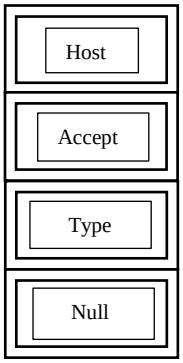

Figura 3-29 Array de cabeceras HTTP. Fuente: elaboración propia.

Este array se le pasa a la función *HTTPCli_setRequestFields* junto con el handle para asociarlo al mismo. Antes de enviarle las cabeceras al servidor es necesario enviar la línea de petición. La misma está estipulada en el *RFC 2616 Hypertext Transfer Protocol -- HTTP/1.1* (Fig. 3-30) e incluye el método utilizado de la petición, la URI y la versión HTTP.

Request-Line = Method **SP** Request-URI **SP** HTTP-Version **CRLF**

Figura 3-30 Línea de petición según descrita en el RFC 2616. Fuente: elaboración propia.

El método a utilizar es POST, la URI es */post* y la versión HTTP es 1.1. Esta línea de petición se envía al servidor mediante la función *HTTPCli_sendRequest* que además tiene la tarea de enviar las cabeceras (Fig. 3-31). Esta función toma 4 argumentos que son el handle, el método, la URI y un flag que se le asigna un 1 si se van a enviar más cabeceras adicionales a las ya definidas.

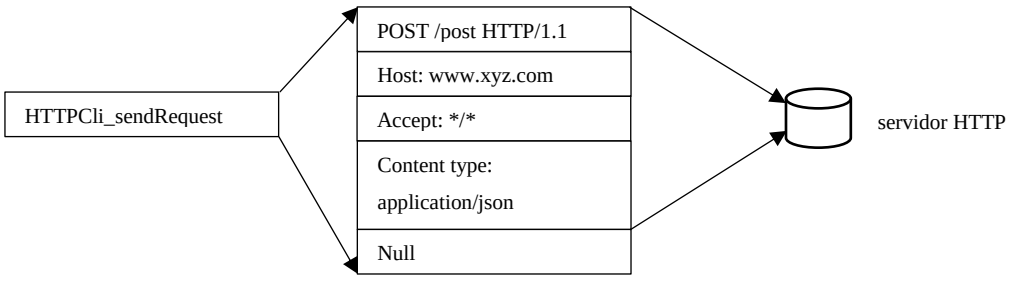

Figura 3-31 Secuencia de petición POST. Fuente: elaboración propia.

En este caso sí se va a enviar una cabecera adicional con el tamaño del JSON construido mediante la función auxiliar *HTTPCli_sendField*. Esta función toma 4 argumentos que son el handle, el tipo de cabecera adicional que en este caso es *content-length*, el tamaño de la cadena de caracteres JSON sin contar el carácter nulo *'/0'* y un flag al cual se le asigna un 1 si no se se van a enviar más cabeceras adicionales. El propósito de enviar la longitud del contenido es saber si el servidor está en dispuesto a aceptar el cuerpo de la petición antes de enviarla. Sobre el tamaño del JSON, hay que tener en cuenta que la temperatura puede ser de uno, dos o tres caracteres. Por ejemplo, si la temperatura es de -10 esto se codifica como tres caracteres ya que la función *sprintf* introduce el símbolo menos (-) para indicar negativo. Teniendo este detalle en cuenta se puede enviar el cuerpo de la

petición POST mediante la función *HTTPCli_sendRequestBody* pasándole los tres argumentos que toma que son el handle, el string con el JSON y el tamaño del string JSON.

3.3.2 Sensor de temperatura

Para programar[3] el sensor de temperatura se va a utilizar el modelo de programación DRA (Direct Register Access) que permite escribir valores directamente sobre los registros del dispositivo. La plataforma dispone de dos módulos analógico digital con una precisión de 12-bit por tanto puede distinguir entre 4096 valores. El valor de entrada es un voltaje analógico y el dispositivo permite distinguir cambios en este voltaje de 0.8 mV. El sensor LM34 tiene como salida un voltaje con una proporción lineal (Fig. 3-32) a grados Fahrenheit y produce 10mV por cada cambio de grado en la temperatura.

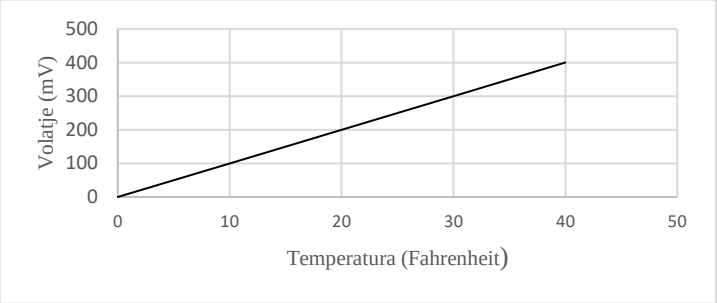

Figura 3-32 Relación lineal entre temperatura y voltaje. Fuente: elaboración propia.

La secuencia (Tab. 3-3) para utilizar uno de los módulos ADC consiste en habilitar un reloj para permitir el acceso a los registros de módulo.

Tarea
Asignar un reloj al puerto
Asignar un reloj al módulo ADC
Habilitar las funciones hardware del pin
Deshabilitar las funciones digitales del pin
Habilitar las funciones analógicas del pin
Deshabilitar el secuenciador
Configurar el evento que activa el muestreo
Seleccionar el canal por donde se realiza el muestreo
Configurar el secuenciador de muestreo
Habilitar el secuenciador

Tabla 3-3 Configuración del módulo ADC. Fuente: elaboración propia.

El registro se llama *RCGCADC* que consta de 32-bit donde los bits [31:2] están reservados y los bits [1:0] permiten activar el módulo ADC asignándole un reloj. También hay que habilitar un reloj para activar el pin del puerto donde está la entrada analógica. Para ello utiliza el registro *RCGCGPIO* que permite habilitar los seis puertos disponibles en el dispositivo mediante los bits [5:0].

[3] Código disponible en https://drive.google.com/drive/folders/1lqTowkURhjRuFvISLLBFYT3_KaffEg-Y?usp=sharing bajo licencia GNU.

Las funciones hardware de un canal ADC son funciones especiales de los pines y deben activarse asignándole un 1 al bit *AFSEL*. La entrada analógica del módulo 0 ADC *AIN0* se encuentra en el pin tres del puerto E. Para activar la función de entrada analógica (Cod. 3-12) se asigna un 1 al bit *AFSEL* del pin tres del puerto E. En el sensor se debe conectar la salida analógica al pin tres del puerto E.

```
/*Habilitar la función especial de entrada analógica del pin tres del puerto E*/
GPIO_PORTE_AFSEL_R |= 8;
```

Código 3-12 Configuración de entrada analógica. Fuente: elaboración propia.

Es necesario deshabilitar las funciones digitales (Cod.3-13) del pin a utilizar como entrada analógica por lo que se tiene que asignar un 0 al bit *DEN* para el pin tres del puerto E.

```
/*Deshabilitar la entrada digital del pin tres del puerto E*/
GPIO_PORTE_DEN_R &= ~8;
```

Código 3-13 Desactivación de función digital. Fuente: elaboración propia.

Para habilitar las funciones analógicas (Cod. 3-14) del pin se escribe un uno en el bit AMSEL.

```
/*Habilitar las funciones analógicas del pin tres del puerto E*/
GPIO_PORTE_AMSEL_R |= 8;
```

Código 3-14 Habilitación de funciones analógicas. Fuente: elaboración propia.

Una vez el pin está habilitado para recibir entradas analógicas se procede a configurar el secuenciador.
El secuenciador pertenece al módulo ADC y realiza la tarea de mover el resultado de la conversión a una de las FIFO disponibles. Existen cuarto [3:0] secuenciadores donde el cuarto, *SS3*, tiene asociado una FIFO capaz de almacenar una muestra. Para evitar errores en caso de dispararse un evento mientras se configura es necesario deshabilitarlo (Cod. 3-15) escribiendo un 0 en el bit asociado al secuenciador en el registro *ACTSS*.

```
/*Deshabilitar el cuarto secuenciador*/
ADC0_ACTSS_R &= ~8;
```

Código 3-15 Desactivación del secuenciador. Fuente: elaboración propia.

Para comenzar a tomar muestras se utiliza el registro *ADCEMUX* que elige el evento que dispara el comienzo del muestreo. Pueden utilizarse varios tipos de eventos como son un timer, PWM, muestreo continuo u otras opciones, aunque por defecto se activa mediante software. Para utilizar el evento por software se escriben ceros en los bits [15:12] (Cod. 3-16). Para iniciar el muestreo se escribe un 1 en el registro *ADCPSSI*.

```
/*El evento que activa el muestreo*/
ADC0_EMUX_R &= ~0xF000;
```

Codigo 3-16 Definición del tipo de evento que dispara la toma de muestras. Fuente: elaboración propia.

Para seleccionar el canal de entrada se utiliza el registro *ADCSSMUXn*. El muestreo se realiza por la entrada *AIN0* (Cod. 3-17) que corresponde al pin 3 del puerto E.

```
/*Seleccionar el canal AIN0 para realizar el muestreo*/
  ADC0_SSMUX3_R = 0;
```

Código 3-17 Selección del canal por donde se realiza el muestreo. Fuente: elaboración propia.

El registro *ADCSSTL3* contiene la información de configuración para una muestra utilizando el secuenciador SS3. Este registro es de 4-bit, el bit 1 *END0* indica el fin de la secuencia y debe escribirse un 1 ya que sólo se está tomando una sola muestra que es a la vez la primera y la última. También es necesario escribir en el bit 2 (Cod. 3-18), *IE0*, para que al terminar la toma de la muestra se genere una interrupción. Una vez se configura este registro se puede habilitar el secuenciador.

```
/*Configurar el secuenciador de muestreo y habilitar el secuenciador 3*/
ADC0_SSCTL3_R |= 6;
ADC0_ACTSS_R |= 8;
```

Código 3-18 Configuración del secuenciador de muestreo y activación. Fuente: elaboración propia.

Para iniciar la toma de muestras desde el tercer secuenciador, SS3, se escribe un 1 en el bit 4 del registro *ADCPSSI*. Una vez obtenida la muestra se puede leer mediante del registro *ADC0_SSFIFO3_R*. Para preparar la siguiente muestra se debe escribir un 1 en el registro *ADCISC* que permite borrar la condición de interrupción del secuenciador de muestreo.

3.3.3 Cliente UDP

La API SimpleLink permite crear sockets basados en los sockets de Berkeley. Los sockets son un mecanismo para la comunicación entre procesos que se encuentran en distintas máquinas. Para lograr esta comunicación

sólo necesitamos una dirección IP y un puerto como identificador. Sin embargo, la desventaja de este protocolo sin conexión es no poder saber si los datos enviados llegan a su destinatario. A pesar de ello, tiene las ventajas de que la latencia es reducida, la implementación es más sencilla y el overhead es menor que en TCP. En caso de querer comprobar errores u obtener un acuse de recibo pueden implementarse estas funcionalidades a nivel de aplicación. Con UDP no existe el problema de framing o saber cuándo termina un mensaje lo cual es muy conveniente para una aplicación como esta donde cada mensaje con la información necesaria se incluye en el payload del datagrama UDP. Los sockets UDP en modo cliente requieren tres funciones (Tab. 3-4).

Función	Número de parámetros	Tarea
sl_Socket	3	Crear un socket
sl_SendTo	6	Enviar datos al destinatario
sl_Close	1	Cerrar el socket

Tabla 3-4 Descripción de las funciones para crear un socket UDP. Fuente: elaboración propia.

Para el cliente UDP de la aplicación de domótica, la secuencia al utilizar sockets UDP comienza con la creación del socket y el envío de datos que se repite mediante un bucle infinito.

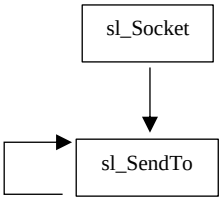

Figura 3-33 Flujo del cliente UDP. Fuente: elaboración propia.

La función *sl_Socket* permite crear el socket. Esta función toma tres argumentos donde el primero de ellos es el protocolo de Internet. Para utilizar IPv4 se utiliza la macro *SL_AF_INET*. El segundo parámetro define la semántica de la comunicación que es *datagram socket* bajo la macro *SL_SOCK_DGRAM*. El tercer y último parámetro indica el protocolo de transporte para el socket que es UDP y se utiliza un 0 como argumento ya que la función escoge el protocolo correspondiente a partir de los dos primeros parámetros. Si no hay errores, la función devuelve un entero de 16-bit positivo que se utiliza como identificador del socket o socket handle. Si hubiese un error, la función devuelve un número negativo.

Para enviar los datos al destinatario se utiliza la función *sl_SendTo* que toma 6 argumentos. El primero es el socket handle que obtenemos de *sl_Socket*. El siguiente es el mensaje que vamos a enviar que consiste en array de tipo *char. El tercer parámetro es el tamaño del mensaje que hemos indicado en el segundo parámetro. El cuarto parámetro es un flag que no se utiliza con UDP por lo que se inicializa a 0. El quinto y sexto parámetro tienen que ver con la dirección del destinatario y el tamaño de la misma respectivamente. Esta dirección se codifica en una estructura de tipo *SlSockAddrIn_t* que se encuentra definida en el fichero *socket.h*. La función devuelve el número de bytes enviados al destinatario.

Se ha utilizado el 19% de la memoria RAM y el 15 % de la memoria Flash 1 disponible en la plataforma Tiva C Series Launchpad para realizar esta implementación.

3.4 Servidor HTTP

El servidor HTTP[4] recibe por un puerto peticiones POST con la temperatura y el identificador que guarda en una base de datos. Por otro puerto recibe peticiones GET y emite eventos desde un socket a un navegador que permite visualizar la temperatura en tiempo real mediante un gráfico (Fig. 3-34).

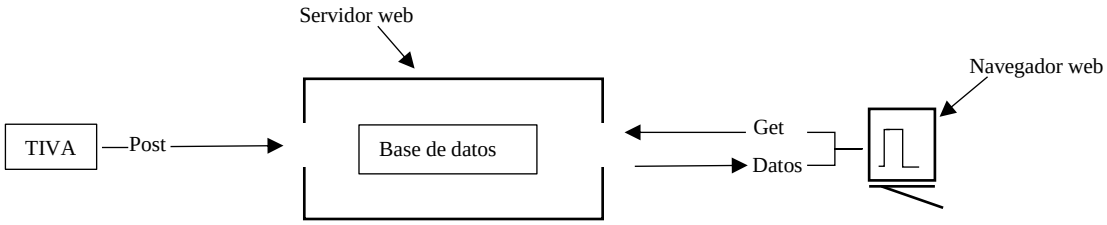

Figura 3-34 Secuencia de entrada y de salida de datos desde el servidor HTTP. Fuente: elaboración propia.

El servidor web se implementa en Node.js y utiliza tanto los módulos *express* como *http* para gestionar las peticiones GET y POST respectivamente. Los eventos del socket, que pertenece al módulo *socket.io* se emiten (Cod. 3-18) cada vez que el servidor recibe datos mediante POST. Estos eventos los captura la página web *index.html* para a continuación pintar el gráfico que se va actualizando el tiempo real.

```
/*Lado servidor*/
io.sockets.emit('temp', json);

/*Lado cliente*/
socket.on('temp', function(data){});
```

Código 3-19 Emisión y recepción de datos que permite mostrar en un gráfico a través de un navegador web. Fuente: elaboración propia.

Cada vez que hay nuevos datos disponibles el servidor envía un mensaje mediante el socket al navegador web que captura para a continuación ejecutar la función callback que pinta los datos en la página web mediante la librería *Chart.js*.

La petición POST de la Tiva S Series Launchpad contiene los datos de temperatura y el identificador de la habitación. Estos datos que llegan en formato JSON son de tipo string y se convierten a un objeto JSON mediante un parser para ser guardados en la base de datos MongoDB. El servidor HTTP gestiona las escrituras en la base de datos de forma asíncrona por lo que puede que no se inserten los datos en el orden de llegada. Por ello, se incluye un campo adicional con un timestamp (Cod. 3-20) de precisión en milisegundos.

```
/*Timestamp junto con los datos*/
var jsondata = JSON.parse('{' + '"Timestamp Hours:Min:Sec:Ms":' +
""+str+""+ ',' + '"Room Id" :' + objetoDatos.id + ','+ '"Temp" :' +
objetoDatos.temp + '}');
```

Código 3-20 Campo adicional con timestamp. Fuente: elaboración propia.

[4] Código disponible en https://drive.google.com/drive/folders/1lqTowkURhjRuFvISLLBFYT3_KaffEg-Y?usp=sharing bajo licencia GNU.

En el gráfico también se muestran los datos sobre la temperatura con precisión en milisegundos en intervalos que se actualizan cada 15 muestras (Fig. 3-34A).

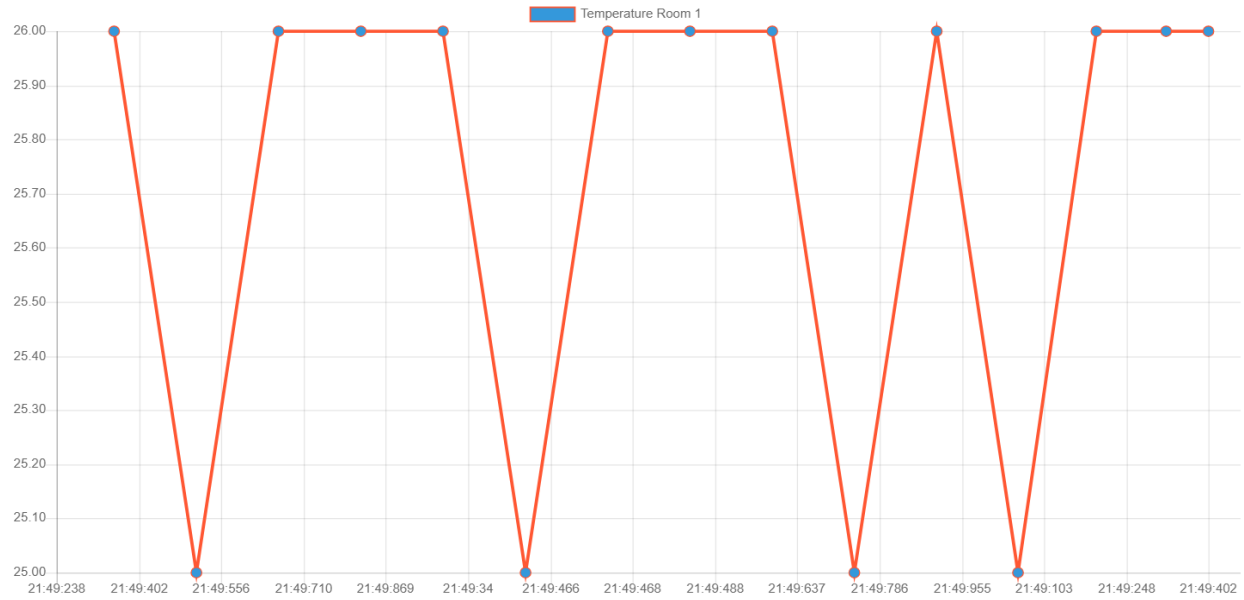

Figura 3-34A Gráfico de temperatura para la habitación 1.

Los datos se almacenan en una colección llamada *roomdata* dentro de la base de datos *tfm*. Estos datos se pueden consultar tanto por línea de comandos como utilizando la interfaz gráfica MongoDB Compass (Fig. 3-34B).

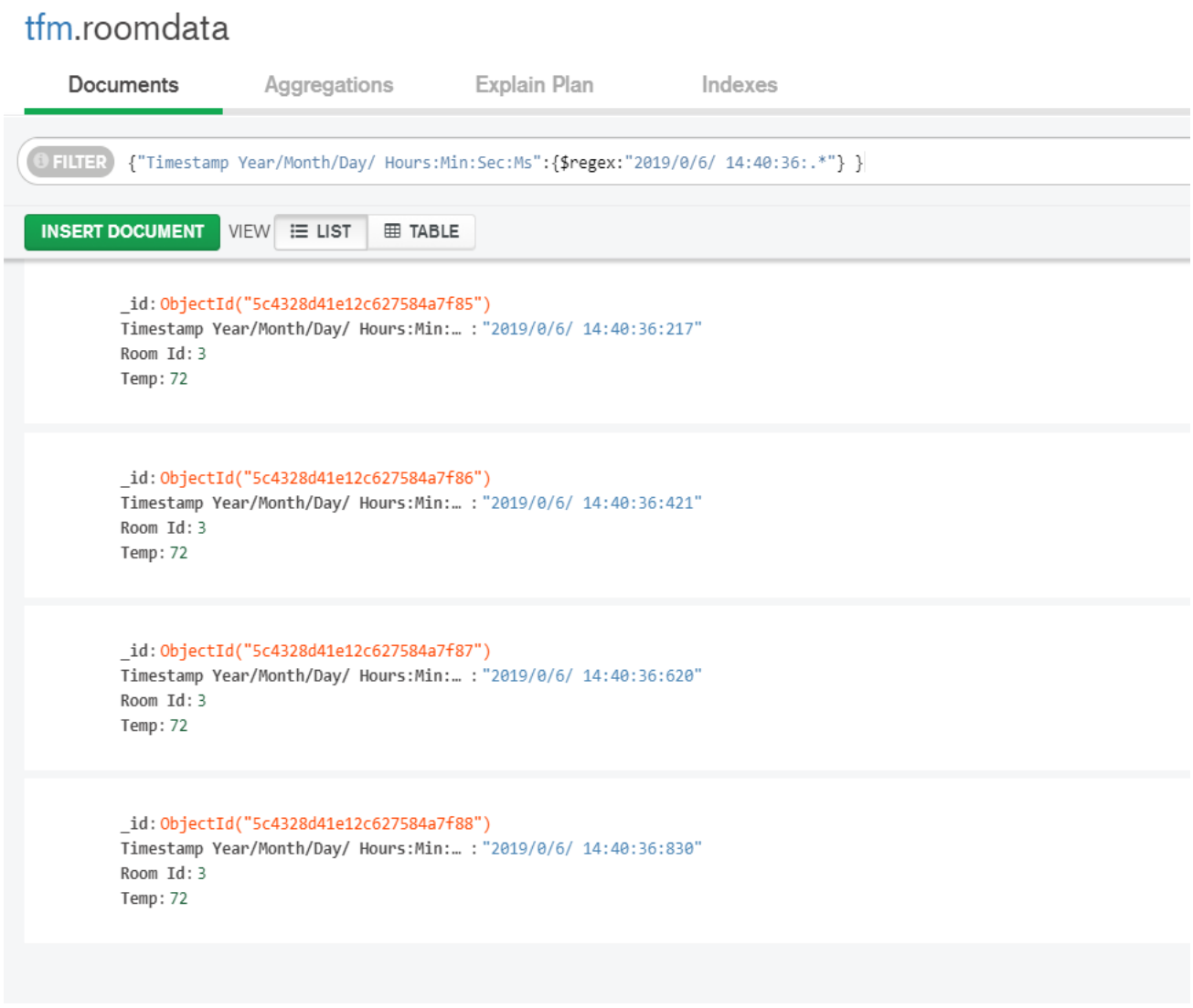

Figura 3-34B Consulta a la base de datos mediante MongoDB Compass.

4 Experimentos

En este apartado se describen las pruebas realizadas para observar el funcionamiento del sistema bajo distintos escenarios.

4.1 Tiempo de transmisión

Durante la ejecución de la aplicación es necesario conocer cuál es el tiempo de transmisión desde que se envían los datagramas UDP desde la plataforma Tiva C Series Launchpad hasta que se reciben en la plataforma MSP. Para realizar este tipo pruebas primeramente se va a utilizar una disposición donde dos plataformas (Tiva C Series Launchpad y MSP430) están conectadas a un mismo PC (Fig. 4-1) que acceden a un AP para intercambiar paquetes UDP.

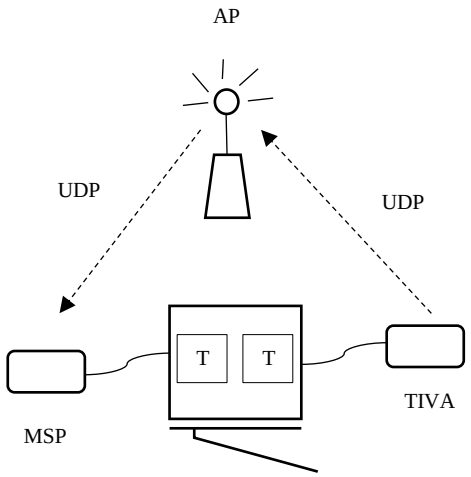

Figura 4-1 Disposición de los microcontroladores para realizar pruebas sobre PRR. Fuente: elaboración propia.

Se utilizan dos emuladores de terminales Tera Term. Estas terminales disponen de una opción para crear logs con un timestamp. Además, la API SimpleLink tiene dos funciones, *CLI_Write* y *Report*, que permiten enviar datos a estas terminales a medida que se va ejecutando el código. Al estar ambas plataformas conectadas al mismo ordenador se obtiene una precisión del timestamp en milisegundos.

Durante la prueba la función *HttpUdpClient* implementada en la plataforma Tiva C Series Launchpad envía datagramas UDP a la función *BsdUdpServer* implementada en la plataforma MSP430 con el identificador de la habitación y la temperatura.

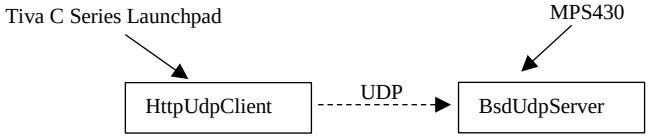

Figura 4-2 Envío de datgramas UPD desde la función HttpUdpClient a la función BsdUdpServer. Fuente: elaboración propia.

Para distinguir entre todos lo datagramas que se reciben durante la realización de la prueba es necesario incluir un identificador (Cod. 4-1) en los datos que envía el emisor.

```
/*Utilización de contador como identificador*/
uBuf.BsdBuf[0] = iden++;
sprintf(str,"%d",uBuf.BsdBuf[0]);
CLI_Write(str);
CLI_Write("\r\n");
Status = sl_SendTo(SockID, uBuf.BsdBuf, BUF_SIZE, 0, (SlSockAddr_t *)&Addr, AddrSize);
```

Código 4-1 Se asigna un identificador único a cada datagrama UDP. Fuente: elaboración propia.

Para ello se utiliza un contador que se va incrementando y que reemplaza al número de habitación como identificador. El contador es un número de 1 byte por lo que se generan 256 identificadores distintos. La prueba por tanto consiste en tomar las muestras sabiendo que cada vez que su supera el número de 255 el identificador se inicializa nuevamente a 0. Esto permite que las modificaciones sobre el código original sean mínimas ya que el miembro de la estructura que contiene el identificador de la habitación es un entero de 1 byte sin signo. Una vez se le asigna a la estructura de la habitación el contador como identificador se crea un string utilizando la función *sprintf* que se envía al emulador de terminal mediante la función *CLI_Write*. El emulador de terminal le asigna un timestamp y a continuación la función *HttpUdpClient* envía el datagrama UDP.

```
/*Muestra tomada del log de emisor*/
[2018-12-30 23:21:34.045] 3
```

Figura 4-3 Timestamp junto al identificador tal como se escribe en el log del emisor. Fuente: elaboración propia.

El log (Fig. 4-3) muestra el timestamp que se obtiene utilizando la hora local del ordenador donde se encuentran conectadas ambas plataformas y tiene una precisión de milisegundos. Con la información del log se puede por un lado saber cuáles datagramas se han perdido y por otro saber con precisión de milisegundos el tiempo de transmisión a través de Wi-Fi del datagrama UDP.

El reloj de la plataforma Tiva C Series Launchpad funciona a una frecuencia de 50 MHz. El IDE Code Composer Studio permite utilizar un reloj de profiling para conocer cuántos ciclos de reloj tarda en ejecutarse un fragmento de código. Utilizando el reloj de profiling transcurren 5,287 ciclos de reloj aproximadamente desde que se llama a la función *sprintf*, hasta que se envía el datagrama mediante la función *sl_SendTo*. El tiempo total por cada ciclo de reloj a 50 MHz es de 20 nanosegundos o 1/50 MHz. Se puede calcular los milisegundos que tarda en ejecutarse el fragmento de código como 20 nanosegundos * 5,287 ciclos que son 0.10 ms de tiempo de ejecución. Por otro lado, el tiempo total de ejecución de la función *HttpUdpClient* de la plataforma Tiva C Series Launchpad en su totalidad es de 135,493 ciclos de reloj * 20 nanosegundos lo que equivale a 2.7 ms. Esta es la frecuencia aproximada con la que se envían los paquetes UDP a la plataforma MSP.

El receptor del datagrama UDP es la plataforma MSP que implementa la función *BsdUdpServer* la cual recibe el datagrama mediante la llamada a la función *sl_RecvFrom.* Una vez recibido el datagrama UDP, se envía al

emulador del terminal (Cod. 4-2) junto al timestamp mediante la función *Report*. Esta función es similar a *CLI_Write*. Es en este momento cuando se le asigna un timestamp y se almacena en el log.

```
/*Enviar al emulador de terminal y escribir en el log*/
Status = sl_RecvFrom(SockID, &uBuf.BsdBuf,BUF_SIZE, 0, (SlSockAddr_t *)&Addr, (SlSocklen_t*)&AddrSize);
if(Status > 0 )
        Report("[ROOM VALUE] ID: %d Temp: %d\n\r",uBuf.BsdBuf[0],uBuf.BsdBuf[1]);
```

Código 4-2 La función Report envía los datos recibidos al emulador de terminal. Fuente: elaboración propia.

El timestamp (Fig. 4-4) tiene el mismo formato que el que se encuentra en el log de la plataforma Tiva C Series Launchpad.

```
/*Muestra en el log del receptor*/
[2018-12-30 23:21:34.763] [ROOM VALUE] ID: 3 Temp: 76
```

Figura 4-4 Timestamp junto al identificador tal como se escribe en el log del receptor. Fuente: elaboración propia.

La frecuencia del reloj de la plataforma MSP es de 25 MHz o 1/25 MHz que son 40 nanosegundos. Se utilizan 749,793 ciclos de reloj aproximadamente desde que se recibe el datagrama mediante la función *sl_RecvFrom* hasta que se ejecuta la función *Report* que asigna el timestamp dentro de la función *BsdUdpServer*. En tiempo equivale a 40 nanosegundos * 749,793 ciclos que son 30 ms aproximadamente. La función *BsdUdpServer* en su totalidad tarda 40 nanosegundos * 1,267,390 ciclos de reloj que equivale a 50 ms aproximadamente. Por tanto, para compensar esta diferencia entre las dos funciones se va a utilizar un delay en la función *HttpUdpClient* entre cada envío de datagrama UDP. Se puede asumir que el sistema es capaz de gestionar hasta 1000/ 50 que son 20 datagramas por segundo. Por ello, se va a probar con un delay de 51 ms que es 1 ms más de lo que tarda en ejecutarse la función *BsdUdpServer*.

Por tanto, la primera prueba consiste en enviar 100 datagramas UDP con un buffer de 2 bytes y con un delay de 51 ms (Cod. 4-3) entre cada datagrama desde la plataforma Tiva C Series Launchpad a la plataforma MSP[5] (Tab. 4-1). El propósito de la prueba es cuantificar tanto el número de datagramas perdidos como el tiempo de transmisión de los datagramas de una plataforma a otra.

Número de datagramas UDP	Delay (ms)	Buffer (Bytes)
100	51	2

Tabla 4-1 Valores de la primera prueba para conocer la pérdida de información y tiempo de transmisión. Fuente: elaboración propia.

El buffer es de 2 bytes debido ya que sólo se envían dos bytes y por tanto no es necesario un buffer mayor.

[5] El router a utilizar como AP es un Comtrend modelo WAP-5813n con el firmware P401-402TLF-C05_R06.

```
/*Delay equivalente a 51 ms */
for(i=0; i< 68;i++)
    for(j=0;j<3180;j++)
    {}
```

Código 4-3 Delay de 51 ms implementado en la plataforma Tiva C Series Launchpad. Fuente: elaboración propia.

El delay utiliza 2,596,488 ciclos de reloj * 20 nanosegundos que equivale a 51 ms aproximadamente. Durante la prueba no se han podido procesar 2 datagramas. Esta pérdida de información supone que el índice de recepción ha sido del 98%.

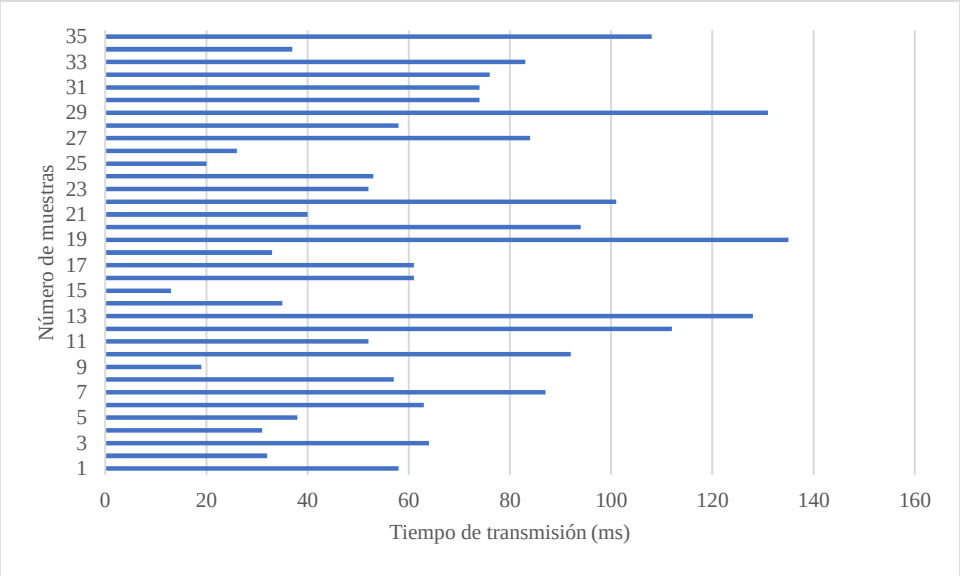

Figura 4-5 Tiempo transcurrido desde la emisión hasta la recepción 35 muestras durante la prueba 1. Fuente: elaboración propia.

La media del tiempo de transmisión para 35 muestras de esta primera prueba ha sido de 65.20 ms (Fig. 4-5). Por otro lado, la plataforma MSP ha recibido 18.87 datagramas por segundo lo que equivale a 1 datagrama cada 51 ms aproximadamente. Un número superior a 20 datagramas UDP supone que, aunque lleguen los datagramas a la plataforma MSP, la función *BsdUdpServer* no será capaz de procesarlos a tiempo y se perderán. Por tanto, un delay de 51 ms es la base sobre la que calcular los delays en relación al número de placas a utilizar teniendo en cuenta que no debe sobrepasarse el umbral de 20 datagramas por segundo en total para obtener el mayor índice de recepción posible.

La segunda prueba (Tab. 4-2) consiste en enviar 100 datagramas UDP manteniendo el buffer de 2 bytes y con un delay de 102 ms o 51 ms *2 (Cod. 4-3) entre cada datagrama desde la plataforma Tiva C Series Launchpad a la plataforma MSP.

Número de datagramas UDP	Delay (ms)	Buffer (Bytes)
100	102	2

Tabla 4-2 Valores de la segunda prueba para conocer la pérdida de información y tiempo de transmisión. Fuente: elaboración propia.

Un delay de 102 ms equivale a 5,116,368 ciclos de reloj a 20 nanosegundos por ciclo (Cod. 4-4).

```
/*Delay equivalente a 102 ms */
for(i=0; i<134;i++)
   for(j=0; j<3180;j++){}
```

Código 4-4 Delay de 102 ms implementado en la plataforma Tiva C Series Launchpad. Fuente: elaboración propia.

En esta segunda prueba no se ha podido procesar 1 datagrama. Esta pérdida de información supone que el índice de recepción ha sido del 99.00% (Fig. 4-6). La media del tiempo de transmisión para esta segunda prueba ha sido de 67.91 ms.

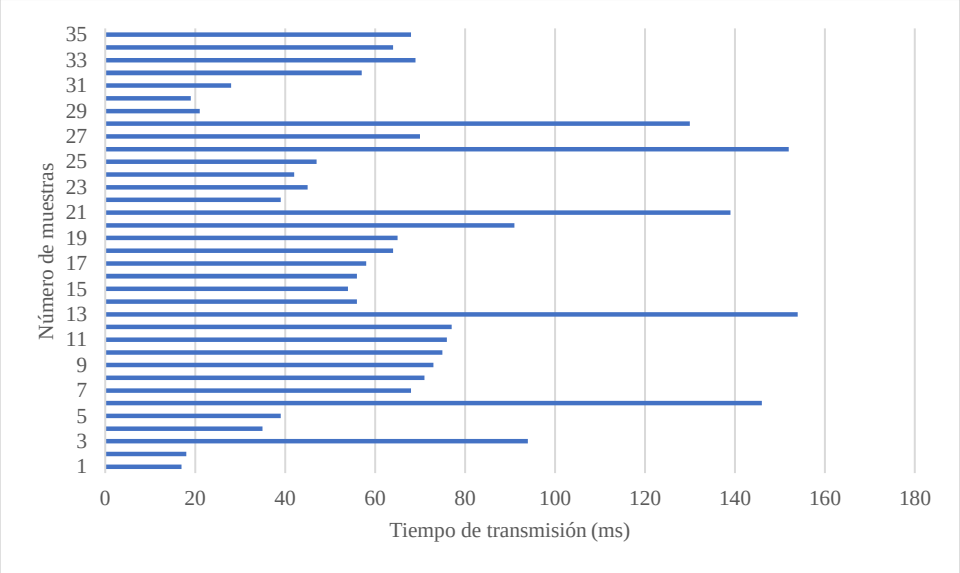

Figura 4-6 Tiempo transcurrido desde la emisión hasta la recepción para 35 muestras durante la prueba 2. Fuente: elaboración propia.

La tercera prueba (Tab. 4-3) consiste en enviar 100 datagramas UDP manteniendo el buffer de 2 bytes y un delay de 152 ms o 51 ms * 3 + 1 (Cod. 4-4) entre cada datagrama.

Número de datagramas	Delay (ms)	Buffer (Bytes)
100	152	2

Tabla 4-3 Valores de la tercera prueba para conocer la pérdida de información y tiempo de transmisión. Fuente: elaboración propia.

Un delay de 152 ms equivale a 7,636,248 ciclos de reloj a 20 nanosegundos por ciclo (Cod. 4-5).

```
/*Delay equivalente a 152 ms */
for(i=0; i<200;i++)
   for(j=0;j<3180;j++){}
```

Código 4-5 Delay de 152 ms implementado en la plataforma Tiva C Series Launchpad. Fuente: elaboración propia.

Durante la tercera prueba no se han perdido datagramas por lo que el índice de recepción ha sido del 100% (Fig 4-7). La media del tiempo de transmisión para esta tercera prueba ha sido de 88.31 ms.

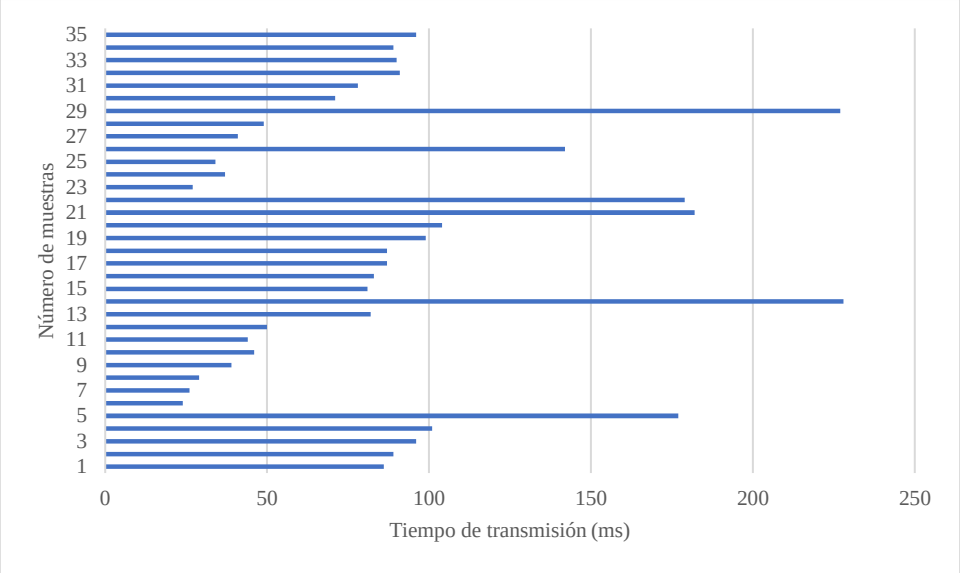

Figura 4-7 Tiempo transcurrido desde la emisión hasta la recepción para 35 muestras durante la prueba 3. Fuente: elaboración propia.

La cuarta prueba (Tab. 4-3) consiste en enviar 100 datagramas UDP por tanto manteniendo el número de datagramas y con el mismo tamaño de buffer de 2 bytes y un delay de 204 ms o 51 ms * 4 (Cod. 4-4) entre cada datagrama.

Número de datagramas	Delay (ms)	Buffer (Bytes)
100	204	2

Tabla 4-4 Valores de la cuarta prueba para conocer la pérdida de información y tiempo de transmisión. Fuente: elaboración propia.

La media del tiempo de transmisión para esta cuarta prueba ha sido de 105.08 ms y no se han perdido datagramas lo que supone índice de recepción del 100% (Fig. 4-8).

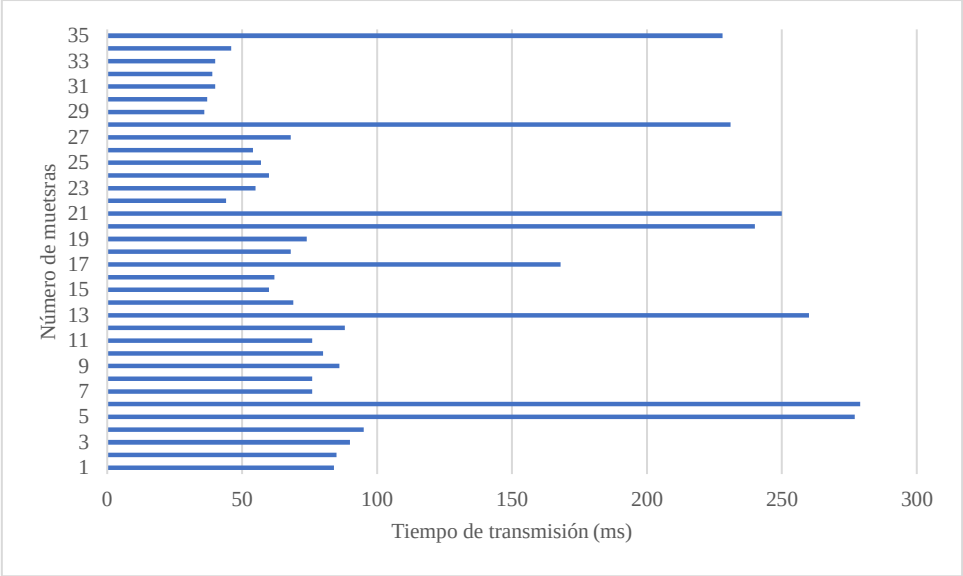

Figura 4-8 Tiempo transcurrido desde la emisión hasta la recepción para 35 muestras durante la prueba 4. Fuente: elaboración propia.

El delay de esta cuarta prueba tarda 10,232,489 ciclos de reloj a 20 nanosegundos por ciclo (Cod. 4-6).

```
/*Delay equivalente a 204 ms */
for(i=0; i<268;i++)
   for(j=0;j<3180;j++
{}
```

Código 4-6 Delay de 204 ms implementado en la plataforma Tiva C Series Launchpad. Fuente: elaboración propia.

Si se comparan los resultados de las cuatro pruebas (Tab. 4-5) se puede ver que con el delay de 51 ms se obtiene mayor velocidad de transmisión a 65.20 ms de media por datagrama.

Número de datagramas UDP	Delay (ms)	Indice de recepción	Tiempo de transmisión (ms)	Buffer (Bytes)
100	51	98%	65.20	2
100	102	99%	67.91	2
100	152	100%	88.31	2
100	204	100%	105.80	2

Tabla 4-5 Comparativa entre las 4 pruebas. Fuente: elaboración propia.

El índice de recepción en las cuatro pruebas es similar. Se realizaron pruebas adicionales por debajo del umbral de 51 ms de delay donde la perdida de información empezó a incrementar considerablemente. Para 26 ms de delay la pérdida fue del 7.5 % y para 25 ms de delay fue de 9.79%. Con 24 ms de delay ya pasó a una pérdida de 12% aproximadamente.

Para comprobar que el delay de 152 ms es el más eficaz de los estudiados si se utilizan tres placas emisoras, se realizó una prueba con tres plataformas Tiva C Series Launchpad donde cada una de las plataformas se programó para que enviase 30,000 datagramas. Por tanto, la plataforma MSP430 debía gestionar 90,000 datagramas. Con un delay de 51 * 3 o 152 ms aproximadamente si se pudo gestionar este número de datagramas de forma eficaz con una pérdida de 447 datagramas UDP lo que supone sólo un .5 % de pérdida. El resto de

delays no pudieron gestionar este número de datagramas. Por tanto, los resultados de estas pruebas muestran que únicamente el delay de 152 ms se mantiene con un buen rendimiento en casos de uso reales con tres sensores conectados a la vez con un tiempo de transmisión de 88 ms de media. El resto de delays se han mostrado insuficientes ante un número simultáneo de datagramas debido a que se ha llenado el buffer de la plataforma MSP430 demasiado rápido. Sin embargo, esto no ocurre con el delay de 152 ms ya que equivale a 6 datagramas por segundo. Es decir, si se utilizan 3 plataformas Tiva C Series Launchpad y cada una envía 6 datagramas por segundo significa que se envían 18 datagramas por segundo lo cual entra dentro del rango aceptado por la función *BsdUdpServer* de la plataforma MSP. Por tanto, el delay de 152 ms se mantiene como la opción más estable dentro de las analizadas para tres dispositivos Tiva C Series Launchpad enviando datos a la vez por ello es el delay que se va a utilizar en las siguientes pruebas.

4.2 Estado del enlace inalámbrico

Para conocer el estado del enlace inalámbrico entre las plataformas es necesario saber los paquetes que llegan desde el emisor al receptor y cuantos son descartados mediante el índice PRR (Packet Reception Ratio). Para ello, la API SimpleLink tiene una serie de funciones que permiten recopilar datos estadísticos sobre los paquetes. Con estas funciones se puede obtener el índice PRR para conocer cuántos paquetes se han recibido sobre los enviados y cuantos han sido descartados. Las estadísticas se toman de la capa de acceso a red. Los paquetes descartados son aquellos cuyos FCS (Frame Check Sequence) no coincide con el CRC (Cyclic Redundancy Check). Esto significa que algún dato se ha perdido y paquete debe volver a enviarse. Además, el PRR permite conocer el lugar idóneo donde debe posicionarse la placa ya que a mayor PRR menor será el consumo energético del chip de red.

Para utilizar el módulo de datos estadísticos sobre el estado del enlace se declara una estructura de tipo *SlGetRxStatResponse_t* en el receptor MSP430. Esta estructura contiene nueve miembros. Para obtener el índice PRR se utilizan los miembros *ReceivedValidPacketsNumber* y *ReceivedFcsErrorPacketsNumber* donde el número total de paquetes recibidos es la suma de los resultados almacenados en estos dos miembros. Por tanto, *ReceivedValidPacketsNumber /(ReceivedValidPacketsNumber + ReceivedFcsErrorPacketsNumber)* permite obtener el número de paquetes recibidos o PRR.

Una vez se ha declarado esta estructura se llama a la función *sl_WlanRxStatStart* que no toma parámetros. Esta llamada indica el inicio de la recopilación de los datos estadísticos. Para almacenar los datos se realiza una llamada a la función *sl_WlanRxStatGet* que toma como argumento la dirección de la estructura de tipo *SlGetRxStatResponse_t* y una serie de flags que no se usan por lo que se pasa un 0. Para finalizar la toma de estadísticas se realiza una llamada a la función *sl_WlanRxStatStop*. Estas llamadas se realizan dentro del bucle infinito de la función *BsdUdpServer*.

```
/*Prueba PRR*/
if(uBuf.BsdBuf[0] == 253){mr1 = done;}
else if(uBuf.BsdBuf[0] == 254){mr2  = done;}
else if(uBuf.BsdBuf[0] == 255){mr3  = done;}
if(mr1 == done && mr2 == done && mr3 == done){
  Status = sl_WlanRxStatGet(&rxStatResp,0);
  Report("Packets that been received OK: %d\n\r", rxStatResp.ReceivedValidPacketsNumber);
  Report("Packets dropped: %d\n\r",rxStatResp.ReceivedFcsErrorPacketsNumber);
  Report("Average RSSI for packets received: %d",rxStatResp.AvarageDataCtrlRssi);
  Status = sl_WlanRxStatStop();
 exit(1); }
```

Código 4-7 Llamada a las funciones que permiten obtener datos estadísticos del estado del enlace. Fuente: elaboración propia.

La primera prueba consiste en enviar a la plataforma MSP430 300 datagramas UDP por cada una de las placas Tiva C Series Launchpad para un total de 900 datagramas UDP. Para ello, se utiliza un delay 152 ms para lograr un ritmo de 18 datagramas por segundo combinando las tres plataformas. La función *BsdUdpServer* es un bucle infinito, por lo que para conocer cuando los dispositivos Tiva Series Launchpad han terminado de enviar los mensajes, cada plataforma Tiva C Series Launcpad al terminar de enviar los 300 datagramas UDP envían a continuación un identificador (Cod. 4-8) que la plataforma MSP430 captura (Cod. 4-7). Una vez se obtienen los tres identificadores se muestran los datos estadísticos en el emulador del terminal, se realiza una escritura en el log y se finaliza.

```
if(datagram_counter == 300
{
   uBuf.BsdBuf[0] = 253;
}
```

Código 4-8 Se asigna un identificar notificando el fin de la transmisión de datagramas. Fuente: elaboración propia.

La disposición de la prueba consiste en colocar las tres plataformas Tiva C Series Launchpad, a distintas distancias *d* metros del AP sin obstáculos y obtener el número de datagramas recibido, el PRR además del tiempo de recepción de cada datagrama. La plataforma MSP430 se coloca a una distancia de 1 m del AP en línea recta (Fig 4-9). Se realizan 5 mediciones para cada distancia y se toma la media.

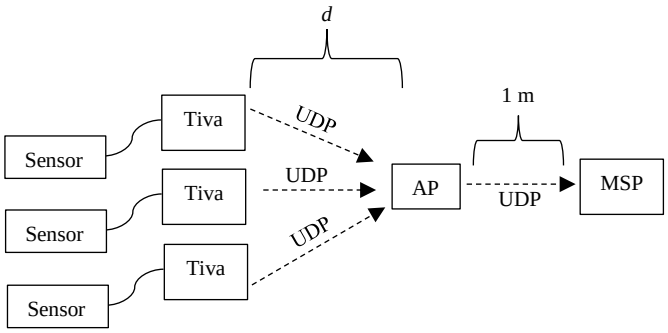

Figura 4-9 Disposición del emisor Tiva C Series Launchpad y el receptor MSP para la primera prueba. Fuente: elaboración propia.

También es útil conocer la media de la fuerza de la señal mediante un valor RSSI expresado en decibelios/vatios para el conjunto de los paquetes. Esta información se obtiene de la antena y permite conocer cuál es la posición idónea para colocar el dispositivo. Mientras mayor sea el valor RSSI más fuerza tiene la señal. Por lo general este valor es negativo. Para realizar esta prueba se han tomado 4 muestras por cada valor de distancia.

Datagramas UDP Enviados	Distancia *d* (m)	Media de paquetes recibidos	Media de paquetes descartados	PRR (%)	Media de RSSI (dBm)	Media de datagramas UDP perdidos	Recepción de datagramas UDP (%)	Media de datagramas por segundo
900	1.50	4,006	2,800	60.35	-33	3	99.67	18.86
900	3	3,860	6,733	36.52	-32	1	99.86	18.92
900	4.5	4,070	21,016	16.27	-36	5	99.42	18.72
900	6	3,619	13,631	29.33	-37	30	96.61	18.59
900	7.5	3,887	5,406	51.87	-30	4	99.50	18.99
900	9	4,112	1,903	71.30	-31	3	99.67	18.76
900	10.5	3,916	769	84.17	-32	2	99.75	18.52

Tabla 4-6 Resultados de la prueba 1 para distintas distancias. Fuente: elaboración propia.

Los resultados (Tab. 4-6) indican que la media de datagramas UDP recibidos ha sido de 99.21 % y la media del índice PRR de 49.97 %. El índice PRR más bajo tuvo lugar a 4.5 m (Fig. 4-10) del AP y el más alto a 10.5 m del AP. La mayor pérdida de datagramas tuvo lugar a 6 m y la menor a 3 m. La media para la señal RSSI ha sido de -33 dBm.

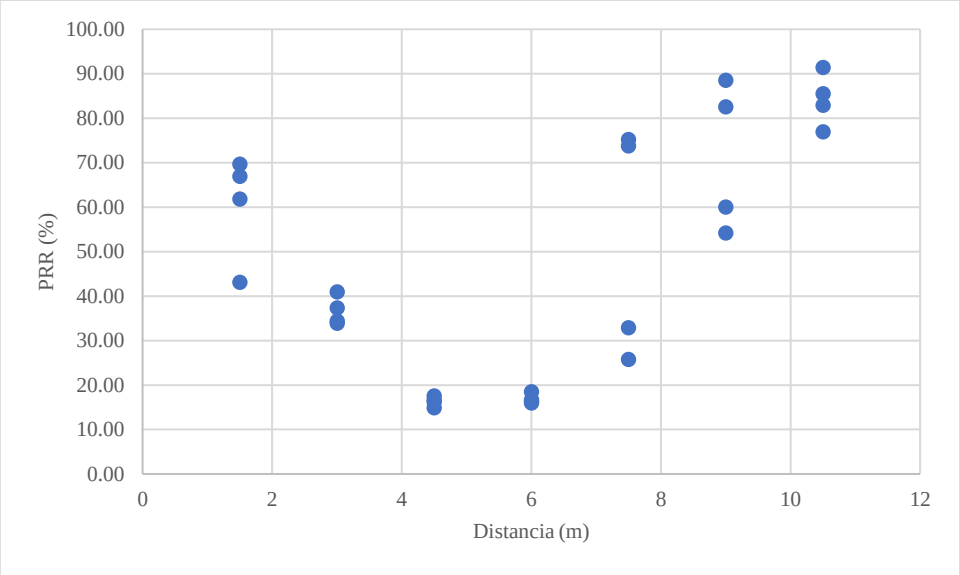

Figura 4-10 Índice PRR en relación a la distancia entre los dispositivos Tiva y el AP para la prueba 1. Fuente: elaboración propia.

La segunda prueba consiste en repetir la primera pero esta vez con una puerta de madera (Fig. 4-11) cerrando el paso entre el AP y el emisor a una distancia de 70 cm del AP. Se realizan 5 mediciones para cada distancia y se toma la media.

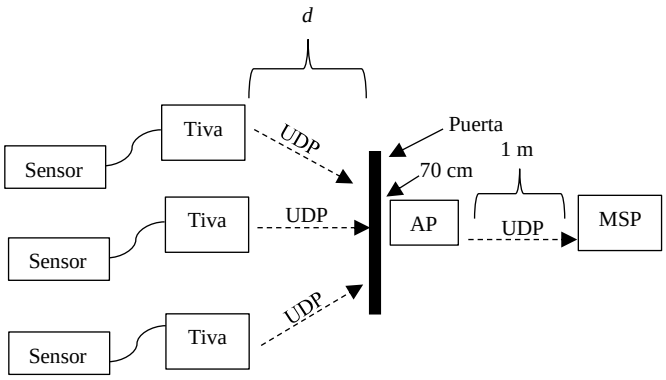

Figura 4-11 Disposición para la segunda prueba con obstáculo. Fuente: elaboración propia.

Los resultados (Tab. 4-7) muestran que la media de datagramas UDP recibidos ha sido de 99.30 % y la media del índice PRR ha sido de 47.71 %. El índice PRR más bajo tuvo lugar a 4.5 m (Fig. 4-12) del AP y el más alto a 9 m del AP. La mayor pérdida de datagramas tuvo lugar a 6 m y la menor a 3 m.

Datagramas UDP Enviados	Distancia *d* (m)	Media de paquetes recibidos	Media de paquetes descartados	PRR (%)	Media de RSSI (dBm)	Media de datagramas UDP perdidos	Recepción de datagramas UDP (%)	Media de datagramas por segundo
900	1.50	3,938	5,492	49.47	-32	2	99.72	18.62
900	3	3,652	12,845	23.79	-34	1	99.83	18.99
900	4.5	3,966	20,409	16.91	-35	3	99.67	18.45
900	6	3,905	11,190	27.50	-32	21	97.64	18.59
900	7.5	3,967	4,516	51.49	-30	2	99.50	18.18
900	9	4,221	641	86.91	-32	4	99.53	18.51
900	10.5	3,925	1,827	77.95	-37	7	99.22	18.94

Tabla 4-7 Resultados de la prueba 2. Fuente: elaboración propia.

La media de la señal RSSI para la prueba 2 ha sido de -33 dBm. La menor cantidad de datagramas por segundo se obtuvo a 7.5 m y la mayor a 3 m.

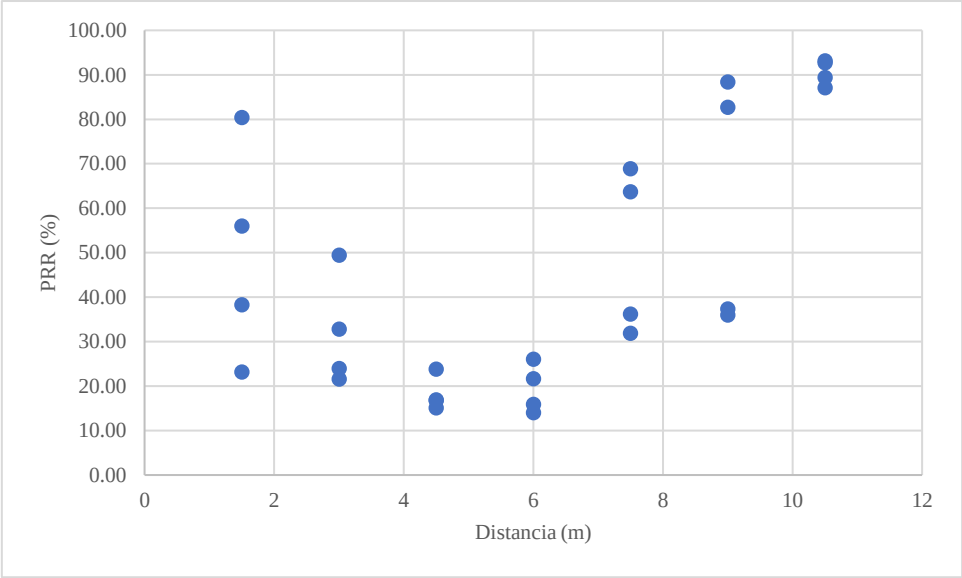

Figura 4-12 Índice PRR en relación a la distancia entre el dispositivo emisor y el AP para la prueba 2. Fuente: elaboración propia.

Si se realiza una comparativa (Fig. 4-13) entre la prueba 1 sin obstáculo y la prueba 2 con una puerta bloqueando la señal entre el AP y las plataformas Tiva C Series Launchpad, se observa que la zona con mayor interferencia para ambas pruebas se encuentra entre los 4 m y los 8 m. Por otro lado, la zona con menor interferencia es la que se encuentra entre los 9 m y los 10.5 m.

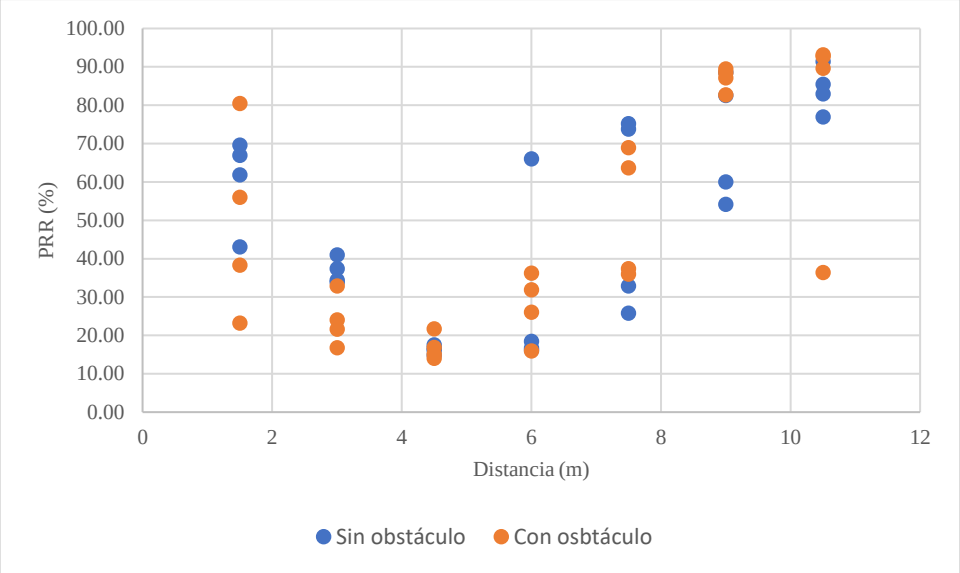

Figura 4-13 Comparativa entre la prueba 1 y la prueba 2. Fuente: elaboración propia.

A tenor de los resultados se aprecia que el estado del enlace inalámbrico fluctúa considerablemente, pero el rendimiento del sistema con o sin obstáculo se mantiene estable con una media de 18.68 datagramas por segundo y una recepción media de datagramas UDP del 99.25 %.

5 Conclusiones

En este trabajo se han estudiado diversas placas de desarrollo de bajo coste y consumo energético. Se ha programado el chip de red CC3100 para permitir conectar a Internet un microcontrolador de ultra bajo consumo energético con tan sólo 8 KB de memoria RAM como el MSP430. Se ha implementado en este microcontrolador un servidor UDP y un cliente MQTT utilizando FreeRTOS. También se ha programado el chip de red CC3100 para utilizarlo en las plataformas Tiva C Series Launchpad. Además, se les ha añadido a los dispositivos Tiva C Series Launchpad sensores para obtener medidas de temperatura. Se ha programado en cada uno de estos dispositivos un cliente UDP y un cliente HTTP para enviar los datos al dispositivo MSP340 y a un servidor web respectivamente. Se ha añadido una base de datos que a través del servidor web permite almacenar la información obtenida de los sensores de temperatura.

Por otro lado, se han realizado una serie de pruebas que han permitido conocer el delay óptimo para sincronizar las funciones de emisión y recepción de los datagramas UDP y minimizar la pérdida de información obteniendo 88 ms de tiempo de transmisión y un índice de recepción de datos superior a 90 % en todas las pruebas realizadas. Además, se ha estudiado el estado del enlace inalámbrico para conocer la recepción de paquetes a nivel de acceso a red, observando que mientras mayor es el PRR por lo general se producen menos pérdidas de datagramas UDP. Por otro lado, un obstáculo como una puerta sí produce una mayor pérdida de datagramas UDP a partir de los 9 metros en las pruebas realizadas. De media el sistema es capaz de gestionar 18 datagramas UDP por segundo con una recepción de datagramas UDP del 99 % y un tiempo medio de transmisión de datagramas UDP de 88 ms. Algunas ampliaciones posibles a este trabajo son la incorporación de un dispositivo que utilice CoAP, la utilización de otros lenguajes de programación y estudiar el rendimiento de HTTP y MQTT.

Bibliografía

[1] P. Lea, The Internet of Things for Architects, Birmingham: Packt Publishing, 2018.

[2] "MQTT," 2014. [Online]. Disponible: https://mqtt.org/. Ultimo Acceso: 8/01/2019

[3] Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP)," 2014. [Online]. Disponible: https://tools.ietf.org/html/rfc7252. Ultimo Acceso: 9/01/2019

[4] A. McEwen and H. Cassimally, Designing the Internet of Things, John Wiley & Sons, 2014.

[5] D. Lora, P. Cerro, A. A. Del Barrio and G. Botella, "Sistema de Seguridad Basado en una Plataforma Heterogénea Distribuida," *Enseñanza y Aprendizaje de Ingeniería de Computadores, 5*, pp. 29-38, 2015. Ultimo Acceso: 8/01/2019

[6] J. Martín and A. A. Del Barrio, "A dirtibuted HW-SW plataform for fireworks," in *In Proceedings of Summer Computer Science Simulation Conference. Article 17*, Montreal, 2016. Ultimo Acceso: 8/01/2019

[7] I. Laclaustra, J. Martín, A. A. Del Barrio and G. Botella, "Sistema domótico distribuido para controlar el riego y el aire acondicionado en el hogar," *Enseñanza y Aprendizaje de Ingeniería de Computadores*, no. 6, pp. 87-101, 2016. Ultimo Acceso: 8/01/2019

[8] J. W. Valvano, Introduction to ARM Cortex Microcontrollers Embedded Systems, Amazon, 2014.

[9] S. Prata, C Primer Plus Sixth Edition, Pearson Education, Inc., 2014.

[10] C. Lozano, Experimentación con el Tiva Launchpad TM4C123G, Amazon.

[11] A. Forster, Introduction to Wireless Sensor Networks, John Wiley & Sons, Inc., 2016.

[12] K. Elk, Embedded Software Development for the Internet Of Things, Amazon , 2016.

[13] B. Dayley, B. Dayley and C. Dayley, Node.js, MongoDB and Angular Web Development, Pearson Education, Inc., 2018.

[14] S. Chen, S. Naimi, S. Naimi and M. A. Mazadi, TI Tiva ARM Programming for Embedded Systems, Amazon, 2017.

[15] J. Casad, TCP/IP, Pearson Education, Inc., 2012.

[16] M. J. D. K. L. Calvert, TCP/IP Sockets in C - Parctical Guide for Programmers, Morgan Kaufmann, 2009.

[17] R. E. Bryant and D. R. O'Hallaron, Computer Systems a Programmer's Perspective Third Edition, Pearson Education, 2016.

[18] C. Bormann, M. Ersue and A. Keranen., Terminology for Constrained-Node Networks [Online]. Disponible: https://tools.ietf.org/html/rfc7228. Ultimo Acceso: 9/01/2019

[19] Y. Bai, Practical Microcontroller Engineering with ARM Technology, John Wiley & Sons, Inc., 2016.

[20] Texas Instruments, Tiva C Series Launchpad Datasheet [Online]. Disponible: http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf. Ultimo Acceso: 5/01/2019

[21] Texas Instruments, MSP430 Datacheet[Online]. Disponible: http://www.ti.com/lit/ds/symlink/msp430f5529.pdf. Ultimo Acceso: 9/01/2019

[22] The Internet Society, Hypertext Transfer Protocol -- HTTP/1.1 [Online]. Disponible: https://tools.ietf.org/html/rfc2616. Ultimo Acceso: 9/01/2019

[23] Texas Instruments, CC3100 User's Guide [Online]. Disponible: http://www.ti.com/lit/ug/swru368b/swru368b.pdf. Ultimo Acceso: 7/01/2019

[24] Texas Instruments, CC3100 Datasheet [Online]. Disponible: http://www.ti.com/lit/ds/symlink/cc3100.pdf. Ultimo Acceso: 3/01/2019

Introduction

Nowadays it is increasingly common for microcontrollers to connect to the internet to publish and consume information. Therefore, the Internet has gone from being an ecosystem where humans publish and consume information to a wider network that includes hundreds of millions of small machines that generate and consume information. This new ecosystem is known as IoT (Internet of Things). IoT is a term coined by Kevin Ashton in 1997 to refer to electronic RFID (Radio Frequency Identification) tags used in supply chain management. These labels were patented by Mario W. Cardullo in 1973 and allow to identify objects without being in the reader's field of vision and without human intervention. In addition, they allow the reader to send data about the object such as the identity of the object and its position through the Internet in real time.

The idea of autonomous communication between machines is prior to IoT and is known as M2M (Machine to Machine). This concept implies that an autonomous device is able to communicate with another autonomous device without the need for human intervention. Therefore, M2M is the precursor of IoT and they are closely related terms. However, one of the main differences between M2M and IoT is that an M2M device can communicate through a non-IP channel as a serial port without having to access the Internet while an IoT device will always access the Internet.

By the year 2021 it is estimated that there will be around 33 billion IoT devices and by the year 2035 it is predicted that the number will exceed one hundred billion devices [1]. The main sectors of modern societies can benefit from IoT. For example, efficiency can be gained with devices that comprehensively monitor the manufacturing process in factories. In addition, IoT can provide greater security detecting possible gas leaks or any other physical phenomenon that jeopardizes the proper functioning of the machinery and security.

More and more consumers are using IoT, especially as part of domotics and smart homes systems, since these applications not only save energy, but also provide comfort and well-being. Stores are also using IoT devices to reduce expenses by tracking inventory by detecting losses and providing data that can be processed to optimize the supply chain.

Currently there are around 500 million of the number of wearable devices used for the monitoring of vital signs. Therefore, the health sector will be one of the great beneficiaries of IOT. A device can monitor a patient who is at home or detect a condition as soon as the first symptoms occur, which can allow immediate treatment with the appropriate medications. Within hospitals, IOT can be used to control medical equipment and other hospital equipment.

The cities are going to be the areas where there will be more and more IOT devices. For example, in the city of Barcelona [1] they have already implemented a system to optimize garbage collection based on the capacity of the containers and the time elapsed since the last time they were emptied. In addition, cities can use IOT devices to monitor pollution levels, improve traffic through smart traffic lights and save energy in street lighting by activating it as needed. Phenomena such as heavy snowfall can be addressed efficiently with IOT devices that indicate which areas have been most affected. IOT can be used to increase street safety through cameras and alerts in case of criminal acts. You can also avoid accidents by monitoring the state of the main infrastructure of the city so that, if necessary, can be repaired or replaced. IOT devices can also be used to detect terrorist threats by aborting possible attacks.

However, a massive deployment of IoT devices leads to an important increase in energy consumption, so it is advisable to use devices that provide the highest possible energy efficiency. There are microcontrollers with very low power consumption modes but the main disadvantage is that they do not support the TCP / IP or have a limited support of the TCP/IP stack because they have great limitations in terms of hardware resources.

Objectives

In this work, through a home automation application to measure temperature in real time, a strategy is proposed that combines hardware and software that allows access to the Internet by microcontrollers with ultra-low energy consumption. It is studied how to incorporate the UDP, HTTP and MQTT protocols to the devices and the use of an RTOS (Real Time Operating System) to achieve greater efficiency in the use of hardware resources. Therefore, the objectives are to study in depth the MQTT protocol and implement an MQTT client and a UDP server using FreeRTOS in a very low energy consumption device. Include devices with greater hardware capacity that use the UDP and HTTP protocols to send temperature data obtained through sensors together with low energy consumption network chips that allow all devices to use TCP / IP protocols and 802.11 standards. Implement a web server using Node.js [2] that together with a database allows to save the information and display it through a real-time graphic through a web browser. Perform experiments to obtain the transmission time of the UDP datagrams, quantify the loss of datagrams and analyse the state of the wireless link.

Conclusions

This thesis studied several low cost development plates and energy consumption. The CC3100 network chip has been programmed to allow an ultra-low power microcontroller to connect to the Internet with only 8 KB of RAM memory, such as the MSP430. A UDP server and an MQTT client using FreeRTOS have been implemented in this microcontroller. The CC3100 network chip has also been programmed to be used on the Tiva C Series Launchpad platforms. In addition, sensors have been added to the Tiva C Series Launchpad devices to obtain temperature measurements. A UDP client and an HTTP client have been programmed on each of these devices to send the data to the MSP340 device and a web server, respectively. A database has been added that together with a web server allows to store the information obtained from the temperature sensors.

On the other hand, a series of tests have been carried out that have allowed to know the optimal delay to synchronize the functions of emission and reception of the UDP datagrams and to minimize the loss of information, obtaining 88 ms of transmission time and an index of data reception. greater than 90% in all tests performed. In addition, the status of the wireless link has been studied to determine the reception of packets at the network access level, noting that the higher the PRR, the less UDP datagram losses usually occur. On the other hand, an obstacle like a wood door does produce a greater loss of UDP datagrams from the 9 meters distance in the tests carried out. On average, the system is able to manage 18 UDP datagrams per second with a UDP datagram reception of 99 % and an average UDP datagram transmission time of 88 ms. Some possible extensions to this work are the incorporation of a device that uses CoAP, the use of other programming languages and studying the performance of HTTP and MQTT.